A SESSION LOGIC FOR RELAXED COMMUNICATION PROTOCOLS

MIRELA-ANDREEA COSTEA

(B.Sc. in Computer Engineering, University Politehnica of Timisoara, Romania)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2017

Supervisor:
Associate Professor Chin Wei Ngan

Examiners:
Professor Joxan Jaffar
Associate Professor Khoo Siau Cheng
Professor Nobuko Yoshida, Imperial College London

DECLARATION

I hereby declare that this thesis is my original work and it has been written by me in its entirety.

I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

Mirela-Andreea Costea

August 2017

ACKNOWLEDGEMENTS

I am grateful to my advisor, Dr. Wei-Ngan Chin, for his careful and kind guidance. His energy, knowledge and passion for technology have inspired me in so many ways. I learned that *persistence* can be tough in research, but then he showed me how to foster it in order to thrive as a researcher and be a stronger version of myself. An advocate of functional languages immersed in designing elegant *abstractions*, he bridged my engineering thinking to that of the current scientist, now trained in higher-order reasoning. He taught me that finding solutions to relevant problems is important, but finding the *neat solutions* even more so. Apart from the many technical things, I also owe him the pleasure of appreciating the renown cuisine of Singapore.

I would like to thank my PhD committee, Dr. Siau-Cheng Khoo, Dr. Joxan Jaffar, and Dr. Nobuko Yoshida for reviewing this dissertation and providing their constructive feedback. Moreover, I would like to mention that the "Principles of Program Analysis" module taught by Dr. Siau-Cheng Khoo has greatly strengthen my fascination for formal reasoning. I would also like to thank Dr. Aquinas Hobor and Dr. Roland Yap for their valuable comments and suggestions during and post my thesis defense.

I would like to thank Dr. Olivier Danvy for sharing his precious thoughts which span across so many fields. As a student, in addition to the technical and pedagogical advices, I greatly benefited from his rigorous feedback on scientific communication; as a friend, I was introduced to fine articles, books, movies and priceless aha moments.

I would like to thank Dr. Cristina David, the friend, colleague, and the role model who has been such a positive companionship since my early days in Singapore. She inspired me to pursue a research career, to pick up yoga, to discover painting, and to always push my boundaries more.

I would like to thank Dr. Razvan Voicu not only for his advices and inspirational discussions but also for introducing me to NUS. I would like to thank Dr. Makoto Tatsuta for the exciting opportunity to visit NII. I would also like to thank my co-authors Dr. Wei-Ngan Chin, Dr. Cristina David, Dr. Asankhaya Sharma, Dr. Aquinas Hobor, Dr. Shengchao Qin, Dr. Florin Craciun, Dr. Tibor Kiss, and Shengyi Wang.

My colleagues from the PLSE2 lab made the working environment a stimulating and exciting one. I would most notably like to thank Trung - the Emacs buddy who reinvented my coding experience - but also Adi, Asankhaya, Behnaz, Chanh, Faisal, Rasool, Thai, Toan, Xia Wei, Xuan Bach, and Zhuohong. I would like to thank Ms. Loo Line Fong and Ms. Hee Tse Wei Emily for putting up with all the last minutes admin matters.

I would like to thank Bogdan, Cristina, Cristi and Narcisa for smoothening my relocation to Singapore. Thanks to the Romanian interns of 2016 and 2017 for their warm friendship, energy, appreciation, and all the funny nicknames I ended up having. Special thanks to Andreea and Valentina for their help and interest in HIP/SLEEK in general, and session logic in particular.

I'm so lucky to have had the best stress busters ever: Erica, Mayuko, Teo, Tommi, Trini, Trung, Sana, Yami, Yurika. At different periods of time, they provided a fun and remarkable platform to share cross-field science and to exchange culture. I thank you for that! I have often found emotional balance in the uplifiting talks with Tavi. I am deeply grateful to him for all these years of remote support and affection.

I owe my deepest gratitude to my family for their words of encouragement, down-to-earth advices and for the inexhaustible love, care and patience shown through these years.

TABLE OF CONTENTS

SUMMARY

This thesis tackles the formalization of concurrent programs, where the synchronization among processes is mainly achieved via message passing. Since message passing is a preferred avenue for achieving high performance in distributed applications, ensuring its safe and correct design and implementation is crucial, yet challenging. The challenge mostly comes from the complex interaction schemes normally used in a distributed system. This thesis highlights how a potential problem, known as communication race, may arise. Such a race occurs when two or more processes access a common channel in the wrong sequence, when performing some send/receive operations. Furthermore, to achieve better concurrency, most of today's systems have the benefit of supporting and employing a variety of synchronization primitives besides message passing, increasing thus the complexity of communicating programs. Previous approaches, largely based on session types, provide a concise way to express protocol specifications and a lightweight approach for checking their implementation. However, current solutions mostly rely on implicit synchronization for achieving safe concurrency, and are based on the less precise types rather than logical formulae.

This thesis proposes a session logic which offers an expressive, yet simple specification language to describe the communication between multiple participants in the presence of explicit synchronization. It first shows both formally and by means of example how such a logic can be used to design and specify communication-centered programs. It next proposes the necessary means to incorporate this logic into a verification framework which helps to either (i) uncover communication-related bugs or (ii) prove that the communication design has correctly emerged into a bug-free implementation.

Furthermore, akin to weak memory models where memory load and store reordering is permitted due to hardware and compiler optimization, our belief is that the communication protocols also afford to expose a relaxed behavior due to the implementation choices or concurrency optimization. The proposed support for such relaxed communication protocols is two-fold. On the one hand, the sequentiality of transmissions need not be strictly enforced provided that the

final outcome of the communication is guaranteed to be the desired one. On the other hand, the user is allowed to specify racy protocols (or relaxed protocols) on the grounds that the implementation may use explicit synchronization to achieve race-free communication.

In a nutshell, the proposed formalization identifies each message transmission or reception as a unique event, and ensures race-freedom from *first principle*. To this purpose the thesis defines two kinds of event-ordering constraints, namely *"happens-before"* $\prec_{\texttt{HB}}$ and *"communicates-before"* $\prec_{\texttt{CB}}$, which *separate in time* the different events. A racy scenario is identified when the "happens-before" between adjacent communcications events (over common channels) cannot be proved to hold. The benefits of this proposal are highlighted by integrating it with a complex programming model which also assumes shared resources. To support *separation in space*, the currently proposed session logic is integrated with the separation logic formalism and the benefits of such a fusion are highlighted by means of examples.

LIST OF FIGURES

Chapter 1

INTRODUCTION

"One of the most important properties of a program is whether or not it carries out its intended function."
C.A.R. Hoare - *An Axiomatic Basis for Computer Programming* [37]

Specification and Verification of Message-Passing Programs. Hoare's statement about the importance of the correctness of programs, made almost half a century back, increases in relevance even more in today's digitally dependent world. A major difference between the context when this remark was made and today's programming environment is that the programming languages and the programming paradigms per se are now remarkably more complex. Saying that "finding the proof of correctness for a system with respect to a class of problems is challenging" is undoubtedly an understatement in the current context.

The programming complexity rose from the need to add extra functionality and extra performance to our computing environment. For example, the notable performance and scalability of our computing platform can be attributed to the advancement of distributed computation. In turn, distributed computation has seen a rise in the level of concurrency with the adoption of the message passing model, where concurrent programs communicate with each other via communication channels. Given that the interaction schemes normally used when developing communication-centered program are manifold and highly tangled, ensuring the safety and correctness of such system is strenuous, yet necessary for safety-critical software.

Testing can be used to uncover part of an application's existing bugs. In some cases exhaustive testing is feasible, but generally it is not, let alone for communication-centered concurrent

applications where the number of ways communications can happen simply soars. The reason why it is so difficult to get this kind of applications right is because communication often depends on timing distributions across different processes, or on external factors such as resource availability, the speed of reading/writing from/to a channel, system's signals, and so on.

An inexperienced programmer or even an experienced one could easily design and develop a buggy communication with nondeterministic corner cases, often not captured by simple testing, and often difficult to reproduce. Even the most advanced empirical methods available today are only used to uncover program bugs, they cannot be used to conclude the absence of bugs. Formal methods have the potential to plug this gap, by making safety and correctness guarantees about concurrent, communicating programs irrespective of their running environment, covering thus all possible running scenarios.

Formal methods, or verification of programs in particular, seek to make safety and correctness guarantees by virtue of a set of mathematical theories expressed in formal languages specific to the class of addressed bugs. Software verification assumes a mathematical model of the program as well as mathematical model of its specification, and verifies whether the two models are equivalent. The outcome is a proof of the program's correctness with respect to the given specification, or in other words a guarantee that the program behaves as expected given all possible calling contexts.

As powerful as formal methods are though, it is not possible to solve a general correctness problem and therefore we need to set a more realistic goal, with precise classes of bugs in mind. This thesis tackles the verification of communication-centered programs, where concurrency synchronization is mainly achieved by, but not limited to, message passing. This thesis's main goal is to check that *communicating programs are safe* with respect to the transmitted messages in any calling context. Additional sub-goals include safe resource usage within the message passing programs, expressive mathematical model of the program's specification, and a proper level of automation.

"It [separation logic] represents a major force for taming the looming problem of concurrent software to be run on multi-core processors."
S. Brookes, P. W. O'Hearn, U. Reddy - *The Essence of Reynolds* [6]

Separation Logic. More than a decade of separation logic [49, 86] managed to successfully deliver significant advances towards software reliability. This is mainly because software verification has greatly benefited from the concise and precise program specifications which separation logic enables through local reasoning. Its versatility in reasoning about resources made separation logic a favourite choice when dealing with concurrency [92, 29, 78]. Its extensions include support for barriers [41], locks [36, 40, 78, 36, 5], joins [90], resource ownership permissions [3, 4, 69], event driven programs [59, 60], and channels [39, 94, 1].

As highlighted in [39], the idea of using separation logic to capture the separation of space, perfectly complements that of separation in time, which is a prerequisite for specifying communication patterns. More specifically, the concurrent execution `P*Q` expresses the fact that `P` and `Q` use different, non-overlapping resources - *separation in space* - while `P;Q` indicates that `P` finishes before `Q` starts - *separation in time*.

On top, separation logic also buys us a free lunch when dealing with copyless message passing which assumes resource sharing [68], since the entree of separation logic is that of safe resource manipulation. With all these benefits in mind, but also with the desire of having a rather general logic able to cater for different concurrent primitives, this thesis proposes to extend separation logic to support safe concurrency via message passing in an environment which also accounts for explicit synchronization primitives such as wait/notify, countdownlatch, etc. This thesis's goal is to have a *specification model* which is expressive enough for the explicit synchronization mechanism to be managed by the verification process, rather than by extending the logic to cope with each desired synchronization primitive.

1.1 Motivation

> "[...] interaction with the programmer is vital. The ultimate goal of creating error-free software will not be met by filtering the results of conventional programming through any kind of verification. Instead specification and verification must be tightly interwoven with program construction so that a usable logic must be concise and readable."
>
> John C. Reynolds - *An Overview of Separation Logic* [87]

Communication protocols are a special class of program specifications which describe different properties related to the interaction between communicating programs, properties such as the messages transmission pattern or the communication's technical requirements. Similar to the general-purpose program specifications, protocols could be written informally, such as providing and maintaining blueprints or user manuals, or formally, having a precise meaning in a well-defined mathematical model. We advocate for the latter model, since formal specifications are the first step towards reliable, certified and maintainable software.

The state of the art in writing formal protocols with multiple players (hence on interchangeably termed as parties or peers) revolves around variants of the *session type* theory [46] predominantly for π-calculus. In this line of work, communication protocols are given as types abstracting the communicating processes. If a process is correctly typed against such a typing context, then that process is said to correctly implement the given protocol. Consider the simple example below, which is an excerpt from a classic ATM protocol (where "." represents sequence, and `end` indicates the end of the conversation):

$$\texttt{Client} \to \texttt{ATM} : \texttt{c}\{\mathit{balance} : \texttt{ATM} \to \texttt{Client} : \texttt{c}\langle\texttt{nat}\rangle.\texttt{end},$$
$$\mathit{withdraw} : \texttt{Client} \to \texttt{ATM} : \texttt{c}\langle\texttt{nat}\rangle. \quad \ldots \quad \}.$$

`Client` informs `ATM` over a channel `c` of its choice to either check for the account balance or to withdraw money. The communication then continues according to the corresponding choice. On the $\mathit{withdraw}$ branch the `Client` informs the `ATM` over the amount it wants to withdraw, constrained by the session type to be a natural number. However, constraints related to the exchanged messages can not go beyond types in the session types theory. For example, should the designer also want to impose an upper limit to the allowable amount that the `ATM` can dispense within one withdrawal, then there is no straightforward solution by relying solely on types. One of the main goals of this thesis is to go beyond type safety, into *designing a logic as a formal support for protocols* with complex message properties, including but not limited to numerical

properties of exchanged messages as well as safe resource sharing via message passing.

Consider next a segment of a communication protocol which involves three parties sharing the same channel, where a collector party DB is supposed to receive information from its peers according to the following scenario:

$$\mathtt{I} \rightarrow \mathtt{DB} : \mathtt{c}\langle \mathtt{string} \rangle \,.\, \mathtt{P} \rightarrow \mathtt{DB} : \mathtt{c}\langle \mathtt{nat} \rangle . \mathtt{end}$$

Assuming asynchronous communication with blocking receive and non-blocking send, such a protocol is considered unsafe by most of the approaches which handle formal communication protocols. The safety issue comes from the fact that sending is non-blocking, and therefore P might race ahead of I, leading thus to an unsafe first read by DB. The assumption made by the current state of the art when analyzing this kind of protocols and deem it as unsafe is that synchronization is mainly achieved implicitly. However, most of modern systems rely on a mix of synchronization mechanisms, therefore concluding that such a communication is always unsafe by solely analyzing the protocol is too early, as emphasized by the following implementations of the three-party protocol mentioned earlier:

I	P	DB
...	...	...
send(c, "EVO 2.5 SATA III SSD");	send(c, 99);	item = recv(c);
		price = recv(c);

(a)

I	P	DB
...	...	...
send(c, "EVO 2.5 SATA III SSD");		item = recv(c);
notifyAll(w);	**wait(w)**;	price = recv(c);
	send(c, 99);	

(b)

While implementation (a) might lead to a race and should therefore be declared as an unsafe code, the same cannot be said about (b) which also uses other synchronization mechanisms besides message passing. And this is a reasonable expectations from today's software, even more so in reactive applications or mobile apps where message passing is preferred due to performance interests, while explicit synchronization is often mingled with message transmission for event notification. This thesis aims to fill this gap by proposing a logic with support for protocol refinement, such that each potentially unsafe protocol is refined into a safe protocol by adding

event ordering guards. The implementation is then verified against the refined protocol to ensure *communication race-freedom.*

Furthermore, while the current state of the art supports the specification of parallel communication, for example $\mathtt{G}, \mathtt{G}'$ as a session type indicates that the communications described by $\mathtt{G}$ and by $\mathtt{G}'$ is consumed concurrently, it strictly disallows for these specifications to mention the same party, that is $\mathtt{G}$ and $\mathtt{G}'$ share no common party. While this might be a valid assumption should we desire a race-free communication, it is too restrictive in certain contexts, i.e. it prohibits the communication designer from specifying nondeterministic communication. Nondeterminism often plays a key-role in achieving performance in distributed computation, where, for example, a common receiver should be insensitive to the order of the collected computation results. This thesis tackles the problem of *nondeterminisms* in communication by formally defining the boundaries between manageable/desirable nondeterminism and unsafe communication.

Having attached a syntax and a semantics to a session logic which tackles all the above mentioned key-points, the next step is to use this logic in the verification of protocols' implementations. The essential properties to be verified refer to:

- protocol conformance: the implementation follows the communication pattern described by the protocol.
- communication safety: race-freedom, deadlock-freedom, type-safety, safe resource access (assuming full ownership transfer), leak-freedom (a channel cannot be closed unless there is no other communication expected over this channel), no unexpected transmission (every send or receive operation in the implementation must be accounted for in the protocol as well).

The ultimate goal is to perform the verification of communication-centered programs locally, despite the distributed nature of the communication, and automatically, i.e. with minimum user effort: the user is only supposed to provide the communication protocols and program specifications.

1.2 Contributions

After a brief introduction over the research area and goals of the current thesis, the summary below lists the contributions which are to be detailed in the subsequent chapters:

A **dyadic session logic** for describing and reasoning about communication centered applications, which has the important benefit against the traditional type system based approaches [62, 46, 57] of using expressive logical formulae to describe the interactions between communicating parties. These formulae go beyond types when describing the exchanged messages, and capture numerical and resource ownership properties. The contributions of this dyadic logic as opposed to other logic-based approaches [93, 94] for two-party communications, are summed up to:

- a novel usage of *disjunctions* to model both internal and external choices, with the immediate effect of loosening the constraints over the available programming constructs (instead of dedicated switch constructs, the program may instead communicate its choices with the commonly available conditional statements, such as if-then-else), and having a more concise specification language which uniformly captures both internal and external choice.
- a novel usage of *higher-order channels* to specify and implement delegation, which treats all the transmissions uniformly irrespective of the communicated message (data, channel reference, memory reference).
- two mechanisms for *deadlock detection* able to pin down both intra- as well as inter-channel deadlocks. While the former is a novel detection mechanisms which uses specification compatibility check to detect deadlocks, the later is a variation of the waits-for graph normally used in the literature for circularity detection, tailored in this work to suit the communication-centered programs.

A **multiparty session logic** which extends the dyadic session logic to account for the conversation between multiple parties possibly sharing the same channels. The novel characteristics to the currently proposed logic are as follows:

- an original *protocol refinement* procedure which decorates a global protocol with event orderings. This refinement allows us to soundly relax the notion of race-free

communications normally used in the session type approaches, where race-freedom is decided based on a protocol analysis (the analysis here refers to the causality analysis [46] of transmissions). In contrast, the refinement process proposed in this thesis converts every protocol into a race-free protocol by introducing event ordering guards. In this context, race-free communication refers not to a race-free protocol, but to the actual implementation of the communication which is checked for conformance against the refined protocols. An important consequence of our approach is that the we can uniformly account for both implicit and explicit synchronization, whereas deciding race-freedom based solely on a causality analysis of the considered protocol disregards the possibility of programs using explicit synchronization.

- a *constraint solving system* for the aforementioned event orderings with a novel usage of constraint propagation lemmas.
- a *projection mechanisms* which derives the local view for each communicating party. Even more granular, the projection also derives the local view for each logical endpoint (the communication view of a party w.r.t. a channel), an important component for local reasoning and for transparent delegation. In contrast, the projection granularity for session types stop at per party view, therefore needing additional equipment to support delegation in a multiparty communication context.
- we propose an enhancement to an existing verification framework [20] to *automatically verify* a program's conformance to a given communication protocol.
- a novel *cooperative proving* mechanisms, where the proof by one party can be relied as assumption by the other concurrent parties.

A machinery to express **modular protocols**, where "plug-and-play" protocols are only designed and refined once, and re-used multiple times in different contexts. This machinery, which is actually a set of logical enhancements to the main multiparty theory, forms the basis for supporting both inductive protocol specification and their corresponding recursive implementation verification.

A novel definition for the well-formedness of **nondeterministic protocols.**

A **storage model and operational semantic** for a language with support for communication primitives, dynamically allocated memory and selected concurrency primitives, such as

thread creation and join, explicit synchronization, etc.

1.3 Thesis Overview

This thesis starts by recording the theories and concepts which represent the foundation of the current proposals, in Chapter 2. Subsequent chapters succeed as follows:

- Chapter 3 describes the dyadic session logic. Even though this chapter is meant to emphasize on the benefits of using logical formulae in designing communication protocols, it also displays a scent of formalism for some of the logic's key points.
- Chapter 4 discusses some of the concurrency related problems in the context of communication protocols, and introduces the multiparty session logic.
- Chapter 5 describes the proposed proof system and the automated verification process.
- Chapter 6 introduces the operational semantics of the target language as a basis for soundness proof.
- Chapter 7 describes the machinery for obtaining modular and inductive protocols.
- Chapter 8 presents the related work distinguishing between two main avenues of research: (i) verification of concurrent communicating programs. and that of (ii) behavioral type analysis for message-passing programs.
- Chapter 9 concludes this thesis and offers directions for future works on this topic.

Chapter 2

PRELIMINARIES

This chapter summarizes some of the relevant technicalities behind the current thesis. In particular, it introduces the programming language, its storage model and its semantics. It next describes the specification language and its semantic, followed by the verification rules and the proof obligations solver.

2.1 Programming Language

The target language employed throughout the thesis is depicted in Fig. 2.1. This core imperative language resembles that of C, with support for user defined data structures *datat*, method definitions *meth* annotated with pre- and post-specifications *mspec* and some concurrency primitives. Spawning of thread is performed by using the || operator, which semantically can be seen as a fork-join concurrency model. The specification language used for describing *mspec* is detailed in Sec. 2.3. For simplicity, the programs are assumed to be well typed.

The language in Fig. 2.1 is expression-oriented, with the body of a method `meth` being a compound expression `e`.Moreover, the language allows both call-by-value and call-by-reference method parameters. As for iterative loops, they are considered as a syntactic sugar for tail-recursive methods, where mutations on parameters are made visible to the caller via pass-by-reference parameters. This technique of translating away iterative loops is standard and is helpful in further minimizing our core language.

Most of the language constructs are standard, with `open` and `close` used for the creation and

Program	$\mathcal{P}$	::=	*datat** *meth**
Data Struct.	*datat*	::=	`struct` d { *(t f)** }
Method Definitions	*meth*	::=	t *mn* $((t\ v)^*)$ *mspec* $\{e\}$
Types	t	::=	$d \mid \tau$
Primitive Types	τ	::=	`int` | `bool` | `float` | `void`
Expressions	e	::=	`NULL` | k^τ | v | `new` $d(v^*)$ | $t\ v;\ e$ | $v.f$ | *mn*(v^*) | `skip`
			| $v{:=}e$ | $v.f{:=}e$ | $e;e$ | $e \| e$ | `if` $(b)\ c$ `else` e | `return` e
			| `open()` `with` *sspec* | `close`(v) | `receive`(v) | `send`(v, e)
Boolean Expressions	b	::=	$e{==}e \mid !(b) \mid b\&b \mid b\|b$
			where $\mathtt{k}^\tau$ is a constant of type τ,
			v is a variable, f denotes a field

Figure 2.1: A Core Imperative Language

destruction of channels, while `send` and `receive` are used for the transmission of messages. A channel is always created with a communication specification *sspec* attached whose syntax differs based on whether it is a dyadic or a multiparty session. The syntax of *sspec* is detailed in Fig. 3.2 and Fig. 4.5, respectively.

As a warm-up example, the following program which simply exchanges a message between two peers, depicts some of the language's main constructs:

```
1  struct node { int val; struct node *next; };
2  struct Channel { int id; int bufsize; struct node *buf; };

4  void foo()  {
5    Channel *c = new Channel(fresh_id(),200,NULL);
6    c = open();
5    // producer                 ||   // consumer
6    struct node *x;             ||   struct node *element;
7    x = new node(10,NULL);      ||   element = receive(c);
8    send(c, x);                 ||   consume_element(element);
10   close(c);
11 }
```

Note that the above is a simple example with no specification, it serves as a code illustration in order get familiar with the programming style. The `node` data structure is used for creating a linked-list style structure, while `Channel` stores the information related to a communication channel.

2.2 Concurrent Separation Logic

Due to its expressive power and elegant proofs, concurrent separation logic (CSL) was selected as the primary theory behind the current work. Separation logic is an attractive extension of Hoare logic in which assertions are interpreted w.r.t. relevant portions of the heap. Spatial conjunction $\kappa_1 * \kappa_2$, the core operator of separation logic, divides the heap between two disjoint heaps described by assertions κ_1 and κ_2, respectively. The main benefit of this approach is the support for local reasoning: the specifications of a program code need only mention the portion of the resources which it uses, the rest are assumed unchanged.

This section lists down the main assertions which are normally used to reason about concurrent heap-manipulating programs. The next chapters will introduce the extensions needed to reason about communicating programs.

Storage Model. Following the traditional storage model for heap manipulating programs, the program state is defined as the pair:

$$\mathtt{State} \triangleq \mathtt{Stack} \times \mathtt{Heap}$$

where a stack $\mathtt{s} \in \mathtt{Stack}$ is a total mapping from local and logical variables `Var` to primitive values `Val` or memory locations `Loc`; a heap $\mathtt{h} \in \mathtt{Heap}$ is a finite partial mapping from memory locations to data structures stored in the heap, `DVal`:

$$\mathtt{Stack} \triangleq \mathtt{Var} \to \mathtt{Val} \cup \mathtt{Loc} \qquad\qquad \mathtt{Heap} \triangleq \mathtt{Loc} \rightharpoonup_{\mathtt{fin}} \mathtt{DVal}$$

Assertions. The core assertions which are specific to separation logic are given below, but a

more expressive language is defined in the next section:

Spatial formula	$\kappa ::=$		
		$\mid \texttt{emp}$	empty heap
		$\mid \texttt{v}\mapsto\texttt{d}\langle\texttt{v}^*\rangle$	points-to
		$\mid \kappa * \kappa$	separating conjunction
Formula	$\Delta ::=$		
		$\mid \kappa \wedge \pi$	composition of spatial and arithmetic
		$\mid \exists\, \texttt{v}^* \cdot \Delta$	existantial quantification
		$\mid \Delta \vee \Delta$	disjunction

where v is a first-order variable, d is a user-defined data structure, and

π is a non-spatial formula to be defined in the subsequent section.

Semantic Model. The semantic model for the separation assertions is characterized by a *satisfaction relation* between a program state $(s,h) \in \texttt{State}$ and a separation logic formula Δ, defined as below:

$$
\begin{array}{ll}
(s,h) \vDash \texttt{emp} & \text{iff } \texttt{true} \\
(s,h) \vDash \texttt{v}\mapsto\texttt{d}\langle\texttt{v}_1,..,\texttt{v}_\texttt{n}\rangle & \text{iff } \texttt{struct d}\{\texttt{t}_1\ \texttt{f}_1;..;\texttt{t}_\texttt{n}\ \texttt{f}_\texttt{n}\} \in \mathcal{P} \text{ and} \\
 & \quad h = [s(\texttt{v}) \mapsto \texttt{d}[\texttt{f}_1 \mapsto s(\texttt{v}_1),..,\texttt{f}_\texttt{n} \mapsto s(\texttt{v}_\texttt{n})]] \\
(s,h) \vDash \kappa_1 * \kappa_2 & \text{iff } \exists\, h_1, h_1 \cdot (h_1 \perp h_2 \text{ and } h_1 \uplus h_2 = h) \text{ and} \\
 & \quad (s,h_1) \vDash \kappa_1 \text{ and } (s,h_2) \vDash \kappa_2 \\
(s,h) \vDash \kappa \wedge \pi & \text{iff } (s,h) \vDash \kappa \text{ and } s \Rightarrow \pi \\
(s,h) \vDash \Delta_1 \vee \Delta_2 & \text{iff } (s,h) \vDash \Delta_1 \text{ or } (s,h) \vDash \Delta_2 \\
(s,h) \vDash \exists\, v \cdot \Delta & \text{iff } \exists\, \texttt{k} \in \texttt{Val} \cdot (s[\texttt{v} \mapsto \texttt{k}], h) \vDash \Delta
\end{array}
$$

where the binary operators $\perp$ and $\uplus$ are defined over the domain of the heap maps, and represent heap disjointness and union, respectively:

$$
\begin{array}{rcl}
h_1 \perp h_2 & \Leftrightarrow & \texttt{dom}(h_1) \cap \texttt{dom}(h_2) = \emptyset \\
h = h_1 \uplus h_2 & \Leftrightarrow & \texttt{dom}(h_1) \cup \texttt{dom}(h_2) = \texttt{dom}(h)
\end{array}
$$

To note the monotonicity of the first satisfaction relation, namely $(s,h) \vDash \texttt{emp}$ iff $\texttt{true}$ which captures the intuitionistic character of our approach: an assertion which evaluates to true for some portion of the heap, remains true for any extension of that heap. However, adopting

intuitionistic reasoning uniformly through the verification process, makes the verifier unable to detect memory leaks. Therefore, to allow leakage detection, the checking of postcondition engages classic reasoning, namely $(s, h) \vDash \texttt{emp}$ iff $\texttt{dom}(h) = \emptyset$.

Main Axioms. One of the main benefits of using separation logic is the conciseness of specifications due to the implicit aliasing information subsumed by the separation conjunction. Yet another strong legacy of separation logic is the pair of main axioms which enable local reasoning, namely the *frame rule* and *disjoint concurrency*:

$$\frac{\{\Delta_1\}\ \ e\ \ \{\Delta_2\}}{\{\Delta_1 * \Delta\}\ \ e\ \ \{\Delta_2 * \Delta\}}\ \ \texttt{fv}(\Delta) \cap \texttt{modif(e)} = \emptyset$$

where the side condition states that no variable occurring free in Δ is being modified by program $\texttt{e}$. Or in other words, that the memory footprint of $\texttt{e}$ and the heap described by Δ are disjoint. The most important consequence of this axiom is that of *local reasoning* as stated by O'Hearn, Reynolds and Yang [75]: "To understand how a program works, it should be possible for reasoning and specification to be confined to the cells that the program actually accesses. The value of any other cell will automatically remain unchanged." This simply describes the monotonicity of the program with respect to its memory safety. If the execution of a program with a certain footprint is safe, then that program is safe even when the heap is extended with arbitrary memory locations since the extra heap is untouched by $\texttt{e}$.

Moving forward, the benefits of CSL culminate with the axiom for disjoint concurrency:

$$\frac{\{\Delta_1\}\ \ e\ \ \{\Delta_2\} \qquad \{\Delta_1'\}\ \ e'\ \ \{\Delta_2'\}}{\{\Delta_1 * \Delta_1'\}\ \ e \,||\, e'\ \ \{\Delta_2 * \Delta_2'\}}\ \ \begin{array}{l}(\texttt{fv}(\Delta_1) \cup \texttt{fv}(\Delta_2)) \cap \texttt{modif(e}') = \emptyset \\ (\texttt{fv}(\Delta_1') \cup \texttt{fv}(\Delta_2')) \cap \texttt{modif(e)} = \emptyset\end{array}$$

where no free variable of Δ_1' or Δ_2' is modified by $\texttt{e}$, and conversely. The result is preserved even if the parallel composition is extended to an arbitrary number of concurrent processes.

The communication reasoning in the subsequent chapters is particularly dependent on the idea of *precise* formulae as defined in [5], which articulates the fact that in every state there is at most one sub-heap in which the formula holds:

Definition 1. *A formula Δ is precise if for all states (s, h) there is at most one sub-heap $h_0 \subseteq h$ such that $(s, h_0) \vDash \Delta$.*

For example, $\texttt{emp}$ and the points-to $\texttt{v}\mapsto\texttt{d}\langle \texttt{v}_1, .., \texttt{v}_\texttt{n}\rangle$ are precise, and so is $\Delta_1 * \Delta_2$ if Δ_1 and

Δ_2 are precise. The precision property also holds for exclusive disjunction $(\Delta_1 \wedge \mathtt{b}) \vee (\Delta_2 \wedge \neg(\mathtt{b}))$, where $\mathtt{b}$ is a Boolean formula and Δ_1 and Δ_2 are precise.

2.3 Specification Language

As stated in Sec. 2.1, a program $\mathcal{P}$ comprises data declarations *datat* and method definitions *meth*. Each method definition is decorated with some program specifications *mspec* in the style of Chin et. al. [20] as follows :

$$\begin{array}{l} \mathtt{requires}\ \ \Phi^{\mathtt{1}}_{pr}\ \ \mathtt{ensures}\ \Phi^{\mathtt{1}}_{po}; \\ \qquad\qquad \vdots \\ \mathtt{requires}\ \ \Phi^{\mathtt{n}}_{pr}\ \ \mathtt{ensures}\ \Phi^{\mathtt{n}}_{po}; \end{array}$$

$\{(\Phi^{\mathtt{i}}_{pr}, \Phi^{\mathtt{i}}_{po})\}^{\mathtt{n}}_{\mathtt{i}=1}$ is a set of pre- and postconditions tailored to suit different calling contexts. If the state of a caller satisfies one of the preconditions, say $\Phi^{\mathtt{i}}_{pr}$, and if the program terminates, then, after the method call, the state of the caller reflects the corresponding postcondition, $\Phi^{\mathtt{i}}_{po}$. Each caller may satisfy multiple postconditions.

A specification is formally described by a DNF formula Φ (Fig. 2.2), where each disjunct comprises both spatial formulae κ as well as first-order logic formulae π (or pure formulae), with pointer equality and inequality, γ, Boolean formulae, b, and Presburger Arithmetic, $\mathtt{s}$. This specification language will be enhanced with support for session specification as later described in Chapters 3 and 4. The spatial formulae are written using the separation logic [85, 50] constructs: `emp` to denote empty heap, the singleton heap $v{\mapsto}d(v^*)$ to denote that the pointer `v` points to a memory location which stores a data structure of type `d`, and the spatial conjunction $\Delta_1 * \Delta_2$ to express that Δ_1 and Δ_2 refer to non-overlapping memory locations.

User-defined data structures. The specification is rich enough to support user-defined data structures. This means that our approach provides the user with flexible and expressive means to specify properties about these data structures, ranging from numerical constraints, to memory shape and content, to resources accessibility footprint and communication patterns.

As an example, let us revisit the `node` data structure defined in the Sec. 2.1, used for building an acyclic linked-list:

```
struct node { int val; struct node *next; };
```

To describe a linked-list of size `n`, our specification language offers support for writing inductively-defined predicates which capture the desired properties:

$$
\begin{array}{lll}
\textit{Symbolic pred.} & \textit{pred} & ::= p(\mathtt{root}, v^*) \equiv \Phi \; \textit{inv} \; \pi \\
\textit{Formula} & \Phi & ::= \bigvee \Delta \qquad\qquad \Delta ::= \exists\, v^* \cdot \kappa \wedge \pi \mid \Delta * \Delta \\
\textit{Heap formula} & \kappa & ::= \mathtt{emp} \mid \mathtt{v} \mapsto \mathtt{d}\langle \mathtt{v}^* \rangle \mid p(v^*) \mid \kappa * \kappa \mid V \\
\textit{Pure} & \pi & ::= v : t \mid b \mid a \mid \pi \wedge \pi \mid \pi \vee \pi \mid \neg \pi \mid \exists v \cdot \pi \mid \forall v \cdot \pi \mid \gamma \\
\textit{Pointer eq./diseq.} & \gamma & ::= v = v \mid v = \mathtt{null} \mid v \neq v \mid v \neq \mathtt{null} \\
\textit{Boolean} & b & ::= \mathtt{true} \mid \mathtt{false} \mid b = b \qquad a ::= s = s \mid s \leq s \mid V = \Delta \\
\textit{Presburger Arith.} & s & ::= k^{\mathtt{int}} \mid v \mid k^{\mathtt{int}} \times s \mid s + s \mid -s
\end{array}
$$

where $k^{\mathtt{int}}$: integer constant; v : first order variable;
V : second-order variable;
d : name of a user-defined data structure

Figure 2.2: The Specification Language

$$\mathtt{pred\ ll(root,n)} \triangleq \mathtt{root{=}null} \wedge \mathtt{n{=}0} \vee \exists\, \mathtt{t} \cdot \mathtt{root} \mapsto \mathtt{node}\langle \mathtt{10}, \mathtt{t} \rangle * \mathtt{ll(t,n{-}1)}$$

where the base case expresses an empty list, while the inductive case states that the list's size is at least one since `root` points to a singleton heap which records two fields, namely `val` and `next`, with $[\mathtt{val} : \mathtt{10};\ \mathtt{next} : \mathtt{t}]$.

Higher-order variables. To support communication primitives with generic specifications we allow the usage of higher order variables `V`. For example, the following spatial formula $\mathtt{x} \mapsto \mathtt{node}\langle \mathtt{10}, \mathtt{t} \rangle * \mathtt{V} \wedge (\mathtt{V{=}ll(t,n)})$ contains the higher-order variable `V` which is instantiated to $\mathtt{ll(t,n)}$. Eliminating `V` from the initial formula leads to the normalized logical form $\mathtt{x} \mapsto \mathtt{node}\langle \mathtt{10}, \mathtt{t} \rangle * \mathtt{ll(t,n)}$.

2.4 Proof Rules

The front-end of our proposal is a Hoare-style forward verifier, in the fashion of HIP/SLEEK [20]. In this section we list down the standard proof rules applied by this verifier, and postpone the description of session related Hoare-triples to subsequent chapters.

Since we built the assertions on top of separation logic, we also benefit from its accompanying Hoare-style proof system. In this system the logic statements take the shape of Hoare triples $\{\Delta_1\}\ \mathtt{e}\ \{\Delta_2\}$, where Δ_1 and Δ_2 are separation logic formulae representing the pre- and post-state, respectively, of an expression `e`. These triples are then formally proved against some well-formed rules using derivation trees.

Moreover, the proposed proof systems assumes that each method is decorated with a set of pre- and post-condition. The actual verification process then follows the traditional forward

verification rules: if the pre-condition of a method call is satisfied at the caller's context, the corresponding post-condition is added to the caller's poststate; assuming its precondition as the initial abstract state, the verification of a method definition inspects whether the postcondition holds after having checked each of the method's body instructions. During the verification process, the verifier makes regular calls to the back-end entailment prover, described in Sec. 2.5, in order to solve the collected proof obligations.

Our Hoare judgment is shaped by triples of the following form $\vdash \{\Delta_1\}\, e\, \{\Delta_2\}$, where Δ_2 is computed given Δ_1. We next generalize this triple to support a set of post states instead of just one post state: $\vdash \{\Delta\}\, e\, \{S\}$, where S is a residual set of heap states discovered by the proof-search based strategy adopted during the verification process. The verification is said to have succeeded with Δ as prestate if the residual set is non-empty, and failed otherwise.

Formally, the proof rule for method definition and method call, are written as below:

$$\frac{\begin{array}{c}[\textbf{METH-DEF}]\\ V{=}\{v' \mid v \in V\}.\\ \forall i = 1,..,p \cdot (\ \vdash \{\Phi^i_{pr}\}\, e\, \{S^i_1\} \qquad \exists V{\cdot}S^i_1 \vdash \Phi^i_{po} * S^i_2 \qquad S^i_2{\neq}\{\}) \qquad S = \bigcup_{i=1}^{p} S^i_2\end{array}}{\vdash \{\texttt{emp}\}\, t\ mn((t_j\ v_j)_{j=1}^{n})\ \{\texttt{requires}\ \Phi^i_{pr}\ \texttt{ensures}\ \Phi^i_{po}\}_{i=1}^{p}\ \{e\}\, \{S\}}$$

$$\frac{\begin{array}{c}[\textbf{METH-CALL}]\\ t_0\ mn((t_j\ v'_j)_{j=1}^{n})\ \{\texttt{requires}\ \Phi^i_{pr}\ \texttt{ensures}\ \Phi^i_{po}\}_{i=1}^{p}\ \{e\} \in \mathcal{P}\\ \rho{=}[v_j/v'_j]_{j=1}^{n} \qquad \forall i{=}1,..,p \cdot (\Delta{\vdash}\rho\Phi^i_{pr} * S_i) \qquad S = \bigcup_{i=1}^{p}(S_i * \Phi^i_{po}) \qquad S \neq \{\}\end{array}}{\vdash \{\Delta\}\, mn(v_1..v_n)\, \{S\}}$$

The proof obligations $\exists V{\cdot}S^{\mathtt{i}}_1 \vdash \Phi^{\mathtt{i}}_{po} * S^{\mathtt{i}}_2$ and $\Delta{\vdash}\rho\Phi^{\mathtt{i}}_{pr} * S_{\mathtt{i}}$ in the above verification rules are solved using the entailment checking rules described in Sec. 2.5. To note that the residual states $S^{\mathtt{i}}_2$ and $S_{\mathtt{i}}$ are derived during the entailment check. ρ denotes substitution, and $\mathcal{P}$ is the program targeted for verification.

The rest of the verification rules are depicted in Fig. 2.3, where the special variable `res` denotes the result of evaluating the target expression. For the brevity of the formalization, we lifted the binary operations normally used for formulae composition, namely $*$, $\wedge$, $\vee$, to

composition of sets and formulae in order to precisely capture the residue of each proof rule (this lifting also applies to the `fv` function):

$$
\begin{array}{lcl}
\exists\, \mathtt{v} \cdot (S \wedge \pi) & \equiv & \{\exists \mathtt{v} \cdot (\Delta_{\mathtt{s}} \wedge \pi) \mid \Delta_{\mathtt{s}} \in S\} \\
\pi \wedge S & \equiv & \{\Delta_{\mathtt{s}} \wedge \pi \mid \Delta_{\mathtt{s}} \in S\} \\
S \wedge \pi & \equiv & \{\Delta_{\mathtt{s}} \wedge \pi \mid \Delta_{\mathtt{s}} \in S\} \\
\Delta * S & \equiv & \{\Delta_{\mathtt{s}} * \Delta \mid \Delta_{\mathtt{s}} \in S\} \\
S * \Delta & \equiv & \{\Delta_{\mathtt{s}} * \Delta \mid \Delta_{\mathtt{s}} \in S\} \\
S_1 \vee S_2 & \equiv & \{\Delta_1 \vee \Delta_2 \mid \Delta_1 \in S_1,\, \Delta_2 \in S_2\} \\
\mathtt{fv}(S) & \equiv & \bigcup_{\Delta \in S} \mathtt{fv}(\Delta)
\end{array}
$$

2.5 Entailment Checking

The back-end of our approach aims to check whether the entailment relation of separation formulae is valid. As indicated in the previous section, the prover uses a search based approach to return the set of residual heap states which make the entailment valid. This is formally expressed with the entailment relation $\Delta_{\mathtt{A}} \vdash \Delta_{\mathtt{C}} * S$, where S is derived and it represents the set of all possible residual heap states. For the entailment to be valid S must be a non-empty set. If S cannot be derived, either because a contradiction has been detected or because the consequent, $\Delta_{\mathtt{C}}$, is too strong for the antecedent $\Delta_{\mathtt{A}}$, we say that the entailment is not valid, or in other words that the entailment relation between $\Delta_{\mathtt{A}}$ and $\Delta_{\mathtt{C}}$ does not hold.

A complete set of entailment checking rules modulo the session related rules can be found in [20]. In this thesis we extend this set of traditional entailment rules with rules tailored to suit the proposed session logic. However we postpone the details of these rules to subsequent chapters.

2.6 Communication Model

To support a wide range of communication interfaces, the current session logic is designed for a permissive communication model, where:

- the transfer of a message dissolves *asynchronously*, that is to say that sending is non-blocking while receiving is blocking.

$$\frac{S = \{\Delta \wedge \texttt{res}{=}k^{\tau}\}}{\vdash \{\Delta\}\, k^{\tau}\, \{S\}}\ [\textbf{CONST}] \qquad \frac{S{=}\{\Delta \wedge \texttt{res}{=}v\}}{\vdash \{\Delta\}\, v\, \{S\}}\ [\textbf{VAR}] \qquad \frac{\vdash \{\Delta\}\, e\, \{S\}}{\vdash \{\Delta\}\, \{t\ v;\ e\}\, \{\exists\, v, v' \cdot S\}}\ [\textbf{LOCAL}]$$

$$\frac{\vdash \{\Delta\}\, e\, \{S_1\} \qquad S_2{=}\exists \texttt{res} \cdot (S_1 \wedge v'{=}\texttt{res})}{\vdash \{\Delta\}\, v{:=}e\, \{S_2\}}\ [\textbf{ASSIGN}] \qquad \frac{S{=}\{\Delta * \texttt{res}{\mapsto}\texttt{d}\langle \texttt{v}_1, .., \texttt{v}_n\rangle\}}{\vdash \{\Delta\}\, \texttt{new}\ d(v_1, .., v_n)\, \{S\}}\ [\textbf{NEW}]$$

$$\frac{\vdash \{\Delta \wedge b\}\, e_1\, \{S_1\} \qquad \vdash \{\Delta \wedge \neg b\}\, e_2\, \{S_2\}}{\vdash \{\Delta\}\, \texttt{if}\ (b)\ e_1\ \texttt{else}\ e_2\, \{S_1 \vee S_2\}}\ [\textbf{IF}]$$

$$\frac{\vdash \{\Delta\}\, e\, \{S_1\} \qquad S{=}\exists v \cdot ((\exists \texttt{res} \cdot (S_1 \wedge v{=}\texttt{res})) \wedge \texttt{res} = v)}{\vdash \{\Delta\}\, \texttt{return}\ e\, \{S\}}\ [\textbf{RETURN}]$$

$$\frac{\Delta \vdash \texttt{v}{\mapsto}\texttt{d}\langle \texttt{v}_1, .., \texttt{v}_n\rangle * S_1 \qquad S_1{\neq}\{\} \qquad \textit{fresh}\ v_1..v_n \qquad S_2 = \exists v_1..v_n \cdot (S_1 * \texttt{v}{\mapsto}\texttt{d}\langle \texttt{v}_1, .., \texttt{v}_n\rangle \wedge \texttt{res}{=}v_i)}{\vdash \{\Delta\}\, v.f_i\, \{S_2\}}\ [\textbf{FIELD-READ}]$$

$$\frac{\Delta \vdash \texttt{v}{\mapsto}\texttt{d}\langle \texttt{v}_1, .., \texttt{v}_n\rangle * S_1 \qquad S_1{\neq}\{\} \qquad \textit{fresh}\ v_1..v_n \qquad S_2 = \exists v_1..v_n \cdot (S_1 * \rho(\texttt{v}'{\mapsto}\texttt{d}\langle \texttt{v}_1, .., \texttt{v}_n\rangle)) \qquad \rho{=}[v_0/v_i]}{\vdash \{\Delta\}\, v.f_i{:=}v_0\, \{S_2\}}\ [\textbf{FIELD-UPDATE}]$$

$$\frac{\vdash \{\Delta\}\, e_1\, \{S_1\} \qquad \vdash \{S_1\}\, e_2\, \{S_2\}}{\vdash \{\Delta\}\, e_1; e_2\, \{S_2\}}\ [\textbf{SEQ}]$$

$$\frac{\vdash \{\Delta_1\}\, e_1\, \{S_1\} \qquad \vdash \{\Delta_2\}\, e_2\, \{S_2\}}{\vdash \{\Delta_1 * \Delta_2\}\, e_1 \| e_2\, \{S_1 * S_2\}}\ [\textbf{PAR}]\ \begin{array}{l}(\texttt{fv}(\Delta_1) \cup \texttt{fv}(S_1)) \cap \texttt{modif}(\texttt{e}_2) = \emptyset \\ (\texttt{fv}(\Delta_2) \cup \texttt{fv}(S_2)) \cap \texttt{modif}(\texttt{e}_1) = \emptyset\end{array}$$

Figure 2.3: Forward Verification Rules

Machine Config.	MC	::=	PState × CH
Channels	CH	::=	$\mathcal{P}$chan⇀(Val ∪ Loc)*
Program State	PState	::=	idt⇀TState
Thread State	TState	::=	State × e
Local State	State	::=	Stack × Heap
Thread Config.	TConfig	::=	TState × CH

PS∈PState TC∈TConfig σ∈State $\overline{\sigma}_{\mathrm{L}}$∈TState chan∈CH

Figure 2.4: A Semantic Model of the Core Language

- the communication interface of choice manipulates *FIFO channels*.
- for simplicity, the communication assumes unbounded buffers.

Chapter 3 applies the above communication model to one in which each channel is shared between exactly two peers, while Chapter 4 takes the more general, yet more realistic approach where channels are shared across multiple communicating parties.

2.7 Operational Semantics

Semantic Model. Fig. 2.4 models the machine for the proposed programming language whose configuration comprises a pair of a *program state* and of *channels states*. The set of channels used by the program are described as a map from a program channel identifier to a *FIFO* list of messages. Moreover, a program is a set of threads, each identified by a unique id and described by their local state, where the local state is modeled as detailed in Sec. 2.2. A thread executing expression e influences its local state σ and the state of the channels CH.

Small-Step Operational Semantics. The operational semantics is given as a set of reduction rules between machine or thread configurations. Each reduction step is indicated by $\hookrightarrow$, where $\hookrightarrow^*$ denotes respectively the reflexive and transitive closure of $\hookrightarrow$. Moreover, $\hookrightarrow^*$ between thread configurations in the machine step, indicates that the proposed semantic is not constrained to a specific scheduler, thus quantifying over all permissible executions.

Similar to the semantics of [5], a thread reduction could either lead to another thread state and interfere in the states of program channels, $\langle\langle\sigma, \mathtt{e}\rangle, \mathtt{CH}\rangle \hookrightarrow \langle\langle\sigma', \mathtt{e}'\rangle, \mathtt{CH}'\rangle$, or it could signal an *error*, $\langle\langle\sigma, \mathtt{e}\rangle, \mathtt{CH}\rangle \hookrightarrow \mathtt{error}$. However, a program which is proved correct should never reach an error state. A program terminates in an error-free state if the final configuration reaches the skip expression.

For convenience, only the semantic reduction rules which do not fault are listed down in the subsequent. If any of the premises in these rules do not hold, then the considered reduction leads to `error`. Moreover, any state which is not captured by the given reduction rules is forced to fault.

We next describe the rules of interest for this work and skip the reduction rules of the stack and heap commands, since they are standard [5], and do not interfere with the channels states.

Fig. 2.5 describes the machine reduction rules:

[OP–MACHINE] quantifies over all kinds of schedulers by allowing thread t to execute an arbitrary number of steps, $\hookrightarrow^*$.

[OP–PAR] spawning two new threads splits the heap into disjoint sub-heaps corresponding to each thread, $\sigma{=}\sigma_1 \uplus \sigma_2$. The parallel composition is modelled in the style of the fork-join framework: the spawning of two new threads, t_1 and t_2, is followed by the `join` operation of the freshly created threads, `join` t_1 t_2.

[OP–JOIN] if both threads have finished their execution, their corresponding resources are transferred to the parent thread, whose local state is updated to capture the disjoint union of the two threads states. Otherwise, the state of the parent thread remains unchanged, as per [OP–JOIN–SKIP1] and [OP–JOIN–SKIP2], until both threads finish their execution. Finally, if any of the two threads faults, as per [OP–JOIN–ERR1] and [OP–JOIN–ERR2], their error is propagated to the parent thread, thus throughout the whole program.

The communication related expressions are given a semantic via the rules in Fig. 2.6:

[OP–OPEN] opens a channel with an empty buffer [ ]. `res` is a special identifier to denote the result of an expression. *sspec* is ignored as of now, since this is part of the communication specification which is only given a meaning during program certification.

[OP–CLOSE] a channel is successfully closed only when its corresponding buffer is empty, otherwise it might leak the undelivered data as highlighted by [OP–CLOSE–LEAK]. However, there might be other leaking scenarios where the buffer is empty during the channel closing, but the communication expected to be consumed on this channel is not completed. Such a scenario cannot be captured by this semantic. Subsequent chapters show how can this scenario be detected both during the communication verification as well as in the operational semantics enhanced with communication knowledge.

$$\frac{[\textbf{OP-MACHINE}]\quad \langle \overline{\sigma}_{\mathtt{L}}, \mathtt{CH}\rangle \hookrightarrow^{*} \langle \overline{\sigma}'_{\mathtt{L}}, \mathtt{CH}'\rangle}{\langle \mathtt{PS}[\mathtt{t} \mapsto \overline{\sigma}_{\mathtt{L}}], \mathtt{CH}\rangle \hookrightarrow \langle \mathtt{PS}[\mathtt{t} \mapsto \overline{\sigma}'_{\mathtt{L}}], \mathtt{CH}'\rangle}$$

$$\frac{[\textbf{OP-PAR}]\quad \overline{\sigma}_{\mathtt{L}} = \langle \sigma, (\mathtt{e}_1 || \mathtt{e}_2)\rangle \quad \mathit{fresh}\ t_1, t_2 \quad \overline{\sigma}'_{\mathtt{L}} := \langle \mathtt{emp}_{\mathtt{sh}}, (\mathtt{join}\ t_1\ t_2)\rangle \quad \sigma = \sigma_1 \uplus \sigma_2 \quad \mathtt{PS}' := \mathtt{PS}[t \mapsto \overline{\sigma}'_{\mathtt{L}}][t_1 \mapsto \langle \sigma_1, \mathtt{e}_1\rangle][t_2 \mapsto \langle \sigma_2, \mathtt{e}_2\rangle]}{\langle \mathtt{PS}[t \mapsto \overline{\sigma}_{\mathtt{L}}], \mathtt{CH}\rangle \hookrightarrow \langle \mathtt{PS}', \mathtt{CH}\rangle}$$

$$\frac{[\textbf{OP-JOIN}]\quad \overline{\sigma}_{\mathtt{L}} = \langle \sigma, \mathtt{join}\ t_1\ t_2\rangle \quad \mathtt{PS} = \mathtt{PS}'[t_1 \mapsto \langle \sigma_1, \mathtt{skip}\rangle][t_2 \mapsto \langle \sigma_2, \mathtt{skip}\rangle] \quad \sigma' := \sigma \uplus \sigma_1 \uplus \sigma_2}{\langle \mathtt{PS}[t \mapsto \overline{\sigma}_{\mathtt{L}}], \mathtt{CH}\rangle \hookrightarrow \langle \mathtt{PS}'[t \mapsto \langle \sigma', \mathtt{skip}\rangle], \mathtt{CH}\rangle}$$

$$\frac{[\textbf{OP-JOIN-SKIP1}]\quad \overline{\sigma}_{\mathtt{L}} = \langle \_, \mathtt{join}\ t_1\ t_2\rangle \quad \mathtt{PS} = \mathtt{PS}'[t_1 \mapsto \langle \_, \mathtt{skip}\rangle][t_2 \mapsto \langle \_, \mathtt{e}\rangle]}{\langle \mathtt{PS}[t \mapsto \overline{\sigma}_{\mathtt{L}}], \mathtt{CH}\rangle \hookrightarrow \langle \mathtt{PS}[t \mapsto \overline{\sigma}_{\mathtt{L}}], \mathtt{CH}\rangle}$$

$$\frac{[\textbf{OP-JOIN-SKIP2}]\quad \overline{\sigma}_{\mathtt{L}} = \langle \_, \mathtt{join}\ t_1\ t_2\rangle \quad \mathtt{PS} = \mathtt{PS}'[t_1 \mapsto \langle \_, \mathtt{e}\rangle][t_2 \mapsto \langle \_, \mathtt{skip}\rangle]}{\langle \mathtt{PS}[t \mapsto \overline{\sigma}_{\mathtt{L}}], \mathtt{CH}\rangle \hookrightarrow \langle \mathtt{PS}[t \mapsto \overline{\sigma}_{\mathtt{L}}], \mathtt{CH}\rangle}$$

$$\frac{[\textbf{OP-JOIN-ERR1}]\quad \overline{\sigma}_{\mathtt{L}} = \langle \_, \mathtt{join}\ t_1\ t_2\rangle \quad \mathtt{PS} = \mathtt{PS}'[t_1 \mapsto \langle \_, \mathtt{e}\rangle][t_2 \mapsto \mathtt{error}]}{\langle \mathtt{PS}[t \mapsto \overline{\sigma}_{\mathtt{L}}], \mathtt{CH}\rangle \hookrightarrow \mathtt{error}}$$

$$\frac{[\textbf{OP-JOIN-ERR2}]\quad \overline{\sigma}_{\mathtt{L}} = \langle \_, \mathtt{join}\ t_1\ t_2\rangle \quad \mathtt{PS} = \mathtt{PS}'[t_1 \mapsto \mathtt{error}][t_2 \mapsto \langle \_, \mathtt{e}\rangle]}{\langle \mathtt{PS}[t \mapsto \overline{\sigma}_{\mathtt{L}}], \mathtt{CH}\rangle \hookrightarrow \mathtt{error}}$$

Figure 2.5: Semantic Rules: Machine Reduction

[OP–SEND] looks up for the program channel $\tilde{c}$ in the channel store CH and prepends the evaluation of the transmitted message e in the buffer corresponding to $\tilde{c}$. An error is flagged either when $\tilde{c}$ is not within the channels store, or when e is ill-formatted.

[OP–RECV] receiving a message over the program channel $\tilde{c}$, assumes that the channel is open $\mathtt{chan} = \mathtt{CH}(\tilde{c})$, and moreover that its corresponding buffer is non-empty, namely that $\mathtt{chan} = \mathtt{chan}' :: [\mathsf{val}]$. A read fetches the earliest message in the buffer (assuming a FIFO buffer), updates the channel store by popping out the received message to indicate that the reading has been consumed $\mathtt{CH}' := \mathtt{CH}[\tilde{c} \mapsto \mathtt{chan}']$, and updates the local stack with the received message $\sigma' := \langle s[\mathtt{res} \mapsto \mathsf{val}], h \rangle$. If the buffer is empty during a read request [[OP–RECV–BLOCK]], the running thread blocks in the current state waiting for its next turn and for the buffer to be populated. An important observation at this point is that this kind of behavior, and the impossibility to describe what is the expected communication scenario might turn this situation into one which contain a race, potentially type unsafe as well, or even leaking messages: if two threads, say t_1 and t_2, are both blocked in this order, waiting for the same buffer to be populated, but then the scheduler first unblocks t_2 (reminder: this thesis aims for a solution which detects error scenarios irrespective of the underlying scheduler implementation), then there is nothing in the machine or thread configuration to stop t_2 from reading the message destinated to t_1. Based on the implementation, such a scenario might or might not fault, potentially leading thus to an incoherent thread state. In most of the implementations, such a scenario would lead to a type error at best, and escape the type check, thus leaking data if type safety is not enforced.

The obvious limitations of such a semantic are the impossibility to account for (i) resource ownership transfer due to message passing, (ii) unexpected message transmissions, (iii) communication race scenarios in the context in which a channel is potentially shared across multiple parties.

The semantic error detection of [5] and the one presented this far capture the error states due to incorrect heap manipulation, but they lack the necessary instrumentation for detecting most of the communication related errors. Detecting unexpected messages or communication race conditions is far from trivial. One of this thesis' goals is to prove the absence of such errors, and show that the certified programs are error-free even at the operational semantic level.

$$\frac{[\textbf{OP–OPEN}]}{\texttt{CH}' = \texttt{CH}[\texttt{res}\mapsto[\,]]}{\langle\langle\sigma, \texttt{open()}\ \texttt{with}\ \mathit{sspec}\rangle, \texttt{CH}\rangle \hookrightarrow \langle\langle\sigma, \texttt{skip}\rangle, \texttt{CH}'\rangle}$$

$$\frac{[\textbf{OP–CLOSE}]}{\texttt{CH} = \texttt{CH}'[\tilde{\texttt{c}}\mapsto[\,]]}{\langle\langle\sigma, \texttt{close}(\tilde{\texttt{c}})\rangle, \texttt{CH}\rangle \hookrightarrow \langle\langle\sigma', \texttt{skip}\rangle, \texttt{CH}'\rangle} \qquad \frac{[\textbf{OP–CLOSE–LEAK}]}{\texttt{CH}(\tilde{\texttt{c}}) \neq [\,]}{\langle\langle\sigma, \texttt{close}(\tilde{\texttt{c}})\rangle, \texttt{CH}\rangle \hookrightarrow \texttt{error}}$$

$$\frac{[\textbf{OP–SEND}]}{\texttt{chan} = \texttt{CH}(\tilde{\texttt{c}}) \quad \texttt{chan}' := (\llbracket\texttt{e}\rrbracket_s) :: \texttt{chan} \quad \texttt{CH}' := \texttt{CH}[\tilde{\texttt{c}}\mapsto\texttt{chan}']}{\langle\langle\sigma, \texttt{send}(\tilde{\texttt{c}}, \texttt{e})\rangle, \texttt{CH}\rangle \hookrightarrow \langle\langle\sigma, \texttt{skip}\rangle, \texttt{CH}'\rangle}$$

$$\frac{[\textbf{OP–RECV}]}{\texttt{chan} = \texttt{CH}(\tilde{\texttt{c}}) \quad \texttt{chan} = \texttt{chan}' :: [\texttt{val}] \quad \texttt{CH}' := \texttt{CH}[\tilde{\texttt{c}}\mapsto\texttt{chan}'] \quad \sigma = \langle s, h\rangle \quad \sigma' := \langle s[\texttt{res}\mapsto\texttt{val}], h\rangle}{\langle\langle\sigma, \texttt{recv}(\tilde{\texttt{c}})\rangle, \texttt{CH}\rangle \hookrightarrow \langle\langle\sigma', \texttt{skip}\rangle, \texttt{CH}'\rangle}$$

$$\frac{[\textbf{OP–RECV–BLOCK}]}{\texttt{chan} = \texttt{CH}(\tilde{\texttt{c}}) \quad \texttt{chan} = [\,]}{\langle\langle\sigma, \texttt{recv}(\tilde{\texttt{c}})\rangle, \texttt{CH}\rangle \hookrightarrow \langle\langle\sigma, \texttt{recv}(\tilde{\texttt{c}})\rangle, \texttt{CH}\rangle}$$

Figure 2.6: Semantic Rules: Per-Thread Reduction

We also make the sound assumption that the `open`/`close`/`send`/`read` operations are *atomic*, and any reference to race scenarios from now on refers strictly to the communication race, not to the resource access race, be it main memory access or channel access. For example, given two threads t_1 and t_2 which share a common channel $\tilde{\texttt{c}}$ and which are expected to access the shared channel in this order, an access race between t_1 and t_2 is assumed to never happen if the communication operations are atomic. In other words, if thread t_2 has gained reading or writing permission to $\tilde{\texttt{c}}$ before t_1 did, then t_1 waits for t_2 to complete its task w.r.t. $\tilde{\texttt{c}}$ before gaining access to $\tilde{\texttt{c}}$. Even though an access race is avoided by assuming the atomicity of the communication, because t_1 was expected to complete prior to t_2 there is however a communication race since the communicating peer of t_1 is unaware that t_2 has first gained access to the shared channel.

Semantic Equivalence. In the style of trace semantics [5], two expressions $\texttt{e}_1$ and $\texttt{e}_2$ are said to be semantically equivalent if their reduction leads to the same set of reachable states. In this context we can describe sequential and parallel composition associativity as a congruence relation:

$$\mathtt{e}_1 \| (\mathtt{e}_2 \| \mathtt{e}_3) \equiv (\mathtt{e}_1 \| \mathtt{e}_2) \| \mathtt{e}_3$$
$$\mathtt{e}_1 ; (\mathtt{e}_2 ; \mathtt{e}_3) \equiv (\mathtt{e}_1 ; \mathtt{e}_2) ; \mathtt{e}_3$$

Parallel composition is also commutative, and `skip` is the identity element for parallel and sequential composition:

$$\mathtt{e}_1 \| \mathtt{e}_2 \equiv \mathtt{e}_2 \| \mathtt{e}_1 \qquad \mathtt{e} \| \mathtt{skip} \equiv \mathtt{e} \qquad \mathtt{skip} ; \mathtt{e} \equiv \mathtt{e}$$

Chapter 3

A SESSION LOGIC FOR DYADIC COMMUNICATION PROTOCOLS

This chapter introduces a session logic with a novel, yet natural use of disjunction to specify and verify the implementation of communication protocols. Even though it might seem like it is a rather simple problem at a first glance, with a clear-cut solution, we tackle a number of interesting issues for which we cater elegant solutions.

We briefly list down most of these issues together with a glimpse to our proposed solution and detail them in the next sections. We start by making the observation that the majority of existing solutions which formalize communication protocols are limited to check that the two communicating peers agree upon the transmitted message type. The current thesis goes beyond this traditional type check into treating the exchanged messages as logical formulae, offering therefore a doorway to the specification of more complex conversation properties.

Next, it is worth mentioning that even though the main purpose of this chapter is to formalize the conversation between two peers, we describe a way to introduce third parties into the communication, without breaking the two-peers communication paradigm. Third-parties are involved in the communication by means of delegation. The formalization of delegation in the context of message-passing is not new, however this thesis proposes a novel approach in that it uses higher-order channels to specify and implement delegation. The advantage over traditional

channels used in the state of the art is that the proposed session logic treats all transmissions uniformly, using the same logical operators at the specification level and the same communication primitives at the program level to transmit data as well as channel or data structures references.

The use of disjunctions to model both internal and external choices, lifts the burden of supporting different operands to distinguish between the choices, hence a simpler specification language and a less constrained programming language.

Similar to the work of Villard et al. [68], this thesis considers an extension of separation logic, supporting therefore heap-manipulating programs and copyless message passing. The logical formulae on protocols can also be localised to each channel and may be freely passed through procedural boundaries. As channels can support a variety of messages, the logic treats the read content as dynamically typed where conditionals are dispatched based on the received types. Alternatively, the current approach also guarantees type-safe casting via verifying communication safety. Moreover, it goes beyond such type cast safety by ensuring that heap memory and the properties of the values transmitted over the channels are suitably captured.

Lastly by using a subsumption between different communication protocols, this approach allows specifications on channels to differ between threads without sacrificing correctness and safety, and ensuring that the specifications remain compatible at each join point. This compatibility check is specifically useful for preventing intra-channel deadlocks.

3.1 Motivation

The proposed logic is introduced by means of a classic business protocol which involves a *Buyer* and a *Seller*. The conversation starts with the Buyer sending the Seller a product name encoded as a `String` object. The Seller replies with an `int` representing the product's price. Should the Buyer be satisfied with the offered price, she sends her address as an object of type `Addr` and waits for her peer, the Seller, to reply back with the delivery date captured as an object of type `Date`. Otherwise, the Buyer quits the conversation. This example is modeled as binary session in Fig. 3.1. In a binary session, one channel is typically sufficient for the communication between two parties. This simple protocol can be summarized by the following binary session type [45] which represents the communication pattern w.r.t. the Buyer:

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

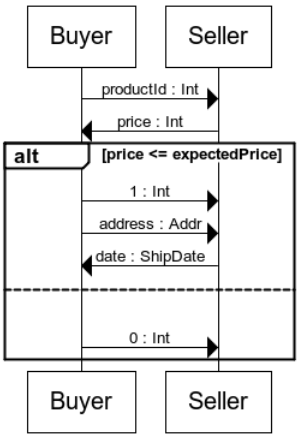


Figure 3.1: Sequence diagram for the Buyer-Seller protocol

The dual of the above session type (where duality is denoted by the $\sim$ unary operator) corresponds to the Seller's communication pattern:

$$\texttt{seller\_ty} \triangleq ?\texttt{String};!\texttt{int};?\{\texttt{ok}:?\texttt{Addr};!\texttt{Date};\texttt{end},\ \texttt{quit}:\texttt{end}\}.$$

$$\texttt{seller\_ty} \equiv \sim\texttt{buyer\_ty}.$$

In the above session type, $!\texttt{t}$ denotes the output of a value of type `t`, while its dual $?\texttt{t}$ denotes the input of a value of type `t`. Moreover, the type $!\{\texttt{ok} : ...,\ \texttt{quit} : ...\}$ denotes an internal choice (decision based on local values) with the `ok` and `quit` options, encoded as labels transmitted over the channel. the type $?\{\texttt{ok} : ...,\ \texttt{quit} : ...\}$, denotes an external choice (decision based on received labels). The type `end` indicates the termination of the conversation for a given channel. Traditionally, a program implementing the above protocol uses specialized switch constructs like `outbranch` and `inbranch` to model the internal and external choices respectively:

```
1  void buyer(buyer_ty c,String p)
2  { send(c,p);
3    Double price = receive(c);
4    Double budget = budget();
5    if (price <= budget){
6      outbranch(c,ok){
7        Address a = address();
8        send(c,a);
9        Date sd = receive(c);
10     }
11   } else {
12     outbranch(c,quit);{
13     }
14   }
15 }
```

```
void seller(seller_ty c)
{ String p = receive(c);
  send(c,get_price(p))
  inbranch(c){
    case ok: {
      Addr a = receive(c);
      ShipDate sd = ship_date(p,a);
      send(c,sd);
    }
    case quit: {
    }
  }
}
```

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

$$\texttt{buyer\_ch} \triangleq \texttt{!String}; \texttt{?int}; ((\texttt{!1}; \texttt{!Addr}; \texttt{?Date}) \vee \texttt{!0})$$

$$\texttt{seller\_ch} \triangleq \texttt{?String}; \texttt{!int}; ((\texttt{?1}; \texttt{?Addr}; \texttt{!Date}) \vee \texttt{?0})$$

$$\texttt{buyer\_ch} \equiv \sim\texttt{seller\_ch}$$

Superficially, this logical specification looks similar to a session type; however, there are several notable differences. Firstly, there is no need for dedicated begin or end session declarations since the logic protocol is expected to be captured locally after creation. Secondly, it uses disjunction[1] instead of the more specialized notations for internal and external choices. Thirdly, instead of message labels (such as ok and quit), the logical protocol uses values (such as 1 or 0) or even types themselves to capture the distinct scenarios for internal and external choices. This supports direct usage of conditionals to express choices which are naturally modelled by disjunctive formulae during program reasoning. Most importantly, instead of limiting the message description to types or values, the logic is able to capture more general properties (including ghost properties) to be passed into the channels. One benefit of such message description is

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

that it facilitates the verification of functional correctness properties, which could go beyond communication safety. Another benefit of supporting message passing described by logical formulae is that of being able to use higher-order channels to model delegation, where channels and their expected specifications are passed as messages.

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

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

The channel message is described using a specification formula as per Fig. 2.2. When the spatial formula is `emp`, `emp` is omitted and replaced by abbreviated notations, such as $?\texttt{r:int} \cdot \texttt{r}{>}1$ as a short-hand for $?\texttt{r} \cdot \texttt{emp} \wedge \texttt{r:int} \wedge \texttt{r}{>}1$. Furthermore, the following abbreviated notations are also accepted: $!\texttt{String}$ as a shorthand for $!\texttt{r} \cdot \texttt{emp} \wedge \texttt{r:String} \wedge \texttt{true}$, $?1$ as a short-hand for $?\texttt{r} \cdot \texttt{emp} \wedge \texttt{r:int} \wedge \texttt{r}{=}1$. 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.

Another benefit of the current approach worth to be highlighted is the ability to use specialized protocols, where a thread's specification and a channel specification need not be identical. As an example, the following code snippet specifies a stronger specification for the seller's communication, imposing a lower bound on the price of the products offered by this seller. The price lower bound is 10 units, as per the following specification:

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

The following code snippet depicts a program implementing the `seller_sp` protocol. Note that it directly uses conditionals instead of the specialized switch constructs specific to the π-calculus in general, and session types approaches in particular.

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

```
4    void buyer(Chan c,String p)
5       requires C(c,buyer_chan)
6       ensures  C(c,emp);
7  { send(c,p);
8    Double price = receive(c);
9    Double budget = budget();
10   if (price <= budget){
11     send(c,1);
12     Address a = address();
13     send(c,a);
14     Date sd = receive(c);
15   } else {
16     send(c,0)
17   }
18 }
19
```

```
20 void seller(Chan c)
21      requires C(c,seller_sp)
22      ensures  C(c,emp);
23 { String p = receive(c);
24   send(c,get_price(p));
25   int usr_opt = receive(c);
26   if(usr_opt == 1){
27     Addr a = receive(c);
28     Date sd = ship_date(p,a);
29     send(c,sd);
30   }
31 }
32
33 int get_price(String p)
34      requires emp
35      ensures  res > 0;
```

open opens the channel which serves as the communication medium between the Buyer and Seller. The open method, with attached channel specification, is called from within the main program. The alias of the opened channel which bears the buyer_chan specification is passed to the thread implementing the Buyer, while the other alias with its dual specification seller_chan is passed to the Seller's thread. Each thread implements a different protocol specification, specifications which are consistent with the channel's view of the communication. The Seller's thread specification seller_sp imposes a stronger property over the sent price, namely that the price's lower bound is 10, that is !r:int · r>10 instead of !r:int · r>0 which is captured by the initial channel specification seller_chan. This behavior implies the verification of certain subsumption relations between the channel specification passed to a thread and the actual thread's specification. In a verification approach which discharges proof obligations in the form of entailment checks, the subsumption relation is defined as $\mathcal{C}(\texttt{c},\texttt{buyer\_chan}) \vdash \mathcal{C}(\texttt{c},\texttt{buyer\_chan})$ for the buyer thread. This entailment relation trivially holds. The seller process involves checking that $\mathcal{C}(\texttt{c},\texttt{seller\_chan}) \vdash \mathcal{C}(\texttt{c},\texttt{seller\_sp})$

holds. This second entailment also succeeds, but it's not as trivial as in the buyer's case. We present the rules for subsumption checks in the context of dyadic communication in Sec. 3.2. Intuitively though, the subsumption for receive is covariant, while the sending operations involve contravariant checks, as illustrated in the proof-tree sketch below:

$$\dfrac{\dfrac{\dfrac{\dfrac{\texttt{true}}{?\texttt{String} \vdash ?\texttt{String}} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\texttt{true}}{\texttt{r>10} \vdash \texttt{r>0}}}{!\texttt{r}\cdot\texttt{r>0} \vdash !\texttt{r}\cdot\texttt{r>10}}\text{(contravariant subsumtion)} \quad \ldots}{!\texttt{r:int}\cdot\texttt{r>0};(\ldots) \vdash !\texttt{r:int}\cdot\texttt{r>10};(\ldots)}\text{(protocol check)}}{?\texttt{String};!\texttt{r:int}\cdot\texttt{r>0};(\ldots) \vdash ?\texttt{String};!\texttt{r:int}\cdot\texttt{r>10};(\ldots)}\text{(protocol check)}}{\texttt{seller\_chan} \vdash \texttt{seller\_sp}}\text{(protocol check)}}{\mathcal{C}(\texttt{c},\texttt{seller\_chan}) \vdash \mathcal{C}(\texttt{c},\texttt{seller\_sp})}\text{(channel match)}$$

Finally, another difference w.r.t. the current state of the art, specially that of Villard et al. [93], is that the opening of a channel with attached channel specification releases a single channel reference in the program state. However, this reference may have aliases such that both the channel specification as well as its dual are made available in the abstract state. In contrast, the work of Villard et al. adopts a less natural behavior where the opening of a channel creates two ends of the considered channel in order to support both the send as well as the receive of the same message.

3.2 A Dyadic Session Logic

The proposed session specification language is developed on top of the specification language presented in Fig. 2.2. The additional constructs to the general specification language as described in Fig. 3.2, comprises the terms and operands which help to specify a session: the sending, $!\texttt{r}\cdot\Delta$, or receiving $?\texttt{r}\cdot\Delta$ of a message referenced by $\texttt{r}$ and described by Δ, inaction $\texttt{emp}$, sequential composition $\overline{\texttt{L}}_1;\overline{\texttt{L}}_2$ and choice $\overline{\texttt{L}}_1 \vee \overline{\texttt{L}}_2$.

A specification for channel $\texttt{c}$ is represented by an abstract predicate $\mathcal{C}(\texttt{c},\overline{\texttt{L}})$ which simply binds a communication specification $\overline{\texttt{L}}$ to a channel $\texttt{c}$. The specification language in Fig. 2.2 is therefore updated as below, where only the derivation of spatial formula is revised:

$$\textit{Spatial Formula} \quad \kappa ::= \texttt{emp} \mid \texttt{v}{\mapsto}\texttt{c}(\texttt{v}^*) \mid \texttt{p}(\texttt{v}^*) \mid \kappa_1 * \kappa_2 \mid \mathcal{C}(\texttt{v},\overline{\texttt{L}})$$

The rules to obtain dual specifications are depicted in Fig. 3.3. We remind the reader that a conjunctive abstract program state Δ comprises two parts: the heap part κ written as a

Session Specification $\overline{L}$ ::=	
Inaction	$\texttt{emp}$
Receive	$\mid ?\texttt{r} \cdot \Delta$
Send	$\mid !\texttt{r} \cdot \Delta$
Sequence	$\mid \overline{\texttt{L}}_1; \overline{\texttt{L}}_2$
Choice	$\mid \overline{\texttt{L}}_1 \vee \overline{\texttt{L}}_2$
Duality	$\mid \sim\overline{\texttt{L}}$

Figure 3.2: A Specification Language for Dyadic Communication

$$\sim!\texttt{r} \cdot \Delta \stackrel{\text{def}}{=} ?\texttt{r} \cdot \Delta \qquad \sim?\texttt{r} \cdot \Delta \stackrel{\text{def}}{=} !\texttt{r} \cdot \Delta$$
$$\sim(\overline{\texttt{L}}_1 \vee \overline{\texttt{L}}_2) \stackrel{\text{def}}{=} \sim\overline{\texttt{L}}_1 \vee \sim\overline{\texttt{L}}_2 \qquad \sim(\overline{\texttt{L}}_1; \overline{\texttt{L}}_2) \stackrel{\text{def}}{=} \sim\overline{\texttt{L}}_1; \sim\overline{\texttt{L}}_2$$

Figure 3.3: Rules for Dual Specification.

spatial formula and the pure part π written in convex polyhedra domain where π may contain Presburger Arithmetic operations, Boolean operands and pointer equality. We also support disjunctive program states, denoted by Φ and which capture the different program traces, denoted by Δ. The logic only considers DNF formulae, which normalized as per [20].

The separation logic prover described in Sec. 2.5 is used for checking whether the antecedent formula $\Delta_{\texttt{a}}$ subsumes the consequent formula $\Delta_{\texttt{c}}$, both written in the combined abstract domain described in Sec. 2.3, with the adjustments of the spatial formula as depicted above. To support entailment checking over session logic formulae, the entailment checker of Sec. 2.5 is extended with the rules of Fig. 5.4. As mentioned in the Sec. 3.1 and as formally described by the entailment rules, 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.

Definition 2 (Well-Formed Choice)**.** *A disjunctive protocol specification,* $\overline{\texttt{L}}_1 \vee \overline{\texttt{L}}_2$*, is said to be well-formed with respect to* $\vee$ *if and only if* $\texttt{disjoint}(\overline{\texttt{L}}_1 \vee \overline{\texttt{L}}_2)$ *holds, where* `disjoint` *is recursively defined as follows:*

$$\frac{[\textbf{ENT-SEND}]}{\Delta_2 \vdash \Delta_1} \quad \frac{[\textbf{ENT-RECEIVE}]}{\Delta_1 \vdash \Delta_2}$$
$$\frac{!\mathtt{r} \cdot \Delta_1 \vdash !\mathtt{r} \cdot \Delta_2}{?\mathtt{r} \cdot \Delta_1 \vdash ?\mathtt{r} \cdot \Delta_2}$$

$$\frac{[\textbf{ENT-SEQUENCE}]}{\mathtt{e} ::= ?\mathtt{r} \cdot \Phi \mid !\mathtt{r} \cdot \Phi \quad \mathtt{e}_1 \vdash \mathtt{e}_2 \quad \overline{\mathtt{L}}_1 \vdash \overline{\mathtt{L}}_2}{\mathtt{e}_1; \overline{\mathtt{L}}_1 \vdash \mathtt{e}_2; \overline{\mathtt{L}}_2} \qquad \frac{[\textbf{ENT-CHAN-MATCH}]}{\overline{\mathtt{L}}_1 \vdash \overline{\mathtt{L}}_2}{\mathcal{C}(\mathtt{c}, \overline{\mathtt{L}}_1) \vdash \mathcal{C}(\mathtt{c}, \overline{\mathtt{L}}_2)}$$

Figure 3.4: Selected Entailment Rules for a Dyadic Session Logic.

$$\frac{\bigwedge_{\mathtt{i}=1..\mathtt{n}} \left(\bigwedge_{\mathtt{j}=1..\mathtt{n}} \mathtt{j} \neq \mathtt{i} \Rightarrow \mathtt{disjointS}(\overline{\mathtt{L}}_\mathtt{i} \vee \overline{\mathtt{L}}_\mathtt{j}) \right)}{\mathtt{disjointS}\left(\bigvee_{\mathtt{i}=1..\mathtt{n}} \overline{\mathtt{L}}_\mathtt{i} \right)} \qquad \frac{\bigwedge_{\mathtt{i}=1..\mathtt{n}} \left(\bigwedge_{\mathtt{j}=1..\mathtt{n}} \mathtt{j} \neq \mathtt{i} \Rightarrow \mathtt{disjointR}(\overline{\mathtt{L}}_\mathtt{i} \vee \overline{\mathtt{L}}_\mathtt{j}) \right)}{\mathtt{disjointR}\left(\bigvee_{\mathtt{i}=1..\mathtt{n}} \overline{\mathtt{L}}_\mathtt{i} \right)}$$

$$\frac{\mathtt{UNSAT}(\Delta_1 \wedge [\mathtt{r}_1/\mathtt{r}_2]\Delta_2)}{\mathtt{disjointS}(!\mathtt{r}_1 \cdot \Delta_1; \overline{\mathtt{L}}_1 \ \vee \ !\mathtt{r}_2 \cdot \Delta_2; \overline{\mathtt{L}}_2)} \qquad \frac{\mathtt{UNSAT}(\Delta_1 \wedge [\mathtt{r}_1/\mathtt{r}_2]\Delta_2)}{\mathtt{disjointR}(?\mathtt{r}_1 \cdot \Delta_1; \overline{\mathtt{L}}_1 \ \vee \ ?\mathtt{r}_2 \cdot \Delta_2; \overline{\mathtt{L}}_2)}$$

$$\frac{\mathtt{true}}{\mathtt{disjointS}(!\mathtt{r} \cdot \Delta; \overline{\mathtt{L}})} \qquad \frac{\mathtt{true}}{\mathtt{disjointR}(?\mathtt{r} \cdot \Delta; \overline{\mathtt{L}})}$$

$$\frac{\mathtt{disjointS}(\overline{\mathtt{L}}) \vee \mathtt{disjointR}(\overline{\mathtt{L}})}{\mathtt{disjoint}(\overline{\mathtt{L}})}$$

In other words, a disjunctive channel specification is well-formed as per Def. 2 if and only if all the message choices are described by disjoint formulae. Moreover, all the choices must refer to the same kind of operation, that is they either all represent reading operations, or they are all sending operations.

Definition 3 (Well-Formed Channel Specification)**.** *A channel specification* $\overline{\mathtt{L}}$ *is said to be well-formed if and only if* $\overline{\mathtt{L}}$ *contains only well-formed choices.*

For the remaining of this chapter we consider only well-formed channel specifications even when this property is not explicitly mentioned.

3.3 Forward Verification

Language. This subsection briefly reiterates through the concurrent imperative language presented in Fig. 2.1. The target language now represents a language where *sspec* is instantiated to the dyadic session specification language $\overline{\mathtt{L}}$ described in Fig. 3.2:

$$\textit{Expressions} \quad e \quad ::= \quad ... \\ \mid \texttt{open}() \texttt{ with } \overline{L} \mid \texttt{close}(v) \mid \texttt{receive}(v) \mid \texttt{send}(v, e)$$

In the target language threads communicate through channels. A channel is created by `new Chan()` but its usage for transmitting messages is deferred until after is has been opened. There are two common kinds of channels, monolithic and double-ended. Monolithic channels allow a channel reference to have multiple aliases to be used by multiple parties. Double-ended channels treat a channel w.r.t. its two ends which are distinct references to the same channels and which are to be later used by exactly two parties. The target language of this thesis assumes the more general monolithic channels, but the proposed reasoning system can support either model by simply using a different set of specifications for double-ended channels.

Proof Rules. As mentioned in the earlier chapter, the proof rules accompanying the proposed logic are in the style of HIP/SLEEK [20]. For the communication related operations though, Fig. 3.5 introduces the verification rules in terms of abstract specifications as pre and post-conditions. It is possible to write the proof rules in terms of abstract specification since the assumption is that the calls to these methods are atomic, therefore they are treated as atomic commands.

A channel can be opened by `open` with some channel specification $\overline{\mathtt{L}}$. As a result of opening a channel, the abstract state contains two aliases of the same channel, one with attached specification $\overline{\mathtt{L}}$ and the other corresponding to the complimentary specification $\sim\overline{\mathtt{L}}$ as depicted by the specification [OPEN] in Fig. 3.5. Dually, a channel is safe to be closed (or destroyed) only when both aliases are available in the abstract program state and both have consumed their attached specifications as indicated in [CLOSE].

A general context which prepares a channel for transmission and safe closing is exemplified below:

```
1  open(c) with L̄;
2  /* C(c,L̄) * C(c,∼L̄) */
3  /* C(c,L̄) */          ||  /* C(c,∼L̄) */
4  Process1(c);           ||  Process2(c);
5  /* C(c,emp) */        ||  /* C(c,emp) */
6  /* C(c,emp) * C(c,emp) */
7  close(c);
```

$$[\textbf{OPEN}]$$
$$\vdash \{\texttt{emp}\}\ \texttt{open()}\ \texttt{with}\ \overline{\texttt{L}}\ \{\mathcal{C}(\texttt{res}, \overline{\texttt{L}}) * \mathcal{C}(\texttt{res}, \sim\overline{\texttt{L}})\}$$

$$[\textbf{CLOSE}]$$
$$\vdash \{\mathcal{C}(\texttt{c}, \texttt{emp}) * \mathcal{C}(\texttt{c}, \texttt{emp})\}\ \texttt{close(c)}\ \{\texttt{emp}\}$$

$$[\textbf{SEND}]$$
$$\vdash \{\mathcal{C}(\texttt{c}, !\texttt{r} \cdot \texttt{V(r)}; \texttt{V}^{\texttt{rest}}) * \texttt{V(x)}\}\ \texttt{send(c, x)}\ \{\mathcal{C}(\texttt{c}, \texttt{V}^{\texttt{rest}})\}$$

$$[\textbf{RECV}]$$
$$\vdash \{\mathcal{C}(\texttt{c}, ?\texttt{r} \cdot \texttt{V(r)}; \texttt{V}^{\texttt{rest}})\}\ \texttt{receive(c)}\ \{\mathcal{C}(\texttt{c}, \texttt{V}^{\texttt{rest}}) * \texttt{V(res)}\}$$

Figure 3.5: Communication Primitives

The send and receive operations are also dual to each other, as depicted by the [SEND] and [RECV] rules. While the sending thread should own the resource that is being transmitted and lose its ownership once the operation has been executed, the receiving gains full ownership of the received message. Besides ownership reasoning, the specification also mentions what is the expected communication specification and how this changes once the operation completes its execution. The word `res` in the specification of `receive` is a reserved keyword denoting the result of the considered expression evaluation (the receiving of a message in this case). `V` is an instantiable higher-order variable, while $\texttt{V(r)}$ indicates that the instantiation of `V` contains a reference to the instantiable first order variable `r`. Below is the sketch of an entailment proof to highlight the manipulation of higher-order variables:

$$\dfrac{\dfrac{\dfrac{\dfrac{\texttt{S}=\{\texttt{emp} \wedge \texttt{V(r)}=(\texttt{emp} \wedge \texttt{r}>0)\}}{\texttt{r}>0 \vdash \texttt{V(r)} \rightsquigarrow \texttt{S}}}{?\texttt{r} \cdot \texttt{r}>0 \vdash ?\texttt{r} \cdot \texttt{V(r)} \rightsquigarrow \texttt{S}}}{\mathcal{C}(\texttt{c}, ?\texttt{r} \cdot \texttt{r}>0) \vdash \mathcal{C}(\texttt{c}, ?\texttt{r} \cdot \texttt{V(r)}) \rightsquigarrow \texttt{S}} \qquad \dfrac{\dfrac{\texttt{true}}{\texttt{emp} \wedge \texttt{V(r)}=(\texttt{emp} \wedge \texttt{r}>0) \wedge \texttt{x}>2 \vdash \texttt{emp} \wedge \texttt{x}>0}}{\texttt{emp} \wedge \texttt{V(r)}=(\texttt{emp} \wedge \texttt{r}>0) \wedge \texttt{x}>2 \vdash \texttt{V(x)}}}{\mathcal{C}(\texttt{c}, ?\texttt{r} \cdot \texttt{r}>0) \wedge \texttt{x}>2 \vdash \mathcal{C}(\texttt{c}, ?\texttt{r} \cdot \texttt{V(r)}) * \texttt{V(x)}}$$

In a dyadic session, one channel is typically sufficient to serve as the communication medium between two parties. Let us denote the two communicating parties by programs $\texttt{P(c)}$ and $\texttt{Q(c)}$, where `c` is the communication channel. Apart from the communication channel specification the current framework also supports the specification of each communicating party, for example $\overline{\texttt{L}}^{1}_{\texttt{pre}}$ and $\overline{\texttt{L}}^{2}_{\texttt{pre}}$ as the communication preconditions for `P` and `Q`, respectively. In general, the

specifications of programs is written as follows:

```
1   t P(Channel c)                          t Q(Channel c)
2     requires C(c, L̄¹_pre) * Φ¹_pr           requires C(c, L̄²_pre) * Φ²_pr
3     ensures  C(c, L̄¹_post) * Φ¹_po .        ensures  C(c, L̄²_post) * Φ²_po .
```

As discussed, the `close` operation is required for the communication to have been completed with respect to the dual specifications in order to be safely fired. In the following example the condition for calling `close` does not hold, hence the verification fails because the communication specification has not been fully consumed. The example uses a recursive session specification $\overline{\mathtt{L}}_2$:

$$\overline{\mathtt{L}}_2 \triangleq !\mathtt{String};\overline{\mathtt{L}}_2$$

```
1   open(c) with L̄₂;
2   /* C(c, L̄₂) * C(c, ∼L̄₂) */
3   /* C(c, L̄₂) */          ||   /* C(c, ∼L̄₂) */
4   for(i=1 to 5)           ||   for(i=1 to 10)
5      send(c,i);           ||      int x = receive(c);
6   /* C(c, L̄₂) */          ||   /* C(c, ∼L̄₂) */
6   /* C(c, L̄₂) * C(c, ∼L̄₂) */
7   close(c); // FAILS!
```

The channels used by the dyadic framework are dynamically typed. Dynamic types in the proposed langauge are denoted by `Dyn`. For instance, the type signature for `send` and `receive` are dynamically typed as depicted in the following code snippet:

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


...
send(c,e);
send(c,"ABC");
int    n = (int) receive(c);
String r = (String) receive(c);
```

The proposed automated verification rules help to guarantee communication safety via type-safe casting. One way to support dynamic typed values is by using a specialized `switch` construct, as follows:

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

Alternatively, it may also be supported via type testing with conditional constructs, as follows:

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

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

$$\overline{\mathtt{L}}_3 \triangleq !\mathtt{Object}; (\overline{\mathtt{L}}_3 \vee !0)$$

However, relying solely on type-safe casting without run-time type testing, a channel specification would have to be written as below, where each disjunct employs the same type for the first operation:

$$\overline{\mathtt{L}}_4 \triangleq !\mathtt{Object}; (!1; \overline{\mathtt{L}}_4 \vee !0)$$

3.4 Higher-Order Session Logic

The communication between two parties may often require additional information which could only be provided by a third party. For example, in the Buyer-Seller protocol introduced in Sec. 3.1, it may be common for Seller to engage a third party to handle the shipping to Buyer. To do that, Seller would have to transmit to the third party the channel used for the communication with Buyer, or in other words it *delegates* the communication with Buyer to the third party.

Delegation is a crucial feature for supporting additional communicating parties within a

dyadic designed communication. An extra benefit for supporting delegation is that it allows channel reuse once the communication is delegated to a third party. Having designed higher-order channels and specifications as detailed in Sec. 3.2 and Sec. 3.3, the current framework does not need any extra feature to support delegation. The theory developed in the said sections sufficies to perform and reason about delegation.

To demonstrate the power of using a higher-order approach, this section introduces and reasons about an example which is borrowed from [28]. The example involves three parties, namely Buyer, Seller and Shipper. The communication unfolds by first transmitting a product identifier from Buyer to Seller. Seller then replies back with a price for the requested product. According to some internal constraints, Buyer decides whether to accept the deal or not. Should Buyer decide to proceed with the transaction, Seller establishes a connection with Shipper in order to arrange for the transportation of the product. In her relation with Shipper, Seller forwards the necessary information about the product to be shipped and delegates the rest of the communication with Buyer to Shipper. Finally, Buyer and Shipper agree upon the final details related to the transportation, that is delivery address and shipping date. As an overview of the communication described above, consider a less formal communication specification using sequence diagrams as depicted in Fig. 3.6.

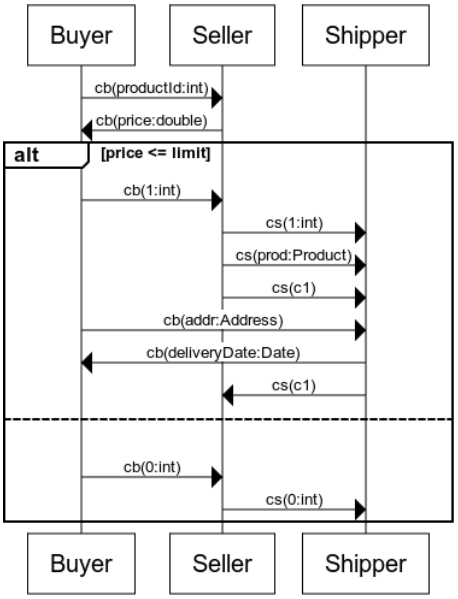


Figure 3.6: Sequence Diagram of Buyer-Seller-Shipper Protocol

The view of Buyer and Shipper w.r.t. the communication in Fig. 3.6, is formally written as

```
data Double {int i; //integer part          data Date {int year; int month;
             int f; //fractional part }                int day; }
data Prod   {int id;}                       data Addr { int nr; String city;}
```

Figure 3.7: User-defined Data Structures

channel specifications in the proposed logic as follows:

$$\texttt{buy\_sp} \triangleq \texttt{!int}; \texttt{?r:Double} \cdot \texttt{sprice(r)}; ((\texttt{!1}; \texttt{!Addr}; \texttt{?Date}) \vee \texttt{!0})$$

$$\texttt{ship\_sp} \triangleq (\texttt{?1}; \texttt{?Prod}; \texttt{?r:Chan} \cdot \mathcal{C}(\texttt{r}, \texttt{?Addr}; \texttt{!Date}); \texttt{!r:Chan} \cdot \mathcal{C}(\texttt{r}, \texttt{emp})) \vee \texttt{?0}$$

where most of the data structures used by the examples in this section are defined in Fig. 3.7, and the auxiliary functions are summarized in the Appendix. `sprice` is a first order predicate describing a price strictly greater than zero and stored as a double value:

$$\texttt{sprice(root)} \triangleq \exists \texttt{i}, \texttt{f} : \texttt{root} \mapsto \texttt{Double}\langle \texttt{i}, \texttt{f} \rangle \wedge \texttt{i} \geq 0 \wedge \texttt{f} \geq 0 \wedge \texttt{i} + \texttt{f} > 0$$

The `buy_sp` specification describes the communication pattern followed by Buyer. From Buyer's perspective the delegation is transparent, therefore not explicitly captured by its specification. Seller's specification however, needs to explicitly add the delegation details within the specification of the channel used for the conversation with Shipper. Therefore, both Seller and Shipper capture the delegation details as depicted in `ship_sp`. More precisely, Seller delegates the communication to Shipper via $\texttt{!r:Chan} \cdot \mathcal{C}(\texttt{r}, \texttt{?Addr}; \texttt{!Date})$, and Shipper receives it via $\texttt{?r:Chan} \cdot \mathcal{C}(\texttt{r}, \texttt{?Addr}; \texttt{!Date})$. The specification of the delegation must be precise, reflecting the state of the transmitted channel. This specification must be ensured by the thread which initiates the delegation and can be assumed by the delegatee.

An implementation of the Buyer-Seller-Shipper protocol which satisfies the `buy_sp`, `ship_sp` and their dual specifications is listed below:

```
1   open(cb) with buy_sp;
2   open(cs) with ship_sp;
3   int prod = get_prod_id();
4   Addr a = get_addr();
5   Double budget = get_budget();

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

9   close(cb);
10  close(cs);
```

where the methods which implement Seller and Shipper might have the following definition (an implementation of Buyer is given in the Appendix):

```
1  void seller(Channel cb, Channel cs)
2    requires C(cb, ~buy_sp) * C(cs, ~ship_sp)
3    ensures  C(cb, emp) * C(cs, emp);
4  { int id = receive(cb);
5    send(cb,get_price(id));
6    int usr_opt = receive(cb);
7    if(usr_opt == 1){
8      send(cs,1);
9      Prod p = get_prod(id);
10     send(cs,p);
11     send(cs,cb);
12     cb = receive(cs);
13   } else if (usr_opt == 0){
14     send(cs,0)
15   } else {
16     assert(false);
17   }
18 }
```

```
1  void shipper(Channel c)
2    requires C(c, ship_sp)
3    ensures  C(cb, emp);
4  { int usr_opt = receive(c);
5    if(usr_opt == 1){
6      Prod p     = receive(c);
7      Channel cd = receive(c);
8      Addr a     = receive(cd);
9      Date sd    = ship_date(a,p);
10     send(cd,sd);
11     send(c,cd);
12   } else if (usr_opt != 0){
13     assert(false);
14   }
15 }
16
17
18
```

The verification phase involves checking that the above implementation follows its corresponding specifications, or in other words that each method plays its designated communication role. The role of Seller poses the most challenges and it is the most interesting to elaborate, since Seller should not only carry the conversation with Buyer, but it should also bridge the

conversation between Buyer and Shipper. The next code snippet highlights some of the salient points in the implementation and verification of Seller:

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

Given the forward verification style of our approach, the symbolic execution of the `seller` method starts by assuming its precondition, namely $\mathcal{C}(\texttt{cb}, \sim\texttt{buy\_sp}) \ * \ \mathcal{C}(\texttt{cs}, \sim\texttt{ship\_sp})$, where $\sim$`buy_sp` and $\sim$`ship_sp` are computed using the duality rules from Fig. 3.3. Next, after the symbolic execution of line 1, the program's symbolic state reflects the receiving of an `int` value over channel `cb` by removing ?`int` from the specification of `cb`. Formally, if the precondition of `receive` holds, the next symbolic state comprises the frame resulted after proving the precondition along with the method's postcondition. The same rationale applies for lines 2-3 and 5-8, with the remark that line 3 is represented as a choice in the specification of `cb`: $((?1; ...) \ \vee \ ?0)$, while line 8 denotes a delegation. It is important to note that our approach is composable, e.g. the verification of the `send` method follows the same rule whether it is used

for delegation or merely for sending an int value over a specified channel.

3.5 Full Expressivity of Separation Logic

This section demonstrates the expressiveness of the proposed approach with a very simple business protocol example between two parties, namely Buyer and Seller. Buyer recursively sends a read-only list of product identifiers, while Seller responds with a price for each product identifier. The sequence diagram of this protocol is illustrated in Fig. 3.8

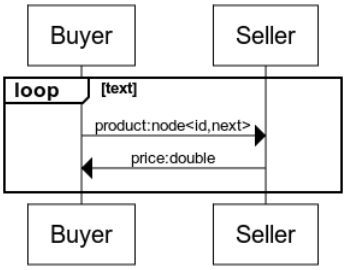


Figure 3.8: Sequence Diagram for a Buyer-Seller Recursive Protocol

Given the following data node declaration:

```
struct node { int val; struct node *next; };
```

a separation logic based specification language conveniently provides support in writing the necessary heap predicates, such as the linked list definition below, which could then be used by Buyer to send to Seller a list of product identifiers. Even though this definition contains ghost values, the receiver side can still exploit this ghost values in reasoning about the received message/choice:

$$\texttt{ll}(\texttt{root}) \triangleq \texttt{root} = \texttt{null} \vee \exists\, \texttt{q} \cdot \texttt{root} \mapsto \texttt{node}\langle \_, \texttt{q}\rangle * \texttt{ll}(\texttt{q})$$

ll defines a linked-list recursively using disjunction to capture both the base case, root $=$ null, as well as the inductive case, $\exists\, \texttt{q} \cdot \texttt{root} \mapsto \texttt{node}\langle \_, \texttt{q}\rangle * \texttt{ll}(\texttt{q})$. This kind of predicates can also be used to describe a protocol choice. However, in order to avoid ambiguous protocols we must impose disjoint conditions within the disjunction, or in other words, the protocol must be well-formed, as per Def. 2. For instance, the following channel is ambiguous:

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

while the following is unambiguous:

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

The communication specification between Buyer and Seller can then be written using inductive definitions, as below:

$$\begin{aligned} \mathtt{buy\_lsp} \triangleq \ & !\mathtt{p}:\mathtt{node} \cdot \mathtt{p} = \mathtt{null} \vee \\ & !\mathtt{p}:\mathtt{node} \cdot \ \mathtt{p} \mapsto \mathtt{node}\langle \_, \_ \rangle; \ ?\mathtt{price}:\mathtt{Double} \cdot \mathtt{sprice}(\mathtt{price}); \mathtt{buy\_lsp}. \end{aligned}$$

The protocol specification asserts that each outward transmission of a non-`null` `node` must be followed by an input of type `Double`. In other words, the party which obeys the `buy_lsp` protocol, Buyer in this case, first transmits an element whose details are stored in the `node` structure, and then expects a response from Seller in the form of a price for the considered element. The communication terminates once Buyer has received a `null` reference from Seller, which marks the end of the list (the list which stores the elements of interest). The communication termination condition is emphasized in the first disjunct of the protocol specification: where `p` must be `null`, as opposed to the case where the list still contains elements signaled by $\mathtt{p} \mapsto \mathtt{node}\langle \_, \_ \rangle$ and thus the specification continues to iterate over `buy_lsp`.

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

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

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

```
void buyer(Channel c)
     requires C(c, buy_lsp)
     ensures  C(c, emp);
{ node it = get_items();
  iterate_prices(c,it);
}

void seller(Channel c)
     requires C(c, ~buy_lsp)
     ensures  C(c, emp);
{ node it = receive(c);
  if(it != null){
    Double p = price(it->id);
    send(c,p);
    free_node(it);
    seller(c);
  }
}
```

```
void iterate_prices(Channel c, node it)
     requires C(c, buy_lsp) * ll(it)
     ensures  C(c, emp);
{ if(it != NULL) {
    struct *node nxt = it->next;
    int id = it->id;
    send(c,it);
    Double price = receive(c);
    proc_price(id,price);
    iterate_prices(c,nxt);
  } else{
    send(c,it);
  }
}
```

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

```
// C(c, buy_lst)
node it = get_items();
// C(c, buy_lst) * ll(it)
iterate_prices(c,it);
// C(c, emp)
```

The list used to store the products is transmitted to the Seller on a node by node basis in the `iterate_prices` method. The fact that the communication uses node transmissions serves a double scope: for sharing product information in a compound manner and for ensuring that the individual loops of Buyer and Seller are synchronized. As opposed to other session types enforcement techniques, the synchronization of the loops can be done via the transmitted data, without the need of a different synchronization package. Generally, the session type techniques require a flag/label transmission at each iteration in order to ensure that the loops iterate consistently. In contrast, the current work allows the verification of a more optimal implementation.

More precisely, in this work the communication entities rely on the transmitted data for synchronization, offering therefore support for a more expressive and precise specification courtesy to separation logic.

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

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

```
// C(c, buy_lsp) ∗ ll(it)
5 if(it != NULL) {
// C(c, buy_lsp) ∗ ll(it) ∧ it ≠ null
6   struct *node nxt = it->next;
// ∃p0, q · C(c, buy_lsp) ∗ it↦node⟨p0, q⟩ ∗ ll(q) ∧ it ≠ null ∧ nxt = q
7   int id = it->id;
// ∃p0, q · C(c, buy_lsp) ∗ it↦node⟨p0, q⟩ ∗ ll(q) ∧ it ≠ null ∧ nxt = q ∧ id = p0
// ------------------------------------------------------------------------------
// ∃p0, q · C(c, !p:node · p = null∨!p:node · p↦node⟨_, _⟩; ?pr:Double · sprice(pr); buy_lsp.)
// ∗ it↦node⟨p0, q⟩ ∗ ll(q) ∧ it ≠ null ∧ nxt = q ∧ id = p0
8   send(c,it);
// ∃p0, q · C(c, ?pr:Double · sprice(pr); buy_lsp.) ∗ ll(q) ∧ it ≠ null ∧ nxt = q ∧ id = p0
9   Double price = receive(c);
// ∃p0, q · C(c, buy_lsp) ∗ ll(q) ∗ sprice(price) ∧ it ≠ null ∧ nxt = q ∧ id = p0
10  proc_price(id,price);
// ∃p0, q · C(c, buy_lsp) ∗ ll(q) ∗ sprice(price) ∧ it ≠ null ∧ nxt = q ∧ id = p0
11  iterate_price(c,nxt);
// ∃p0, q · C(c, emp) ∗ sprice(price) ∧ it ≠ null ∧ nxt = q ∧ id = p0
12 } else{
// C(c, buy_lsp) ∧ it = null
// ------------------------------------------------------------------------------
// C(c, !p:node · p = null∨!p:node · p↦node⟨_, _⟩; ?pr:Double · sprice(pr); buy_lsp.) ∧ it = null
13 send(c,it);
// C(c, emp) ∧ it = null
  }
// ∃p0, q · C(c, emp) ∗ sprice(price) ∧ it ≠ null ∧ nxt = q ∧ id = p0 ∨ C(c, emp) ∧ it = null
//------------------------------------------------------------------------------
// (C(c, emp) ∗ sprice(price) ∧ it ≠ null) ∨ (C(c, emp) ∧ it = null)
//------------------------------------------------------------------------------
// C(c, emp)
```

The verification of the `iterate_prices` method follows closely the same rationale as the examples detailed before. Special attention, however, is given to the verification of the `send` call at line 8, which emphasizes the copyless character of the underlying communication. More precisely, `send` consumes the `node` pointed by `it` from Buyer's abstract state (while its receiving counterpart at Seller's side inherits the permission to access the transmitted `node`). Moreover, the program state of Buyer correctly reflects the progression of the communication protocol by consuming the output !p :node from the session specification. It is important to note that our

entailment technique safely prunes the contradicting disjuncts while unfolding the specialized session predicate. The contradiction usually refers to the conflict between the conditions of the session choice and the program's state guard. Specifically, our calculus prunes the session disjunct which refers to the `null` iterator, provided that the calling context is guarded by the recursion condition, namely $\texttt{it} \neq \texttt{null}$.

3.6 Deadlock Detection

This section discusses how deadlocks can be detected during two-party asynchronous communication within the current framework. The proposed deadlock detection mechanisms is based on the assumption that `send` method is non-blocking, while `receive` may block.

Consider a single channel communication between two parties. Let us analyse the possible situations that may lead to a deadlock. After opening, each channel has two aliases: one with the given specification $\overline{\mathtt{L}}$ and the other one with its dual specification $\sim\overline{\mathtt{L}}$. The two aliases of a single channel are used by two parties sharing the considered channel. In the main procedure, these aliases are passed to the respective parties (threads) involved in the communication. Intra-channel deadlock may occur when the two parties of the channel are not synchronized with an excess of blocking `receive` methods, before `send` methods. That means both parties have different consumption of the resource $\mathcal{C}(\mathtt{c}, \overline{\mathtt{L}})$ and $\mathcal{C}(\mathtt{c}, \sim\overline{\mathtt{L}})$. The dealock is detected by performing a synchronization check at the point where the processes of the parties are merged (or joined) together. This check consists of a compatibility investigation of the remaining resources from the same channel, say $\mathcal{C}(\mathtt{c}, \overline{\mathtt{L}_1})$ and $\mathcal{C}(\mathtt{c}, \overline{\mathtt{L}_2})$. The rules are given in Fig. 3.9. `compatible` is recursively defined to check the compatibility of specifications $\overline{\mathtt{L}_1}$ and $\overline{\mathtt{L}_2}$. `comp` ensures that the two specifications are dual to each other modulo an unbounded number of leading reads which one of the specifications might contain. The duality of the specifications is checked via `compE`, which also tackles a more salient duality point w.r.t. disjunction compatibility, namely that the number of sending disjuncts must not exceed the possibilities of the choices expected by the receiving party. This last check is needed to ensure that the sender is subsumed by the options that the receiver is designed to handle.

Let us consider a simple example where $\overline{\mathtt{L}} \triangleq ?\mathtt{String}$. The example shows the situation when at the merging point one party has the session specification $\mathcal{C}(\mathtt{c}, \mathtt{emp})$ and the other has the specification $\mathcal{C}(\mathtt{c}, !\mathtt{String})$. These two channel specifications are incompatible, with an

$$\frac{\mathtt{n} \leq \mathtt{m} - \mathtt{n} \quad \forall \mathtt{i} \in \{1,..,\mathtt{n}\}, \exists \mathtt{j} \in \{(\mathtt{n}{+}1),..,\mathtt{m}\} \cdot \mathtt{compatible}(!\mathtt{r_i} \cdot \Delta_\mathtt{i}; \overline{\mathtt{L}}_\mathtt{i}, ?\mathtt{r_j} \cdot \Delta_\mathtt{j}; \overline{\mathtt{L}}_\mathtt{j})}{\mathtt{comp\_aux}(\bigvee_{\mathtt{i}=1..\mathtt{n}} !\mathtt{r_i} \cdot \Delta_\mathtt{i}; \mathtt{L_i}, \bigvee_{\mathtt{j}=(\mathtt{n}+1)..\mathtt{m}} ?\mathtt{r_j} \cdot \Delta_\mathtt{j}; \mathtt{L_j})}$$

$$\frac{\Delta_1 \vdash [\mathtt{r}_1/\mathtt{r}_2]\Delta_2}{\mathtt{comp\_aux}(!\mathtt{r}_1 \cdot \Delta_1, ?\mathtt{r}_2 \cdot \Delta_2)} \qquad \frac{\mathtt{comp\_aux}(\overline{\mathtt{L}}_2, \overline{\mathtt{L}}_1)}{\mathtt{comp\_aux}(\overline{\mathtt{L}}_1, \overline{\mathtt{L}}_2)}$$

$$\frac{\mathtt{compE}(\overline{\mathtt{L}}_1, \overline{\mathtt{L}}_2) \vee \mathtt{comp}(\overline{\mathtt{L}}_1, \overline{\mathtt{L}}_2)}{\mathtt{comp}(?\mathtt{r} \cdot \Delta; \overline{\mathtt{L}}_1, \overline{\mathtt{L}}_2)} \qquad \frac{\mathtt{comp\_aux}(\mathtt{h}_1, \mathtt{h}_2) \quad \mathtt{compE}(\overline{\mathtt{L}}_1, \overline{\mathtt{L}}_2)}{\mathtt{compE}(\mathtt{h}_1; \overline{\mathtt{L}}_1, \mathtt{h}_2; \overline{\mathtt{L}}_2)}$$

$$\frac{\mathtt{comp}(\overline{\mathtt{L}}_1, \overline{\mathtt{L}}_2) \vee \mathtt{comp}(\overline{\mathtt{L}}_2, \overline{\mathtt{L}}_1) \vee \mathtt{compE}(\overline{\mathtt{L}}_1, \overline{\mathtt{L}}_2)}{\mathtt{compatible}(\overline{\mathtt{L}}_1, \overline{\mathtt{L}}_2)}$$

Figure 3.9: Compatibility Check for Channel Specifications.

outstanding `send`, leading to an intra-channel deadlock due to the blocking `receive(c)` method.

```
1  open(c) with ?String;
2  // C(c, ?String) * C(c, !String)
3  // C(c, ?String)              ||  // C(c, !String)
4  String s = receive(c);        ||  skip;
5  // C(c, emp)                  ||  // C(c, !String)
6  // C(c, emp) * C(c, !String)
7  // Synchronization check FAILS since ¬(compatible(emp, !String))
8  send(c,"xyz");
```

Let us now swap the positions of the `receive(c)` and `send(c, "xyz")` methods, such that the non-blocking `send` method is executed first. In this case, the specifications of the communication channel, $\mathcal{C}(\mathtt{c}, ?\mathtt{String})$ and $\mathcal{C}(\mathtt{c}, \mathtt{emp})$ respectively, remain compatible since it contains an excess of the `receive` operation that could always be invoked at a later time. The compatibility check succeeds on this merge point. Therefore the call to `receive` is executed after the merging point, hence no intra-channel deadlock in this scenario.

```
1  open(c) with ?String;
2  // C(c, ?String) ∗ C(c, !String)
3  // C(c, !String)          ||  // C(c, ?String)
4  send(c, "xyz");           ||  skip;
5  // C(c, emp)              ||  // C(c, ?String)
6  // C(c, emp) ∗ C(c, ?String)
7  // Synchronization OK since compatible(emp, ?String) holds
8  String s = receive(c);
9  // C(c, emp) ∗ C(c, emp)
```

A more general scenario is outlined next for some arbitrary $\mathtt{P}(\mathtt{c})$ and $\mathtt{Q}(\mathtt{c})$ method. Applying the compatibility checking rules at the merge point to check if intra-channel deadlock occurred sufficies for the two-party communication scenario sharing a single channel.

```
1  open(c) with L̄;
2  // C(c, L̄) ∗ C(c, ∼ L̄)
3  // C(c, L̄)      ||  // C(c, ∼ L̄)
4  P(c);            ||  Q(c);
5  // C(c, L̄1)     ||  // C(c, L̄2)
6  // C(c, L̄1) ∗ C(c, L̄2)
7  // Synchronization check compatible(L̄1, L̄2)?
```

Inter-Channel Deadlock Detection. Thus far, this chapter discussed deadlock detection in the case of two-party communication sharing a single channel. However, should multiple channels be used in the communication, the intra-channel deadlock detection certainly misses the cross-channel deadlocks, as highlighted by the following toy-example where two channels are specified by the same protocol:

$$\overline{\mathtt{L}} \triangleq \mathtt{!String}$$

Consider two methods, as follows:

```
1   void P1(Chan c1, Chan c2)
2    requires C(c1, ~L̄) * C(c2, L̄)
3    ensures  C(c1, emp) * C(c2, emp)
4   {
5    String s = receive(c1);
6    send(c2,s);
7   }
```

```
1   void Q1(Chan c1, Chan c2)
2    requires C(c1, L̄) * C(c2, ~L̄)
3    ensures  C(c1, emp) * C(c2, emp)
4   {
5    String s = receive(c2);
6    send(c1,s);
7   }
```

The main program would involve:

```
1   open(c1) with L̄;
2   // C(c1, L̄) * C(c1, ~ L̄)
3   open(c2) with L̄;
4   // C(c1, L̄) * C(c1, ~ L̄)*C(c2, L̄) * C(c2, ~ L̄)
5   // C(c1, ~ L̄)*C(c2, L̄)        ||   // C(c1, L̄)*C(c2, ~ L̄)
6   P1(c1,c2);                     ||   Q1(c1,c2);
7   // C(c1, emp) * C(c2, emp)     ||   // C(c1, emp) * C(c2, emp)
6   // C(c1, emp) * C(c2, emp) * C(c1, emp) * C(c2, emp)
```

Even though the two threads are compatible at the join point, the configuration leads to deadlock due to the fact that both threads enter a block state (as per line 5 in the implementation of $\mathtt{P}_1$ and $\mathtt{Q}_1$, respectively). In order to detect such a deadlock, this thesis proposes a variation of the wait-for graphs where nodes represent channels instead of threads. This is a novel approach and a contribution of this thesis for the static deadlock detection in the context of communicating programs.

To support wait-for graph, the accumulation predicate $\mathtt{WAIT}\{\mathtt{R},\mathtt{W}\}$ captures $\mathtt{W}$ as the set of accumulated wait-graph and $\mathtt{R}$ as the set of prior blocking receive channel. To support reasoning about such graphs, the communication primitives should also be instrumented to manipulate the accumulation predicate. A proposal of such instrumentation in depicted in Fig. 3.10.

Using the new rules for the communication primitives, the abstract execution of $\mathtt{P}_1$ leads to $\mathtt{WAIT}(\{\mathtt{c1}\},\{\mathtt{c1} \mapsto \mathtt{c2}\})$ (where $\mathtt{c1} \mapsto \mathtt{c2}$ is read as “c2 waits for c1"), and that of $\mathtt{Q}_1$ leads to $\mathtt{WAIT}(\{\mathtt{c2}\},\{\mathtt{c2} \mapsto \mathtt{c1}\})$. The join of the two threads allows the combination of the accumulations leading therefore to $\mathtt{WAIT}(\{\mathtt{c1},\mathtt{c2}\},\{\mathtt{c1} \mapsto \mathtt{c2}, \mathtt{c2} \mapsto \mathtt{c1}\})$ which contains a cycle, hence a deadlock.

Consider the following implementation of the same protocol that does not lead to deadlock.

$$\frac{}{\vdash \{\mathtt{emp}\}\ \mathtt{open()}\ \mathtt{with}\ \overline{\mathtt{L}}\ \{\mathcal{C}(\mathtt{res}, \overline{\mathtt{L}}) * \mathcal{C}(\mathtt{res}, \sim\overline{\mathtt{L}}) * \mathtt{WAIT}\{\{\}, \{\}\}\}}\ [\mathbf{dOPEN}]$$

$$\frac{}{\vdash \{\mathcal{C}(\mathtt{c}, \mathtt{emp}) * \mathcal{C}(\mathtt{c}, \mathtt{emp}) * \mathtt{WAIT}\{\mathtt{R}, \mathtt{W}\}\}\ \mathtt{close(c)}\ \{\mathtt{emp}\}}\ [\mathbf{dCLOSE}]$$

$$\frac{\mathtt{WAIT}_{\mathtt{pre}} = \mathtt{WAIT}\{\mathtt{R}, \mathtt{W}\} \quad \mathtt{WAIT}_{\mathtt{post}} = \mathtt{WAIT}\{\mathtt{R}, \mathtt{W} + \{\mathtt{c}_1 \mapsto \mathtt{c} | \mathtt{c}_1 \in \mathtt{R}, \mathtt{c}_1 \neq \mathtt{c}\}\}}{\vdash \{\mathcal{C}(\mathtt{c}, !\mathtt{r} \cdot \mathtt{V}(\mathtt{r}); \mathtt{V}^{\mathtt{rest}}) * \mathtt{V}(\mathtt{x}) * \mathtt{WAIT}_{\mathtt{pre}}\}\ \mathtt{send(c, x)}\ \{\mathcal{C}(\mathtt{c}, \mathtt{V}^{\mathtt{rest}}) * \mathtt{WAIT}_{\mathtt{post}}\}}\ [\mathbf{dSEND}]$$

$$\frac{\mathtt{WAIT}_{\mathtt{pre}} = \mathtt{WAIT}\{\mathtt{R}, \mathtt{W}\} \quad \mathtt{WAIT}_{\mathtt{post}} = \mathtt{WAIT}\{\mathtt{R} + \{\mathtt{c}\}, \mathtt{W}\}}{\vdash \{\mathcal{C}(\mathtt{c}, ?\mathtt{r} \cdot \mathtt{V}(\mathtt{r}); \mathtt{V}^{\mathtt{rest}} * \mathtt{WAIT}_{\mathtt{pre}})\}\ \mathtt{receive(c)}\ \{\mathcal{C}(\mathtt{c}, \mathtt{V}^{\mathtt{rest}}) * \mathtt{V}(\mathtt{res}) * \mathtt{WAIT}_{\mathtt{post}}\}}\ [\mathbf{dRECV}]$$

Figure 3.10: Communication Primitives with Support for Deadlock Detection

Collecting the accumulation we can prove that it remains acyclic and thus deadlock-free.

```
1   void P2(Chan c1, Chan c2)
2   requires C(c1, ~L̄) * C(c2, L̄)
3   ensures  C(c1, emp) * C(c2, emp)
4   {
5     String s = "hello";
6     send(c2,s);
7     s = receive(c1);
8   }
```

Replacing $\mathtt{P}_1$ with $\mathtt{P}_2$ in the main code leads to $\mathtt{WAIT}(\{\mathtt{c1}\}, \{\})$ as the effect of abstractly execution $\mathtt{P}_2$, while the effect of executing $\mathtt{Q}_1$ remains unchanged: $\mathtt{WAIT}(\{\mathtt{c2}\}, \{\mathtt{c2} \mapsto \mathtt{c1}\})$. It is easy to notice that merging the post-state of $\mathtt{P}_2$ and that of $\mathtt{Q}_1$ at the join point, leaves the execution in deadlock-free state.

However, this section is only meant to informally discuss selected deadlock scenarios and their detection in order to raise awareness over the ease of triggering a deadlock and the challenge of detecting it. A slight change in the previously discussed example, and the proposed method might lead to false positives. For example, if the implementation of $\mathtt{P}_2$ and $\mathtt{Q}_1$ would be changed to contain a final transmission on channel c2, from $\mathtt{P}_2$ to $\mathtt{Q}_1$, the proposed wait-for graph would falsely detect a deadlock. The generalization, correction and formalism of the discussed mechanisms are further developed in Sec. 5.3 in the more realistic scenario of multiparty

communication. The proposal of Sec. 5.3 for deadlock detection avoids false alarms by identifying each communication event as a unique event, as opposed to the above discussion where two sends over the same channel and initiated by the same party are indistinguishable w.r.t. the channel-directed wait-for graph.

Chapter 4

A SESSION LOGIC FOR MULTIPARTY COMMUNICATION PROTOCOLS

Research progress towards specification-based development for the message passing paradigm has mostly revolved around the session-based concurrency. In these approaches, the communication patterns are abstracted as session types for variants of π-calculus [47, 26] which are either statically checked against given specifications or are being used to guide the generation of deadlock-free code [17].

While providing a foundational approach towards communication correctness, most of these works make the assumption that the underlying system communicates exclusively via message passing using unlimited channels. That is, extra channels are created when necessary to resolve any ambiguity in the communication protocol. This assumption is broken in current systems which may rely on more tightly synchronized concurrency. Concurrency is often managed by using a combination of inter-process communication mechanisms designed to work together for better performance over shared channels. Therefore, software developers should also not be constrained into a single model of communication or computation. For instance, Android systems recognize the benefits of using message passing, but are not hesitant in combining this style with other synchronization mechanisms such as `CountDownLatch`. Similarly, UNIX mingles a number of mechanisms to obtain synchronization, each with its own benefits and tradeoffs. Moreover, while sequential and disjunctive combinations of communication channels

are quite well understood, the same cannot be said for concurrent compositions of transmissions over shared channels.

In summary, most of the current approaches which guarantee communication safety, conveniently limit their results to the implicit synchronization of protocols - the synchronization which relies solely on message passing. Should race-related conflict arise, extra transmission channels are added to ensure unambiguous communication. This thesis considers how to abstractly extend communication protocols to those which require *explicit synchronization*. One of the guiding principles of this thesis is to keep the specification in concurrency abstract, but with sufficient detail so that the implementation can be safely written to meet the intention of protocol designers. With this, the aim is to design an expressive logic for session-based concurrency, and to provide a set of tools for automated reasoning using the enhanced session logic. This thesis considers how to ensure well-formedness of the session logic, and how to build summaries that can help ensure race-freedom and type safety amongst shared channels.

As an example, session types approaches only impose sequencing on the actions of the same participant, as illustrated below:

$$(\mathtt{A}\rightarrow\mathtt{C})\,;\,(\mathtt{B}\rightarrow\mathtt{C})$$

This example involves three communicating parties with C receiving data from A and B, respectively. We assume the asynchronous communication model described in Sec. 2.6 where multiple parties may share the same channel. In terms of communications, the two sends (by A and B) are un-ordered, while the two receives by C are strictly sequentialized (by C itself) with the receipt of message from A expected to occur before the message from B. Such implicit synchronization may only work if two distinct channels are being used to serve the transmissions of $\mathtt{A}\rightarrow\mathtt{C}$ and that of $\mathtt{B}\rightarrow\mathtt{C}$, respectively, since the two sends by A and B would then arrive un-ambiguously, irrespective of their arrival order over the two distinct channels, say $\mathtt{c}_1$ and $\mathtt{c}_2$, as shown below.

$$(\mathtt{A}\rightarrow\mathtt{C}:\mathtt{c}_1)\,;\,(\mathtt{B}\rightarrow\mathtt{C}:\mathtt{c}_2)$$

However, if we had to use a common channel, say c for the mailbox of party C, we would have the following specification:

$$(\mathtt{A}\xrightarrow{i_1}\mathtt{C}:\mathtt{c}\langle\mathtt{t}_1\rangle)\,;\,(\mathtt{B}\xrightarrow{i_2}\mathtt{C}:\mathtt{c}\langle\mathtt{t}_2\rangle)$$

which is a legitimate communicating scenario.

We also provide unique labels i_1 and i_2 for each transmission, and its corresponding message types $\mathtt{t}_1$ and $\mathtt{t}_2$. We assume each party never sends message to itself. This allows us to use $\mathtt{P}^{(i)}$

to unambiguously refer to a send or receive event by party P at a transmission labelled *i*. With this scenario, we now have a situation where two senders (by A and B) must be explicitly ordered to have their messages arrive in strict sequential order. The two receives by the same party C are already implicitly ordered. It may even become a communication safety issue if $\mathtt{t}_1 \neq \mathtt{t}_2$ and should the two messages arrive in the wrong order. We refer to this problem as a *channel race* where messages could reach unintended destinations. In this case, we would have to arrange for the two sends (initiated by A and B) to be strictly ordered. For this to be supported, we propose an *explicit synchronization* mechanism that would force the second send by B to occur after the first send by A has been initiated. While explicit synchronization can be handled by a number of mechanisms, such as `notifyAll` − `wait`, `barriers` or `CountDownLatch`, the current approach abstractly captures this requirement by an *ordering*, namely $\mathtt{A}^{(i_1)} \prec_{\mathtt{HB}} \mathtt{B}^{(i_2)}$, which explicitly requires event $\mathtt{A}^{(i_1)}$ to happen before event $\mathtt{B}^{(i_2)}$.

Such explicit orderings should be minimised, where possible, but they are an essential component of the concurrency control for the message passing paradigm. Our contributions are:

- We design an *expressive session logic* that is both precise (in that it uses the more expressive logical formulae to describe the transmitted message as opposed to the more general types) and concise (it relies on less communication constructs than session types using disjunction to describe both internal and external choice) for modelling multi-party protocols.

- We ensure *race-freedom* in communications over common channels with both *implicit* and *explicit* synchronization. Prior works mostly relied on implicit synchronization, reducing thus the number of useful communicating scenarios or overusing communication channels to avoid race.

- We provide a *specification refinement* that explicitly introduces both assumptions and proof obligations. These may be either local or global events and proofs, but can be used together to ensure adherence to the global protocol, communication safety and race-freedom between common channels.

- Each refined specification may be firstly projected for each party and then for each channel, with a set of shared global assumptions. There are two novelties here. Firstly, global proof obligations may be projected to support *cooperative proving*, where the proof by

one party can be relied on as assumption by the other concurrent parties. Secondly, we use *event guards* to ensure that channel specifications are verified in correct sequence.

- We limit our protocol to well-formed disjunction where a common sender and receiver is always and exclusively present in the disjunction to express communications for all branches. This restriction provides a *simplification* where every $\prec_{\mathtt{HB}}$ ordering discovered is a *must*-ordering, since transitivity holds globally whenever it holds in one of the branches.

- We show how *automated verification* is applied on a per party basis with global assumptions and proof obligations. Assumptions are released as soon as possible. Proof obligations can be delayed and locally proven at the appropriate time.

4.1 Informal Development

Fig. 4.1 depicts the terms and composition operators of the proposed logic for the specification of global protocols. The protocols written with this logic use $\mathtt{G}_1 * \mathtt{G}_2$ to denote the concurrent composition of $\mathtt{G}_1$ and $\mathtt{G}_2$, and $\mathtt{G}_1 \vee \mathtt{G}_2$ for disjunctive choice between either $\mathtt{G}_1$ or $\mathtt{G}_2$, and finally $\mathtt{G}_1 \,;\, \mathtt{G}_2$ for the implicit sequentialization of $\mathtt{G}_1$ before $\mathtt{G}_2$ w.r.t. the same party and/or the same channel communication. Moreover, we express the message type of a transmission $\mathtt{S} \xrightarrow{i} \mathtt{R}$: $\mathtt{c}\langle \mathtt{v} \cdot \Delta \rangle$ by $\mathtt{v} \cdot \Delta$ which expects a message $\mathtt{v}$ on some resource expressed in logical form Δ (whose definition is given in Fig. 2.2) transmitted over the logical channel $\mathtt{c}$. As an example, $\mathtt{v} \cdot \mathtt{ll}(\mathtt{v}, \mathtt{n})_{\frac{1}{2}}$ would capture the transmission of half fractional permission [1] of a linked list with $\mathtt{n}$ elements $\mathtt{ll}(\mathtt{v}, \mathtt{n})$ rooted at $\mathtt{v}$. Apart from messages in logical form, our session logic captures logical disjunction and can enforce explicit synchronization to ensure communication safety.

Apart from unique transmission labels, we provide a specification refinement that adds guards (of the form $\ominus(\Psi)$) and assumptions (of the form $\oplus(\Psi)$), as more precise refinement of its global protocol. The former is used to add assertions (or proof obligations), such as explicit synchronization, into its global protocol to ensure safe communications; while the latter is used to support flow of information across multiple parties to facilitate local verification. To sketch the main purpose of using these guards and assumptions, we briefly introduce elements of its information flow language later in this section. Further details about Ψ are given in Sec.

[1] Fractional permissions are important for supporting concurrent programming. Its use allows read access when a resource is shared by multiple parties, but also write access when a resource later becomes exclusively owned.

Global protocol	$\mathtt{G} ::=$
Single transmission	$\mathtt{S} \xrightarrow{i} \mathtt{R} : \mathtt{c}\langle \mathtt{v} \cdot \Delta \rangle$
Concurrency	$\mid \mathtt{G} * \mathtt{G}$
Choice	$\mid \mathtt{G} \vee \mathtt{G}$
Sequencing	$\mid \mathtt{G} ; \mathtt{G}$
Guard	$\mid \ominus(\Psi)$
Assumption	$\mid \oplus(\Psi)$
Inaction	$\mid \mathtt{emp}$

(Parties) $\mathtt{P}, \mathtt{S}, \mathtt{R} \in \mathcal{R}\mathtt{ole}$ *(Channels)* $\mathtt{c} \in \mathcal{L}\mathtt{chan}$ *(Messages)* $\mathtt{v} \cdot \Delta$ *(Labels)* $i \in \mathtt{Nat}$

Figure 4.1: A Logic for Global Protocol Specification

4.2.3.

These new specification constructs are added automatically by our refinement procedure to provide assumptions and proof obligations which ensure race-freedom in shared channels. Our refinement could add channel information based on some expected communication patterns. For example, if mailbox communication style is preferred, each communication always directs its message to the mailbox of its receiver. If bi-directional channels of communication are preferred, a distinct channel is provided for every pair of parties that are in communication. To support diversity, we assume channels are captured explicitly in our protocol specifications.

Our approach thus takes in a protocol specified in an expressive logic, *refine* it, then *project* it on a per party and per channel basis. Once these are done, we can proceed to apply automated verification on the entire program code using the derived specs.

The automated verification is conducted against a general and expressive specification language based on Separation Logic [86, 20], as shown in Fig. 2.2, which supports the use of (inductive) predicates, the specification of separation and numerical properties, as well as channel predicates.

4.1.1 Specification Refinement

Let us use a well-known example to illustrate the key ideas of our approach. Consider the multi-party protocol from [47] where we use a distinct mailboxes (here on channels) for each three parties, namely seller $\mathtt{S}$, first buyer $\mathtt{B}_1$ and the second collaborative buyer $\mathtt{B}_2$.

$$\mathtt{B}_1 \rightarrow \mathtt{S} : \mathtt{s}\langle \mathtt{Order} \rangle \ ;$$
$$(\mathtt{S} \rightarrow \mathtt{B}_1 : \mathtt{b}_1\langle \mathtt{Price} \rangle * \mathtt{S} \rightarrow \mathtt{B}_2 : \mathtt{b}_2\langle \mathtt{Price} \rangle) \ ;$$
$$\mathtt{B}_1 \rightarrow \mathtt{B}_2 : \mathtt{b}_2\langle \mathtt{Amt} \rangle \ ;$$
$$(\mathtt{B}_2 \rightarrow \mathtt{S} : \mathtt{s}\langle \mathtt{No} \rangle \vee (\mathtt{B}_2 \rightarrow \mathtt{S} : \mathtt{s}\langle \mathtt{Yes} \rangle \ ; \ \mathtt{B}_2 \rightarrow \mathtt{S} : \mathtt{s}\langle \mathtt{Addr} \rangle))$$

Note that $\mathtt{S} \rightarrow \mathtt{R} : \mathtt{c}\langle \mathtt{T} \rangle$ is a short-hand for $\mathtt{S} \rightarrow \mathtt{R} : \mathtt{c}\langle \mathtt{v} \cdot \mathtt{T} \rangle$ where $\mathtt{T}$ is a type or a predicate for some resource, such as `Order`, `Price`, `Amt` or `Addr`.

For the above example, our refinement would add information to signify the timely completion of each communication using $\oplus(\mathtt{S} \xrightarrow{i} \mathtt{R})$. This event itself captures two locally observable events and a globally observable ordering, as follows:

$$\oplus(\mathtt{S} \xrightarrow{i} \mathtt{R}) \Rightarrow \oplus(\mathtt{S}^{(i)}) \wedge \ \oplus(\mathtt{R}^{(i)}) \wedge \oplus(\mathtt{S}^{(i)} \prec_{\mathtt{CB}} \mathtt{R}^{(i)})$$

which indicates the occurrence of the send and receive events and the expected communicates-before ordering[2], denoted by $\prec_{\mathtt{CB}}$, for its successfully completed communication. The send and receive events are only observable locally. For example, the completion of event $\oplus(\mathtt{P}_1^{(i)})$ is only observable in party $\mathtt{P}_1$, but not by other distinct parties, such as $\mathtt{P}_2$. Its occurrence in the global protocol is thus location sensitive and can be used to capture relative orderings within the same party. However, the ordering $\oplus(\mathtt{S}^{(i)} \prec_{\mathtt{CB}} \mathtt{R}^{(i)})$ is globally observable since it relates two parties and our assurance that all send/receive events are uniquely labelled. Later, we will see how immediately adjacent $\prec_{\mathtt{HB}}$ ordering information is generated from locally observable events within each party, and how $\prec_{\mathtt{HB}}$ orderings can be transitively propagated from other globally observable $\prec_{\mathtt{CB}}$ and $\prec_{\mathtt{HB}}$ orderings.

When common channels are involved, we also use proof obligations of the form $\ominus(i_1 \prec_{\mathtt{HB}} i_2)$ to denote two transmissions, labelled i_1 and i_2, that are expected to be sequentialized. Assuming each transmission i be denoted by $\mathtt{S}_\mathtt{i} \xrightarrow{i} \mathtt{R}_\mathtt{i} \ : \ \mathtt{c}\langle \mathtt{v} \cdot \Delta_\mathtt{i} \rangle$ with $\mathtt{c}$ as common channel, these proof obligations can be further reduced, as follows:

$$\ominus(i_1 \prec_{\mathtt{HB}} i_2) \equiv \ominus(\mathtt{S}_1^{(i_1)} \prec_{\mathtt{HB}} \mathtt{S}_2^{(i_2)} \wedge \mathtt{R}_1^{(i_1)} \prec_{\mathtt{HB}} \mathtt{R}_2^{(i_2)})$$

[2]Under race-free assumption, $\prec_{\mathtt{CB}}$ ordering is the same as $\prec_{\mathtt{HB}}$-ordering.

The $\prec_{\mathtt{HB}}$ is overloaded for both transmissions e.g. $i_1 \prec_{\mathtt{HB}} i_2$ and events e.g. $\mathtt{P}_1^{(i_1)} \prec_{\mathtt{HB}} \mathtt{P}_2^{(i_2)}$. In the definition of $\ominus(i_1 \prec_{\mathtt{HB}} i_2)$, we used the weakest condition to ensure race-freedom over the common channels for transmission i_1 and i_2. We ensure that the two senders are ordered by $\ominus(\mathtt{S}_1^{(i_1)} \prec_{\mathtt{HB}} \mathtt{S}_2^{(i_2)})$, and the two receivers are ordered by $\ominus(\mathtt{R}_1^{(i_1)} \prec_{\mathtt{HB}} \mathtt{R}_2^{(i_2)})$. Our refined specification follows:

$$
\begin{aligned}
&\mathtt{G} \triangleq \mathtt{B}_1 \xrightarrow{1} \mathtt{S} : \mathtt{s}\langle \mathtt{Order} \rangle \ ; \ \oplus(\mathtt{B}_1 \xrightarrow{1} \mathtt{S}) \ ; \\
&\quad ((\mathtt{S} \xrightarrow{2} \mathtt{B}_1 : \mathtt{b}_1\langle \mathtt{Price} \rangle \ ; \ \oplus(\mathtt{S} \xrightarrow{2} \mathtt{B}_1); \oplus(\mathtt{S}^{(1)} \prec_{\mathtt{HB}} \mathtt{S}^{(2)}); \oplus(\mathtt{B}_1^{(1)} \prec_{\mathtt{HB}} \mathtt{B}_1^{(2)})) \\
&\qquad * \ (\mathtt{S} \xrightarrow{3} \mathtt{B}_2 : \mathtt{b}_2\langle \mathtt{Price} \rangle \ ; \ \oplus(\mathtt{S} \xrightarrow{3} \mathtt{B}_2); \oplus(\mathtt{S}^{(1)} \prec_{\mathtt{HB}} \mathtt{S}^{(3)}))) \ ; \\
&\quad \mathtt{B}_1 \xrightarrow{4} \mathtt{B}_2 : \mathtt{b}_2\langle \mathtt{Amt} \rangle \ ; \ \oplus(\mathtt{B}_1 \xrightarrow{4} \mathtt{B}_2); \oplus(\mathtt{B}_1^{(2)} \prec_{\mathtt{HB}} \mathtt{B}_1^{(4)}); \oplus(\mathtt{B}_2^{(3)} \prec_{\mathtt{HB}} \mathtt{B}_2^{(4)}) \ ; \ \ominus(3 \prec_{\mathtt{HB}} 4) \ ; \\
&\quad ((\mathtt{B}_2 \xrightarrow{5} \mathtt{S} : \mathtt{s}\langle \mathtt{No} \rangle \ ; \ \oplus(\mathtt{B}_2 \xrightarrow{5} \mathtt{S}); \oplus(\mathtt{B}_2^{(4)} \prec_{\mathtt{HB}} \mathtt{B}_2^{(5)}); \oplus(\mathtt{S}^{(2)} \prec_{\mathtt{HB}} \mathtt{S}^{(5)}); \oplus(\mathtt{S}^{(3)} \prec_{\mathtt{HB}} \mathtt{S}^{(5)}); \ominus(1 \prec_{\mathtt{HB}} 5)) \\
&\quad \vee (\mathtt{B}_2 \xrightarrow{6} \mathtt{S} : \mathtt{s}\langle \mathtt{Yes} \rangle; \oplus(\mathtt{B}_2 \xrightarrow{6} \mathtt{S}); \oplus(\mathtt{B}_2^{(4)} \prec_{\mathtt{HB}} \mathtt{B}_2^{(6)}); \oplus(\mathtt{S}^{(2)} \prec_{\mathtt{HB}} \mathtt{S}^{(6)}); \oplus(\mathtt{S}^{(3)} \prec_{\mathtt{HB}} \mathtt{S}^{(6)}); \ominus(1 \prec_{\mathtt{HB}} 6); \\
&\qquad \mathtt{B}_2 \xrightarrow{7} \mathtt{S} : \mathtt{s}\langle \mathtt{Addr} \rangle \ ; \ \oplus(\mathtt{B}_2 \xrightarrow{7} \mathtt{S}); \oplus(\mathtt{B}_2^{(6)} \prec_{\mathtt{HB}} \mathtt{B}_2^{(7)}); \oplus(\mathtt{S}^{(6)} \prec_{\mathtt{HB}} \mathtt{S}^{(7)}) \ ; \ \ominus(6 \prec_{\mathtt{HB}} 7)))
\end{aligned}
$$

The refined specification contains information on observable local events and global communications, which can be used to handle proof obligations on race-free channels, e.g. $\ominus(3 \prec_{\mathtt{HB}} 4)$. When race-freedom cannot be guaranteed, we could also introduce explicit synchronization to ensure it. Unique labels on each transmission, and thus its events, are trivially added by our refinement process. Moreover, the assumptions attached to a certain transmission are composed using strict sequencing even though the order in which they are released in the state (w.r.t. each other) is not important. The choice of using sequence composition is in agreement with our goal of having a concise specification language, since adding a new operator would unnecessarily complicate the proposed model.

We also generate immediate $\prec_{\mathtt{HB}}$ orderings between each pair of adjacent events of the same party in the global protocol, as follows:

$$.. \oplus (\mathtt{P}^{(i_1)}) .. \ ; \ .. \oplus (\mathtt{P}^{(i_2)}) .. \Rightarrow \oplus(\mathtt{P}^{(i_1)} \prec_{\mathtt{HB}} \mathtt{P}^{(i_2)})$$

While the $\prec_{\mathtt{HB}}$ and $\prec_{\mathtt{CB}}$ are globally observable events, some of them are partial (or dynamic) in nature due to the presence of disjunction. Our present solution to disjunctive formulae is to enforce a restricted form of disjunction where only a common sender and a common receiver

are involved in each disjunctive formula. This well-formed restriction ensures that parties involved in disjunction know precisely which conditional branches are being executed. With this restriction, we can soundly use the following global propagation lemma since it will always be transitively undertaken for all branches of the disjunct.

$$\oplus(\mathtt{P}_1^{(i_1)} \prec_{\mathtt{HB}} \mathtt{P}_2^{(i_2)}) \wedge \oplus(\mathtt{P}_2^{(i_2)} \prec_{\mathtt{HB}} \mathtt{P}_3^{(i_3)}) \quad \Rightarrow \quad \oplus(\mathtt{P}_1^{(i_1)} \prec_{\mathtt{HB}} \mathtt{P}_3^{(i_3)})$$

Another kind of propagation lemmas involve the $\prec_{\mathtt{CB}}$-ordering. There are two possible lemmas:

$$\oplus(\mathtt{P}_1^{(i_1)} \prec_{\mathtt{HB}} \mathtt{S}_2^{(i_2)}) \wedge \oplus(\mathtt{S}_2^{(i_2)} \prec_{\mathtt{CB}} \mathtt{R}_3^{(i_2)}) \quad \Rightarrow \quad \oplus(\mathtt{P}_1^{(i_1)} \prec_{\mathtt{HB}} \mathtt{R}_3^{(i_2)}) \quad (1)$$

$$\oplus(\mathtt{S}_1^{(i_1)} \prec_{\mathtt{CB}} \mathtt{R}_2^{(i_1)}) \wedge \oplus(\mathtt{R}_2^{(i_1)} \prec_{\mathtt{HB}} \mathtt{P}_3^{(i_2)}) \quad \Rightarrow \quad \oplus(\mathtt{S}_1^{(i_1)} \prec_{\mathtt{HB}} \mathtt{P}_3^{(i_2)}) \quad (2)$$

Both of these global propagation lemmas, which use the $\prec_{\mathtt{CB}}$ ordering, are sound if we assume race-free communication. Under such race-free condition, the $\prec_{\mathtt{CB}}$ ordering behaves in a manner identical to $\prec_{\mathtt{HB}}$. However, we are also using these propagation rules to verify race-freedom by checking if proof obligations of the form $\ominus(\mathtt{i}_1 \prec_{\mathtt{HB}} \mathtt{i}_2)$ hold. As it turns out, only the latter lemma (2) is sound for use in ensuring race-freedom, but not (1). We discuss more around the soundness of these lemmas in the subsequent section. This observation is a novel contribution of our approach.

4.1.2 Specification Projection

Once we have a specification that has been annotated with assumptions and obligations, we can proceed to project them for use in code verification. The projection can be done in two phases. Firstly, on a per-party basis, and secondly on a per-channel basis. A per-channel specification for each party allows us to observe all the communications and proof obligation activities within each channel. We propose to undertake per-channel projections on specifications for two reasons. Firstly, it supports backward compatibility with the two-party logic presented in Chapter 4. Secondly, it is likely to be easier to add channel-specific quantitative proof obligation(s), such as boundedness of channel's buffer [25], using this approach. However, this topic is reserved for future investigation. Furthermore, we also separately project assumptions on global orderings, that are to be shared by all parties, so that they may be separately utilized by each party for their respective local verification.

Let us first discuss the specification that would be projected for each party. For the running example, we would project the following local specifications for the three parties.

$$\begin{aligned}
\mathtt{G}\#\mathtt{B}_1 \triangleq\ & \mathtt{s}!\mathtt{Order};\oplus(\mathtt{B}_1^{(1)});\mathtt{b}_1?\mathtt{Price};\oplus(\mathtt{B}_1^{(2)});\mathtt{b}_2!\mathtt{Amt};\oplus(\mathtt{B}_1^{(4)});\ominus(3{\prec_{\mathtt{HB}}}4)_{\mathtt{B}_1}\\
\mathtt{G}\#\mathtt{B}_2 \triangleq\ & \mathtt{b}_2?\mathtt{Price};\oplus(\mathtt{B}_2^{(3)});\mathtt{b}_2?\mathtt{Amt};\oplus(\mathtt{B}_2^{(4)});\ominus(3{\prec_{\mathtt{HB}}}4)_{\mathtt{B}_2};\\
& ((\mathtt{s}!\mathtt{No};\oplus(\mathtt{B}_2^{(5)});\ominus(1{\prec_{\mathtt{HB}}}5)_{\mathtt{B}_2})\vee\\
& \ (\mathtt{s}!\mathtt{Yes};\oplus(\mathtt{B}_2^{(6)});\ominus(1{\prec_{\mathtt{HB}}}6)_{\mathtt{B}_2};\mathtt{s}!\mathtt{Addr};\oplus(\mathtt{B}_2^{(7)});\ominus(6{\prec_{\mathtt{HB}}}7)_{\mathtt{B}_2}))\\
\mathtt{G}\#\mathtt{S} \triangleq\ & \mathtt{s}?\mathtt{Order};\oplus(\mathtt{S}^{(1)});(\mathtt{b}_1!\mathtt{Price};\oplus(\mathtt{S}^{(2)}))*(\mathtt{b}_2!\mathtt{Price};\oplus(\mathtt{S}^{(3)}));\ominus(3{\prec_{\mathtt{HB}}}4)_{\mathtt{S}};\\
& ((\mathtt{s}?\mathtt{No};\oplus(\mathtt{S}^{(5)});\ominus(1{\prec_{\mathtt{HB}}}5)_{\mathtt{S}})\vee\\
& \ (\mathtt{s}?\mathtt{Yes};\oplus(\mathtt{S}^{(6)});\ominus(1{\prec_{\mathtt{HB}}}6)_{\mathtt{S}};\mathtt{s}?\mathtt{Addr};\oplus(\mathtt{S}^{(7)});\ominus(6{\prec_{\mathtt{HB}}}7)_{\mathtt{S}}))
\end{aligned}$$

Each of the local specification contains send/receive proof obligations, such as $\mathtt{s}!\mathtt{Order}$ and $\mathtt{s}?\mathtt{Order}$, which capture expected channel and message type (or property). It also contains assumptions on the local events, such as $\oplus(\mathtt{B}_1^{(1)})$ and $\oplus(\mathtt{S}^{(1)})$ that are immediately released once the corresponding send/receive obligations have been locally verified. We may also have global proof obligations, such as $\ominus(3{\prec_{\mathtt{HB}}}4)$, that is meant to ensure that the two transmissions, labelled as 3 and 4, are race-free since they share a common channel. This particular global obligation needs to be collectively proven by both the sender and the receiver of transmission 4, namely $\mathtt{B}_1$ and $\mathtt{B}_2$, but could be assumed by other parties. Thus, we project the relevant portion that has to be proven and assumed by the three parties, $\mathtt{S}$, $\mathtt{B}_1$ and $\mathtt{B}_2$, as $\ominus(3{\prec_{\mathtt{HB}}}4)_{\mathtt{B}_1}$, $\ominus(3{\prec_{\mathtt{HB}}}4)_{\mathtt{B}_2}$ and $\ominus(3{\prec_{\mathtt{HB}}}4)_{\mathtt{S}}$. As $\ominus(3{\prec_{\mathtt{HB}}}4)$ involves $\ominus((\mathtt{S}^{(3)}{\prec_{\mathtt{HB}}}\mathtt{B}_1^{(4)})\wedge(\mathtt{B}_2^{(3)}{\prec_{\mathtt{HB}}}\mathtt{B}_2^{(4)}))$, these three projections are translated, as follows:

$$\begin{aligned}
\ominus(3{\prec_{\mathtt{HB}}}4)_{\mathtt{B}_1} &= \ominus(\mathtt{S}^{(3)}{\prec_{\mathtt{HB}}}\mathtt{B}_1^{(4)});\oplus(\mathtt{B}_2^{(3)}{\prec_{\mathtt{HB}}}\mathtt{B}_2^{(4)})\\
\ominus(3{\prec_{\mathtt{HB}}}4)_{\mathtt{B}_2} &= \oplus(\mathtt{S}^{(3)}{\prec_{\mathtt{HB}}}\mathtt{B}_1^{(4)});\ominus(\mathtt{B}_2^{(3)}{\prec_{\mathtt{HB}}}\mathtt{B}_2^{(4)})\\
\ominus(3{\prec_{\mathtt{HB}}}4)_{\mathtt{S}} &= \oplus(\mathtt{S}^{(3)}{\prec_{\mathtt{HB}}}\mathtt{B}_1^{(4)});\oplus(\mathtt{B}_2^{(3)}{\prec_{\mathtt{HB}}}\mathtt{B}_2^{(4)})
\end{aligned}$$

Each party must prove the obligation that is attributable to it, but may assume the obligations of the other parties. In the case of $\mathtt{S}$, no proof obligation is attributable to it; allowing $\mathtt{S}$ to benefit from this global proof obligation (done by others) as an assumption. This is the power of cooperative global proving, where relevant proof obligations are undertaken locally, but may otherwise be assumed to hold globally. Such cooperative proving allows us to prove a global obligation, such as $\ominus(3{\prec_{\mathtt{HB}}}4)$, at multiple places.

The other global orderings from each transmission, such as communicates-before ordering like $\oplus(\mathtt{B}_1^{(1)} \prec_{\mathtt{CB}} \mathtt{S}^{(1)})$, and happens-before ordering between adjacent events within the same party, like $\oplus(\mathtt{S}^{(1)} \prec_{\mathtt{HB}} \mathtt{S}^{(2)})$, are placed in a shared space that is visible to all parties. As these global ordering information are modelled for unique events and are also immutable, we propose to release all orderings in a single step at the beginning of the protocol. Our proposal to release all global ordering together as a single assumption is done for simplicity. Though future orderings are not required to prove current global obligations, they never cause any inconsistency to the current state, but merely add some new orderings that are available in advance.

$$
\begin{aligned}
\mathtt{G\#All} \triangleq\ & \oplus(\mathtt{B}_1^{(1)} \prec_{\mathtt{CB}} \mathtt{S}^{(1)});\\
& ((\oplus(\mathtt{B}_1^{(1)} \prec_{\mathtt{HB}} \mathtt{B}_1^{(2)}); \oplus(\mathtt{S}^{(1)} \prec_{\mathtt{HB}} \mathtt{S}^{(2)}); \oplus(\mathtt{S}^{(2)} \prec_{\mathtt{CB}} \mathtt{B}_1^{(2)}))*\\
& \ (\oplus(\mathtt{S}^{(1)} \prec_{\mathtt{HB}} \mathtt{S}^{(3)}); \oplus(\mathtt{S}^{(3)} \prec_{\mathtt{CB}} \mathtt{B}_2^{(3)})));\\
& \oplus(\mathtt{B}_1^{(2)} \prec_{\mathtt{HB}} \mathtt{B}_1^{(4)}); \oplus(\mathtt{B}_2^{(3)} \prec_{\mathtt{HB}} \mathtt{B}_2^{(4)}); \oplus(\mathtt{B}_1^{(4)} \prec_{\mathtt{CB}} \mathtt{B}_2^{(4)});\\
& ((\oplus(\mathtt{B}_2^{(4)} \prec_{\mathtt{HB}} \mathtt{B}_2^{(5)}); \oplus(\mathtt{S}^{(2)} \prec_{\mathtt{CB}} \mathtt{S}^{(5)}); \oplus(\mathtt{S}^{(3)} \prec_{\mathtt{CB}} \mathtt{S}^{(5)}); \oplus(\mathtt{B}_2^{(5)} \prec_{\mathtt{CB}} \mathtt{S}^{(5)}))\\
& \vee (\oplus(\mathtt{B}_2^{(4)} \prec_{\mathtt{HB}} \mathtt{B}_2^{(6)}); \oplus(\mathtt{S}^{(2)} \prec_{\mathtt{HB}} \mathtt{S}^{(6)}); \oplus(\mathtt{S}^{(3)} \prec_{\mathtt{HB}} \mathtt{S}^{(6)}); \oplus(\mathtt{B}_2^{(6)} \prec_{\mathtt{CB}} \mathtt{S}^{(6)});\\
& \ \oplus(\mathtt{B}_2^{(6)} \prec_{\mathtt{HB}} \mathtt{B}_2^{(7)}); \oplus(\mathtt{S}^{(6)} \prec_{\mathtt{HB}} \mathtt{S}^{(7)}); \oplus(\mathtt{B}_2^{(7)} \prec_{\mathtt{CB}} \mathtt{S}^{(7)})))
\end{aligned}
$$

Given global specification $\mathtt{G}$ over $\mathtt{n}$ parties, $\mathtt{P}_1 \cdots \mathtt{P}_\mathtt{n}$, our projection would transform $\mathtt{G}$ into a per-party specification, as follows:

$$\mathtt{G} \quad \Rightarrow \quad \mathtt{G\#P}_1 * \cdots * \mathtt{G\#P}_\mathtt{n} * \mathtt{G\#All}$$

where $\mathtt{G\#P}_1 * \cdots * \mathtt{G\#P}_\mathtt{n}$ denote the per-party specifications and $\mathtt{G\#All}$ denotes the global orderings that are shared by all parties.

Once we have a per-party specification, say $\mathtt{G\#P}_\mathtt{i}$ over channels $\mathtt{c}_1 \cdots \mathtt{c}_\mathtt{m}$, we can further project each of these specifications into a spatial conjunction of several per-channel specifications, namely $\mathtt{G\#P\#c}_1 * \cdots * \mathtt{G\#P\#c}_\mathtt{m}$, as follows:

$$\mathtt{G\#P}_\mathtt{i} \quad \Rightarrow \quad \mathtt{G\#P}_\mathtt{i}\mathtt{\#c}_1 * \cdots * \mathtt{G\#P}_\mathtt{i}\mathtt{\#c}_\mathtt{m}$$

For our running example, the seller and two buyers protocol can be projected into the three channels, as follows:

$$
\begin{aligned}
\mathtt{G}\#\mathtt{B}_1 \triangleq\ & \mathtt{s}!\mathtt{Order}; \oplus(\mathtt{B}_1^{(1)}); \mathtt{b}_1?\mathtt{Price}; \oplus(\mathtt{B}_1^{(2)}); \mathtt{b}_2!\mathtt{Amt}; \oplus(\mathtt{B}_1^{(4)}); \ominus(3 \prec_{\mathtt{HB}} 4)_{\mathtt{B}_1} \\
& \Downarrow \text{(per channel projection)} \\
& \mathtt{G}\#\mathtt{B}_1\#\mathtt{s} \triangleq\ !\mathtt{Order}; \oplus(\mathtt{B}_1^{(1)}) \\
& \mathtt{G}\#\mathtt{B}_1\#\mathtt{b}_1 \triangleq \ominus(\mathtt{B}_1^{(1)}); ?\mathtt{Price}; \oplus(\mathtt{B}_1^{(2)}) \\
& \mathtt{G}\#\mathtt{B}_1\#\mathtt{b}_2 \triangleq \ominus(\mathtt{B}_1^{(2)}); !\mathtt{Amt}; \oplus(\mathtt{B}_1^{(4)}); \ominus(3 \prec_{\mathtt{HB}} 4)_{\mathtt{B}_1} \\
\mathtt{G}\#\mathtt{B}_2 \triangleq\ & \mathtt{b}_2?\mathtt{Price}; \oplus(\mathtt{B}_2^{(3)}); \mathtt{b}_2?\mathtt{Amt}; \oplus(\mathtt{B}_2^{(4)}); \ominus(3 \prec_{\mathtt{HB}} 4)_{\mathtt{B}_2}; \\
& ((\mathtt{s}!\mathtt{No}; \oplus(\mathtt{B}_2^{(5)}); \ominus(1 \prec_{\mathtt{HB}} 5)_{\mathtt{B}_2}) \vee \\
& \ (\mathtt{s}!\mathtt{Yes}; \oplus(\mathtt{B}_2^{(6)}); \ominus(1 \prec_{\mathtt{HB}} 6)_{\mathtt{B}_2}; \mathtt{s}!\mathtt{Addr}; \oplus(\mathtt{B}_2^{(7)}); \ominus(6 \prec_{\mathtt{HB}} 7)_{\mathtt{B}_2})) \\
& \Downarrow \text{(per channel projection)} \\
& \mathtt{G}\#\mathtt{B}_2\#\mathtt{b}_2 \triangleq\ ?\mathtt{Price}; \oplus(\mathtt{B}_2^{(3)}); ?\mathtt{Amt}; \oplus(\mathtt{B}_2^{(4)}); \ominus(3 \prec_{\mathtt{HB}} 4)_{\mathtt{B}_2} \\
& \mathtt{G}\#\mathtt{B}_2\#\mathtt{s} \triangleq\ \ominus(\mathtt{B}_2^{(4)}); ((!\mathtt{No}; \oplus(\mathtt{B}_2^{(5)}); \ominus(1 \prec_{\mathtt{HB}} 5)_{\mathtt{B}_2}) \vee \\
& \qquad\qquad (!\mathtt{Yes}; \oplus(\mathtt{B}_2^{(6)}); \ominus(1 \prec_{\mathtt{HB}} 6)_{\mathtt{B}_2}; !\mathtt{Addr}; \oplus(\mathtt{B}_2^{(7)}); \ominus(6 \prec_{\mathtt{HB}} 7)_{\mathtt{B}_2})) \\
\mathtt{G}\#\mathtt{S} \triangleq\ & \mathtt{s}?\mathtt{Order}; \oplus(\mathtt{S}^{(1)}); (\mathtt{b}_1!\mathtt{Price}; \oplus(\mathtt{S}^{(2)})) * (\mathtt{b}_2!\mathtt{Price}; \oplus(\mathtt{S}^{(3)})); \ominus(3 \prec_{\mathtt{HB}} 4)_{\mathtt{S}}; \\
& ((\mathtt{s}?\mathtt{No}; \oplus(\mathtt{S}^{(5)}); \ominus(1 \prec_{\mathtt{HB}} 5)_{\mathtt{S}}) \vee \\
& \ (\mathtt{s}?\mathtt{Yes}; \oplus(\mathtt{S}^{(6)}); \ominus(1 \prec_{\mathtt{HB}} 6)_{\mathtt{S}}; \mathtt{s}?\mathtt{Addr}; \oplus(\mathtt{S}^{(7)}); \ominus(6 \prec_{\mathtt{HB}} 7)_{\mathtt{S}})) \\
& \Downarrow \text{(per channel projection)} \\
& \mathtt{G}\#\mathtt{S}\#\mathtt{b}_1 \triangleq \ominus(\mathtt{S}^{(1)}); !\mathtt{Price}; \oplus(\mathtt{S}^{(2)}) \\
& \mathtt{G}\#\mathtt{S}\#\mathtt{b}_2 \triangleq \ominus(\mathtt{S}^{(1)}); !\mathtt{Price}; \oplus(\mathtt{S}^{(3)}); \ominus(\mathtt{S}^{(2)}); \ominus(3 \prec_{\mathtt{HB}} 4)_{\mathtt{S}} \\
& \mathtt{G}\#\mathtt{S}\#\mathtt{s} \triangleq\ ?\mathtt{Order}; \oplus(\mathtt{S}^{(1)}); (\ominus(\mathtt{S}^{(2)}) * \ominus(\mathtt{S}^{(3)})); \\
& \qquad\qquad ((?\mathtt{No}; \oplus(\mathtt{S}^{(5)}); \ominus(1 \prec_{\mathtt{HB}} 5)_{\mathtt{S}}) \vee \\
& \qquad\qquad \ (?\mathtt{Yes}; \oplus(\mathtt{S}^{(6)}); \ominus(1 \prec_{\mathtt{HB}} 6)_{\mathtt{S}}; ?\mathtt{Addr}; \oplus(\mathtt{S}^{(7)}); \ominus(6 \prec_{\mathtt{HB}} 7)_{\mathtt{S}}))
\end{aligned}
$$

Each of these per-channel specification may actually contain event guards, such as $\ominus(\mathtt{S}^{(1)})$. These guards on events are important for ensuring that send/receive operations over different channels but within the same party are correctly synchronized. For example, we require that $?\mathtt{s} \cdot \mathtt{Order}$ be executed before $!\mathtt{b}_1 \cdot \mathtt{Price}$ within party $\mathtt{S}$, and this is ensured by placing $\ominus(\mathtt{S}^{(1)})$ before $!\mathtt{b}_1 \cdot \mathtt{Price}$.

4.1.3 Automated Local Verification

Once we have the refined specification that had been suitably projected on a per party and per channel basis, we can proceed to use these specifications for automated local verification.

The shared global orderings can be immediately added to the initial program state. The per-channel specification are released during the verification on each of the respective program codes, namely $(\texttt{Code}_\texttt{S} \mid\mid \texttt{Code}_{\texttt{B}_1} \mid\mid \texttt{Code}_{\texttt{B}_2})$, for the different parties. For our running example, the initial and final program states are:

$$\{\texttt{Common}(\texttt{G\#All}) * \texttt{Party}(\texttt{S}, \texttt{G\#S}) * \texttt{Party}(\texttt{B}_1, \texttt{G\#B}_1) * \texttt{Party}(\texttt{B}_2, \texttt{G\#B}_2)\}$$
$$(\texttt{Code}_\texttt{S} \mid\mid \texttt{Code}_{\texttt{B}_1} \mid\mid \texttt{Code}_{\texttt{B}_2})$$
$$\{\texttt{Party}(\texttt{S}, \texttt{emp}) * \texttt{Party}(\texttt{B}_1, \texttt{emp}) * \texttt{Party}(\texttt{B}_2, \texttt{emp})\}$$

The emp in the final state of each party can help to confirm the completion of all transmissions in the global protocol. The abstract predicate $\texttt{Party}(\texttt{S}, \texttt{G\#S})$ associates each party with its corresponding specification. Our approach supports both events that are either *immutable* or *mutable*. Each immutable event (seen earlier) is labelled uniquely, while mutable events can be updated flow-sensitively. To signify each executing party P, we use a mutable event $\texttt{Peer}(\texttt{P})$ that is updated when P is executing. Such mutable events are added to as ghost specifications, as shown below.

$$\{\texttt{Common}(\texttt{G\#All}) * \texttt{Party}(\texttt{S}, \texttt{G\#S}) * \texttt{Party}(\texttt{B}_1, \texttt{G\#B}_1) * \texttt{Party}(\texttt{B}_2, \texttt{G\#B}_2)\}$$
$$\{\texttt{Party}(\texttt{S}, \texttt{G\#S}) * \texttt{Party}(\texttt{B}_1, \texttt{G\#B}_1) * \texttt{Party}(\texttt{B}_2, \texttt{G\#B}_2) \wedge \texttt{G\#All}\}$$

$$\left(\begin{array}{l} \{\texttt{Party}(\texttt{S}, \texttt{G\#S}) \wedge \texttt{G\#All}\} \ \oplus(\texttt{Peer}(\texttt{S})); \texttt{Code}_\texttt{S} \ \{\texttt{Party}(\texttt{S}, \texttt{emp}) \wedge \texttt{V}_\texttt{S}\} \\ \mid\mid \\ \{\texttt{Party}(\texttt{B}_1, \texttt{G\#B}_1) \wedge \texttt{G\#All}\} \oplus(\texttt{Peer}(\texttt{B}_1)); \texttt{Code}_{\texttt{B}_1} \{\texttt{Party}(\texttt{B}_1, \texttt{emp}) \wedge \texttt{V}_{\texttt{B}_1}\} \\ \mid\mid \\ \{\texttt{Party}(\texttt{B}_2, \texttt{G\#B}_2) \wedge \texttt{G\#All}\} \oplus(\texttt{Peer}(\texttt{B}_2)); \texttt{Code}_{\texttt{B}_2} \{\texttt{Party}(\texttt{B}_2, \texttt{emp}) \wedge \texttt{V}_{\texttt{B}_2}\} \end{array}\right)$$

$$\{\texttt{Party}(\texttt{S}, \texttt{emp}) * \texttt{Party}(\texttt{B}_1, \texttt{emp}) * \texttt{Party}(\texttt{B}_2, \texttt{emp}) \wedge \texttt{V}_\texttt{S} \wedge \texttt{V}_{\texttt{B}_1} \wedge \texttt{V}_{\texttt{B}_2}\}$$
$$\{\texttt{Party}(\texttt{S}, \texttt{emp}) * \texttt{Party}(\texttt{B}_1, \texttt{emp}) * \texttt{Party}(\texttt{B}_2, \texttt{emp})\}$$

Lastly, to support local verification, we allow the per-channel specification to be released by

lemmas of the form:

$$\texttt{Party}(\texttt{B}_1, \texttt{G\#B}_1) \Leftrightarrow \mathcal{C}(\texttt{s}, \texttt{B}_1, \texttt{G\#B}_1\texttt{\#s}) * \mathcal{C}(\texttt{b}_1, \texttt{B}_1, \texttt{G\#B}_1\texttt{\#b}_1) * \mathcal{C}(\texttt{b}_2, \texttt{B}_1, \texttt{G\#B}_1\texttt{\#b}_2)$$

On completion, we also have a set of lemmas over the emp state to signify completion of protocol. An example is shown below:

$$\texttt{Party}(\texttt{B}_1, \texttt{emp}) \Leftrightarrow \mathcal{C}(\texttt{s}, \texttt{B}_1, \texttt{emp}) * \mathcal{C}(\texttt{b}_1, \texttt{B}_1, \texttt{emp}) * \mathcal{C}(\texttt{b}_2, \texttt{B}_1, \texttt{emp})$$

With the help of these lemmas, we can now allow local verification to proceed, as follows:

$$\{\texttt{Party}(\texttt{B}_1, \texttt{G\#B}_1) \wedge \texttt{G\#All}\}$$
$$\{\mathcal{C}(\texttt{s}, \texttt{B}_1, \texttt{G\#B}_1\texttt{\#s}) * \mathcal{C}(\texttt{b}_1, \texttt{B}_1, \texttt{G\#B}_1\texttt{\#b}_1) * \mathcal{C}(\texttt{b}_2, \texttt{B}_1, \texttt{G\#B}_1\texttt{\#b}_2) \wedge \texttt{G\#All}\}$$
$$\oplus(\texttt{Peer}(\texttt{B}_1)); \texttt{Code}_{\texttt{B}_1}$$
$$\{\mathcal{C}(\texttt{s}, \texttt{B}_1, \texttt{emp}) * \mathcal{C}(\texttt{b}_1, \texttt{B}_1, \texttt{emp}) * \mathcal{C}(\texttt{b}_2, \texttt{B}_1, \texttt{emp}) \wedge \texttt{V}_{\texttt{B}_1}\}$$
$$\{\texttt{Party}(\texttt{B}_1, \texttt{emp}) \wedge \texttt{V}_{\texttt{B}_1}\}$$
$$\{\texttt{Party}(\texttt{B}_1, \texttt{emp})\}$$

4.2 Global Protocols

This section formalizes the multiparty specification into a logical system whose syntax is depicted in Fig. 4.1. We assume the asynchronous communication model described in Sec. 2.6.

4.2.1 Formal Definitions

Transmission. As depicted in Fig. 4.1, a *transmission* $\texttt{S} \xrightarrow{i} \texttt{R} : \texttt{c}\langle \texttt{v} \cdot \Delta \rangle$ involves a sender S and a receiver R transmitting a message v expressed in logical form Δ (where Δ is *precise*) over a buffered channel c. This transmission is uniquely identified by a label i. In the subsequent we often use only the unique label i to refer to a particular transmission. To access the components of a transmission we define the following auxiliary functions: $\texttt{send}(i) \stackrel{\text{def}}{=} \texttt{S}^{(i)}$, $\texttt{recv}(i) \stackrel{\text{def}}{=} \texttt{R}^{(i)}$, $\texttt{chan}(i) \stackrel{\text{def}}{=} \texttt{c}$ and $\texttt{msg}(i) \stackrel{\text{def}}{=} \texttt{v} \cdot \Delta$. A transmission is irreflexive, since it would make no sense for the sending and the receiving to be performed by the same peer. We define a function $\texttt{TR}(\texttt{G})$ which decomposes a given protocol to collect a set of all its constituent transmissions, and a function $\texttt{TR}^{\texttt{fst}}(\texttt{G})$ to return the set of all possible first transmissions:

$$\begin{array}{llll}
\mathtt{TR}(\mathtt{G}_1;\mathtt{G}_2) & \stackrel{\text{def}}{=} \mathtt{TR}(\mathtt{G}_1)\cup\mathtt{TR}(\mathtt{G}_2) & \mathtt{TR}(\mathtt{S}\xrightarrow{i}\mathtt{R}:\mathtt{c}\langle\mathtt{v}\cdot\Delta\rangle) & \stackrel{\text{def}}{=} \{\mathtt{S}\xrightarrow{i}\mathtt{R}:\mathtt{c}\langle\mathtt{v}\cdot\Delta\rangle\}\\
\mathtt{TR}(\mathtt{G}_1 * \mathtt{G}_2) & \stackrel{\text{def}}{=} \mathtt{TR}(\mathtt{G}_1)\cup\mathtt{TR}(\mathtt{G}_2) & \mathtt{TR}(\oplus(\Psi)) & \stackrel{\text{def}}{=} \emptyset\\
\mathtt{TR}(\mathtt{G}_1 \vee \mathtt{G}_2) & \stackrel{\text{def}}{=} \mathtt{TR}(\mathtt{G}_1)\cup\mathtt{TR}(\mathtt{G}_2) & \mathtt{TR}(\ominus(\Psi)) & \stackrel{\text{def}}{=} \emptyset
\end{array}$$

$$\begin{array}{llll}
\mathtt{TR}^{\mathtt{fst}}(\mathtt{S}\xrightarrow{i}\mathtt{R}:\mathtt{c}\langle\mathtt{v}\cdot\Delta\rangle) & \stackrel{\text{def}}{=} \{\mathtt{S}\xrightarrow{i}\mathtt{R}:\mathtt{c}\langle\mathtt{v}\cdot\Delta\rangle\} & \mathtt{TR}^{\mathtt{fst}}(\mathtt{G}_1 * \mathtt{G}_2) & \stackrel{\text{def}}{=} \mathtt{TR}^{\mathtt{fst}}(\mathtt{G}_1)\cup\mathtt{TR}^{\mathtt{fst}}(\mathtt{G}_2)\\
\mathtt{TR}^{\mathtt{fst}}(\mathtt{G}_1;\mathtt{G}_2) & \stackrel{\text{def}}{=} \mathtt{TR}^{\mathtt{fst}}(\mathtt{G}_1) & \mathtt{TR}^{\mathtt{fst}}(\mathtt{G}_1 \vee \mathtt{G}_2) & \stackrel{\text{def}}{=} \mathtt{TR}^{\mathtt{fst}}(\mathtt{G}_1)\cup\mathtt{TR}^{\mathtt{fst}}(\mathtt{G}_2)
\end{array}$$

Event. An *event* $\mathtt{E}$ is a pair $\mathtt{P}^{(i)}$ where $\mathtt{P}\in\mathcal{R}\mathtt{ole}$ is the sending or receiving party of a transmission identified by $i\in\mathtt{Nat}$. Given the uniqueness of the identifier i, an event uniquely identifies a send or receive operation. Note that the irreflexive property of transmissions is important in uniquely identifying each event. If a party were allowed to communicate with itself, for example $\mathtt{P}\xrightarrow{i}\mathtt{P}:\mathtt{c}\langle\mathtt{v}\cdot\Delta\rangle$, then $\mathtt{P}^{(i)}$ would ambiguously refer to both the sending and the receiving events of transmission i. We denote by $\mathtt{E.peer}$ and $\mathtt{E.id}$ the elements of an event, e.g. $\mathtt{P}^{(i)}.\mathtt{peer}\stackrel{\text{def}}{=}\mathtt{P}$ and $\mathtt{P}^{(i)}.\mathtt{id}\stackrel{\text{def}}{=} i$. The following function collects the set of all the events within a protocol:

$$\begin{array}{llll}
\mathtt{EV}(\mathtt{G}_1;\mathtt{G}_2) & \stackrel{\text{def}}{=} \mathtt{EV}(\mathtt{G}_1)\cup\mathtt{EV}(\mathtt{G}_2) & \mathtt{EV}(\mathtt{S}\xrightarrow{i}\mathtt{R}:\mathtt{c}\langle\mathtt{v}\cdot\Delta\rangle) & \stackrel{\text{def}}{=} \{\mathtt{S}^{(i)},\mathtt{R}^{(i)}\}\\
\mathtt{EV}(\mathtt{G}_1 * \mathtt{G}_2) & \stackrel{\text{def}}{=} \mathtt{EV}(\mathtt{G}_1)\cup\mathtt{EV}(\mathtt{G}_2) & \mathtt{EV}(\mathtt{P}_1^{(i_1)}\prec_{\mathtt{HB}}\mathtt{P}_2^{(i_2)}) & \stackrel{\text{def}}{=} \{\mathtt{P}_1^{(i_1)},\mathtt{P}_2^{(i_2)}\}\\
\mathtt{EV}(\mathtt{G}_1 \vee \mathtt{G}_2) & \stackrel{\text{def}}{=} \mathtt{EV}(\mathtt{G}_1)\cup\mathtt{EV}(\mathtt{G}_2) & \mathtt{EV}(\mathtt{P}_1^{(i_1)}\prec_{\mathtt{CB}}\mathtt{P}_2^{(i_2)}) & \stackrel{\text{def}}{=} \{\mathtt{P}_1^{(i_1)},\mathtt{P}_2^{(i_2)}\}.
\end{array}$$

$$\mathtt{EV}(\mathtt{P}^{(i)}) \stackrel{\text{def}}{=} \{\mathtt{P}^{(i)}\} \qquad \mathtt{EV}(\oplus(\Psi)) \stackrel{\text{def}}{=} \mathtt{EV}(\Psi) \qquad \mathtt{EV}(\ominus(\Psi)) \stackrel{\text{def}}{=} \mathtt{EV}(\Psi)$$

The messages of two arbitrary but distinct transmissions, say i_1 and i_2, are said to be *disjoint*, denoted $\mathtt{msg}(i_1)\#\mathtt{msg}(i_2)$ if $\mathtt{UNSAT}(\Delta_{\mathtt{i}_1}\wedge[\mathtt{v}_1/\mathtt{v}_2]\Delta_{\mathtt{i}_2})$, where $\mathtt{msg}(i_1){=}\mathtt{v}_1\cdot\Delta_1$ and $\mathtt{msg}(i_2){=}\mathtt{v}_2\cdot\Delta_2$.

We abuse the set membership symbol, $\in$, to denote the followings:

$$\begin{array}{llll}
(\in_{\text{transm.}}) & i\in\mathtt{G} \;\Leftrightarrow\; i\in\mathtt{TR}(\mathtt{G}) & (\in_{\text{party}}) & \mathtt{P}\in i \;\Leftrightarrow\; \mathtt{send}(i).\mathtt{peer}=\mathtt{P}\text{ or}\\
(\in_{\text{channel}}) & \mathtt{c}\in i \;\Leftrightarrow\; \mathtt{chan}(i)=\mathtt{c} & & \mathtt{recv}(i).\mathtt{peer}=\mathtt{P}\\
(\in_{\text{channel}}) & \mathtt{c}\in\mathtt{G} \;\Leftrightarrow\; \exists i\in\mathtt{G}\cdot\mathtt{c}\in i & (\in_{\text{party}}) & \mathtt{P}\in\mathtt{G} \;\Leftrightarrow\; \exists i\in\mathtt{G}\cdot\mathtt{P}\in i
\end{array}$$

Correspondingly, $\notin$ is used to denote the negation of the above.

Transmissions are organized into a global protocol using a combination of parallel composition $\mathtt{G}_1 * \mathtt{G}_2$, disjunction $\mathtt{G}_1\vee\mathtt{G}_2$ and sequential composition $\mathtt{G}_1;\mathtt{G}_2$. The parallel composition of global protocols forms a commutative monoid $(\mathtt{G},*,\mathtt{emp})$ with $\mathtt{emp}$ as identity element, while

disjunction and sequence form semigroups, $(\mathtt{G}, \vee)$ and $(\mathtt{G}, ;)$, with the former also satisfying commutativity. $\mathtt{emp}$ acts the left identity element for sequential composition:

$$\begin{array}{rcl} (\mathtt{G}_1 \,;\, \mathtt{G}_2) \,;\, \mathtt{G}_3 & \equiv & \mathtt{G}_1 \,;\, (\mathtt{G}_2 \,;\, \mathtt{G}_3) \\ (\mathtt{G}_1 * \mathtt{G}_2) * \mathtt{G}_3 & \equiv & \mathtt{G}_1 * (\mathtt{G}_2 * \mathtt{G}_3) \\ (\mathtt{G}_1 \vee \mathtt{G}_2) \vee \mathtt{G}_3 & \equiv & \mathtt{G}_1 \vee (\mathtt{G}_2 \vee \mathtt{G}_3) \end{array} \qquad \begin{array}{rcl} \mathtt{G}_1 * \mathtt{G}_2 & \equiv & \mathtt{G}_2 * \mathtt{G}_1 \\ \mathtt{G}_1 \vee \mathtt{G}_2 & \equiv & \mathtt{G}_2 \vee \mathtt{G}_1 \end{array} \qquad \begin{array}{rcl} \mathtt{G} * \mathtt{emp} & \equiv & \mathtt{G} \\ \mathtt{emp} \,;\, \mathtt{G} & \equiv & \mathtt{G} \end{array}$$

Sequential composition is not commutative, unless it satisfies certain disjointness properties:

$$\mathtt{G}_1 \,;\, \mathtt{G}_2 \equiv \mathtt{G}_2 \,;\, \mathtt{G}_1 \text{ when } \exists \mathtt{c}_1 \in \mathtt{G}_1 \wedge \exists \mathtt{c}_2 \in \mathtt{G}_2 \text{ and } \forall \mathtt{c}_1 \in \mathtt{G}_1, \mathtt{c}_2 \in \mathtt{G}_2 \Rightarrow \mathtt{c}_1 \neq \mathtt{c}_2$$
$$\text{and } \forall \mathtt{P}_1 \in \mathtt{G}_1, \mathtt{P}_2 \in \mathtt{G}_2 \Rightarrow \mathtt{P}_1 \neq \mathtt{P}_2$$

Sub-protocol. A protocol $\mathtt{G}'$ is said to be a sub-protocol of $\mathtt{G}$ when $\mathtt{G}'$ is a decomposition of $\mathtt{G}$, where the set of all possible decompositions of $\mathtt{G}$ is recursively defined on the structure of $\mathtt{G}$ as follows:

$$\begin{array}{lll} \mathtt{sub}(\mathtt{S} \xrightarrow{i} \mathtt{R} : \mathtt{c}\langle \mathtt{v} \cdot \Delta \rangle) \stackrel{\text{def}}{=} \mathtt{sub}(\mathtt{S} \xrightarrow{i} \mathtt{R} : \mathtt{c}\langle \mathtt{v} \cdot \Delta \rangle) & \mathtt{sub}(\mathtt{G}_1; \mathtt{G}_2) & \stackrel{\text{def}}{=} \{\mathtt{G}_1; \mathtt{G}_2\} \cup \mathtt{sub}(\mathtt{G}_1) \cup \mathtt{sub}(\mathtt{G}_2) \\ \mathtt{sub}(\oplus(\Psi)) \stackrel{\text{def}}{=} \emptyset & \mathtt{sub}(\mathtt{G}_1 \vee \mathtt{G}_2) & \stackrel{\text{def}}{=} \{\mathtt{G}_1 \vee \mathtt{G}_2\} \cup \mathtt{sub}(\mathtt{G}_1) \cup \mathtt{sub}(\mathtt{G}_2) \\ \mathtt{sub}(\ominus(\Psi)) \stackrel{\text{def}}{=} \emptyset & \mathtt{sub}(\mathtt{G}_1 * \mathtt{G}_2) & \stackrel{\text{def}}{=} \{\mathtt{G}_1 * \mathtt{G}_2\} \cup \mathtt{sub}(\mathtt{G}_1) \cup \mathtt{sub}(\mathtt{G}_2) \end{array}$$

Graph ordering. We capture global protocol in terms of a directed acyclic graph $\mathcal{G}(\mathtt{G}) \stackrel{\text{def}}{=} (\mathtt{V}, \mathtt{O})$, where the set of vertices is the set of all the transmissions in $\mathtt{G}$, $\mathtt{V} = \mathtt{TR}(\mathtt{G})$, and the edges represent the sequence relation between these nodes, $\mathtt{P} = \mathtt{seq}(\mathtt{G})$, where $\mathtt{seq}$ is defined as follows:

$$\mathtt{seq}(\mathtt{G}_1; \mathtt{G}_2) \stackrel{\text{def}}{=} \{(i_1, i_2) | i_1 \in \mathtt{TR}(\mathtt{G}_1) \wedge i_2 \in \mathtt{TR}(\mathtt{G}_2)\} \cup \mathtt{seq}(\mathtt{G}_1) \cup \mathtt{seq}(\mathtt{G}_2).$$
$$\mathtt{seq}(\oplus(\Psi)) \stackrel{\text{def}}{=} \emptyset \,. \quad \mathtt{seq}(\ominus(\Psi)) \stackrel{\text{def}}{=} \emptyset. \quad \mathtt{seq}(\mathtt{S} \xrightarrow{i} \mathtt{R} : \mathtt{c}\langle \mathtt{v} \cdot \Delta \rangle) \stackrel{\text{def}}{=} \emptyset \,.$$
$$\mathtt{seq}(\mathtt{G}_1 * \mathtt{G}_2) \stackrel{\text{def}}{=} \mathtt{seq}(\mathtt{G}_1) \cup \mathtt{seq}(\mathtt{G}_2). \qquad \mathtt{seq}(\mathtt{G}_1 \vee \mathtt{G}_2) \stackrel{\text{def}}{=} \mathtt{seq}(\mathtt{G}_1) \cup \mathtt{seq}(\mathtt{G}_2).$$

Two transmissions, i_1 and i_2, are *sequenced* in a protocol $\mathtt{G}$, denoted $i_1 \prec i_2$, if there is a path in $\mathcal{G}(\mathtt{G})$ from i_1 to i_2. Two transmissions, i_1 and i_2, are *adjacent* in $\mathtt{G}$ if they share the same channel $\mathtt{c}$, they are sequenced in $\mathcal{G}(\mathtt{G})$, and there are no other transmission on $\mathtt{c}$ in between i_1 and i_2. And lastly, two transmissions are *linked* if they are sequenced and they share the same channel. These relations are formally described in Fig. 4.2. Since these relations are defined as edges of a directed acyclic group it is straightforward to show that they are irreflexive and antisymetric. The transitivity of sequence also follows directly from the reachability relation

Given a global protocol G:
Sequenced
$(i_1, i_2) \in \mathtt{seq}(\mathtt{G})$ is denoted by $i_1 \prec i_2$
Adjacent
$\mathtt{Adj}(i_1, i_2) \stackrel{\text{def}}{=} \mathtt{chan}(i_1){=}\mathtt{chan}(i_2) \wedge i_1 \prec i_2 \wedge \neg\exists i' \cdot (\mathtt{chan}(i_1){=}\mathtt{chan}(i') \wedge i_1 \prec i' \wedge i' \prec i_2)$.
Linked
$\mathtt{Adj}^+(i_1, i_2) \stackrel{\text{def}}{=} \mathtt{Adj}(i_1, i_2) \vee (\exists i' \cdot \mathtt{Adj}(i_1, i') \wedge \mathtt{Adj}^+(i', i_2))$.

Figure 4.2: Transmission Sequencing

of DAGs which is the transitive closure of the edges in $\mathcal{G}(\mathtt{G})$. A simple case analysis on the definition of linked transmissions shows that the linked relation is also transitive.

4.2.2 Well-Formedness

Concurrency. The $*$ operator offers support for arbitrary-ordered (concurrent) transmissions, where the completion order is not important for the final outcome.

Definition 4 (Well-Formed Concurrency)**.** *A protocol specification,* $\mathtt{G}_1 * \mathtt{G}_2$*, is said to be well-formed with respect to* $*$ *if and only if* $\forall \mathtt{c} {\in} \mathtt{G}_1 \Rightarrow \mathtt{c} {\notin} \mathtt{G}_2$*, and vice versa.*

This restriction is to avoid non-determinism from concurrent communications over the same channel.

Choice. The $\vee$ operator is essential for the expressiveness of proposed logic but its usage must be carefully controlled:

Definition 5 (Well-Formed Choice)**.** *A disjunctive protocol specification,* $\mathtt{G}_1 \vee \mathtt{G}_2$*, is said to be well-formed with respect to* $\vee$ *if and only if all of the following conditions hold, where* $\mathtt{T}_1$ *and* $\mathtt{T}_2$ *account for all first transmissions of* $\mathtt{G}_1$ *and* $\mathtt{G}_2$*, respectively,* $\mathtt{T}_1{=}\mathtt{TR}^{\mathtt{fst}}(\mathtt{G}_1)$ *and* $\mathtt{T}_2{=}\mathtt{TR}^{\mathtt{fst}}(\mathtt{G}_2)$*:*

(a) (same first channel) $\forall i_1, i_2 \in \mathtt{T}_1 \cup \mathtt{T}_2 \Rightarrow \mathtt{chan}(i_1) = \mathtt{chan}(i_2)$;

(b) (same first sender) $\forall i_1, i_2 \in \mathtt{T}_1 \cup \mathtt{T}_2 \Rightarrow \mathtt{send}(i_1).\mathtt{peer} = \mathtt{send}(i_2).\mathtt{peer}$;

(c) (same first receiver) $\forall i_1, i_2 \in \mathtt{T}_1 \cup \mathtt{T}_2 \Rightarrow \mathtt{recv}(i_1).\mathtt{peer} = \mathtt{recv}(i_2).\mathtt{peer}$;

(d) (mutually exclusive "first" messages) $\forall i_1, i_2 \in \mathtt{T}_1 \cup \mathtt{T}_2 \Rightarrow \mathtt{msg}(i_1) \# \mathtt{msg}(i_2) \vee i_1{=}i_2$;

(e) (same peers)

$\forall i_1, i_2 \in \mathtt{G}_1 \cup \mathtt{G}_2 \Rightarrow \{\mathtt{send}(i_1).\mathtt{peer}, \mathtt{recv}(i_1).\mathtt{peer}\} = \{\mathtt{send}(i_2).\mathtt{peer}, \mathtt{recv}(i_2).\mathtt{peer}\}$;

(f) (recursive well-formedness) $\mathtt{G}_1$ *and* $\mathtt{G}_2$ *are well-formed with respect to* $\vee$.

Definition 6 (Well-Formed Protocol)**.** *A protocol* G *is said to be well-formed, if and only if* G *contains only well-formed concurrent sub-protocols, and well-formed choices.*

To ensure the correctness of our approach, we disregard as unsound any usage of $*$ or $\vee$ which is not well-formed.

4.2.3 Protocol Safety with Refinement

As highlighted in the overview of this chapter, even simple protocols which only involve two transmissions can easily lead to flimsy communication, where either the receivers, (A→C ; B→C), or the senders, (A→B ; A→C), race for the same channel.

To avoid such race conditions it is essential to study the linearity of the communication. A communication is said to be linear when all the shared channels are used in a linear fashion, or in other words when a send and its corresponding receive are temporally ordered. Since the communication implementation is guided by the communication protocol, it is therefore essential for the protocol to satisfy this safety principle as well. In the subsequent we proceed in:

- defining a minimal set of causality relations relevant in the study of linearity and build an ordering system to reason about these relations.
- defining race-freedom w.r.t. these causality relations in the context of asynchronous communication.
- transform any given protocol into a race-free protocol via a refinement phase which adds assumptions about the causal relation between ordered events, and guards to enforce race-free communication.

Ordering Constraint System. Given a set of events $\mathcal{E}\mathtt{vents}$ and a relation $\mathcal{R} \subseteq \mathcal{E}\mathtt{vents} \times \mathcal{E}\mathtt{vents}$, we denote by $\mathtt{E}_1 \prec_{\mathcal{R}} \mathtt{E}_2$ the fact that $(\mathtt{E}_1, \mathtt{E}_2) \in \mathcal{R}$. We next distinguish between two kinds of relations: the so called *"happens-before"* relations to reflect the temporal ordering between events, and *"communicates-before"* relations to relate communicating peers.

Definition 7 (Communicates-before)**.** *A communicates-before relation* CB *, denoted by* $\mathtt{E}_1 \prec_{\mathtt{CB}} \mathtt{E}_2$*, is defined as:*

$$\{(\mathtt{E}_1, \mathtt{E}_2) \mid \exists\, i \cdot \mathtt{E}_1.\mathtt{id}{=}\mathtt{E}_2.\mathtt{id}{=}i \text{ and } \mathtt{E}_1{=}\mathtt{send}(i) \text{ and } \mathtt{E}_2{=}\mathtt{recv}(i)\}.$$

$$\begin{array}{lll} \textit{Send/Recv Event} & \mathtt{E} & ::= \mathtt{P}^{(i)} \\ \textit{Ordering Constraints} & \vartheta & ::= \mathtt{E} \prec_{\mathtt{CB}} \mathtt{E} \mid \mathtt{E} \prec_{\mathtt{HB}} \mathtt{E} \\ \textit{Race-Free Assertions} & \Psi & ::= \mathtt{E} \mid \neg(\mathtt{E}) \mid \vartheta \mid \Psi \wedge \Psi \mid \mathtt{E} \Rightarrow \Psi \end{array}$$

(a) Syntax of the ordering-constraints language

$$\begin{array}{lll} \mathtt{E}_1 \prec_{\mathtt{HB}} \mathtt{E}_2 \wedge \mathtt{E}_2 \prec_{\mathtt{HB}} \mathtt{E}_3 & \Rightarrow \mathtt{E}_1 \prec_{\mathtt{HB}} \mathtt{E}_3 & \text{[HB-HB]} \\ \mathtt{E}_1 \prec_{\mathtt{CB}} \mathtt{E}_2 \wedge \mathtt{E}_2 \prec_{\mathtt{HB}} \mathtt{E}_3 & \Rightarrow \mathtt{E}_1 \prec_{\mathtt{HB}} \mathtt{E}_3 & \text{[CB-HB]} \end{array}$$

(b) Constraint propagation rule

$$\begin{array}{llll} \Pi \vDash_{\mathtt{RF}} \mathtt{E} & \text{iff } \mathtt{E} \in \Pi & \Pi \vDash_{\mathtt{RF}} \mathtt{E} \Rightarrow \Psi & \text{iff } \neg(\Pi \vDash_{\mathtt{RF}} \mathtt{E}) \text{ or } \Pi \vDash_{\mathtt{RF}} \Psi \\ \Pi \vDash_{\mathtt{RF}} \neg(\mathtt{E}) & \text{iff } \mathtt{E} \notin \Pi & \Pi \vDash_{\mathtt{RF}} \Psi_1 \wedge \Psi_2 & \text{iff } \Pi \vDash_{\mathtt{RF}} \Psi_1 \text{ and } \Pi \vDash_{\mathtt{RF}} \Psi_2 \end{array}$$

$$\Pi \vDash_{\mathtt{RF}} \mathtt{E}_1 \prec_{\mathtt{HB}} \mathtt{E}_2 \quad \text{iff } (\bigwedge_{\Psi \in \Pi} \Psi) \Rightarrow^* \mathtt{E}_1 \prec_{\mathtt{HB}} \mathtt{E}_2$$

(c) Semantics of race-free assertions

Figure 4.3: The Ordering-Constraints Language

The CB relation is not transitive, since its purpose is to simply assist the transitivity of HB by relating events which involve different peers. The CB is however irreflexive since we do not allow for a message to be sent and received by the same party, and antisymmetric since each event is unique (described by the $\mathcal{R}\mathtt{ole} \times \mathtt{Nat}$ pair), that is, it only occurs within a single transmission.

Definition 8 (Happens-before)**.** *Given a global protocol* $\mathtt{G}$*, two events* $\mathtt{E}_1 \in \mathtt{EV}(\mathtt{G})$ *and* $\mathtt{E}_2 \in \mathtt{EV}(\mathtt{G})$ *are said to be in a happens-before relation in* $\mathtt{G}$ *if* $(\mathtt{E}_1.\mathtt{id}, \mathtt{E}_2.\mathtt{id}) \in \mathtt{seq}(\mathtt{G})$ *or it can be derived via a closure using the propagation rules [HB-HB] and [CB-HB]. This is denoted by* $\mathtt{E}_1 \prec_{\mathtt{HB}} \mathtt{E}_2$.

Similar to the happened-before a la Lamport [61], the HB relation is transitive, irreflexive and asymmetric, therefore HB is a strict partial order over events. However, one *novel* aspect of our approach is the use of [CB-HB] propagation lemma.

Lemma 1 (Soundness of [CB-HB] Lemma)**.** *Consider two possible propagation lemmas involving the [CB] and [HB] ordering.*

$$\begin{array}{ll} \mathtt{E}_1 \prec_{\mathtt{HB}} \mathtt{E}_2 \wedge \mathtt{E}_2 \prec_{\mathtt{CB}} \mathtt{E}_3 & \Rightarrow \mathtt{E}_1 \prec_{\mathtt{HB}} \mathtt{E}_3 \quad \textit{[HB-CB]} \\ \mathtt{E}_1 \prec_{\mathtt{CB}} \mathtt{E}_2 \wedge \mathtt{E}_2 \prec_{\mathtt{HB}} \mathtt{E}_3 & \Rightarrow \mathtt{E}_1 \prec_{\mathtt{HB}} \mathtt{E}_3 \quad \textit{[CB-HB]} \end{array}$$

Under the context of proving race freedom for each pair of adjacent transmissions over some common channel, it is sound to use [CB-HB] but not [HB-CB].

Proof: Consider a pair of adjacent transmissions over some common channel to be $\mathtt{S}_1 \prec_{\mathtt{CB}} \mathtt{R}_1$ and $\mathtt{S}_2 \prec_{\mathtt{CB}} \mathtt{R}_2$. Assume we have already established $\mathtt{R}_1 \prec_{\mathtt{HB}} \mathtt{S}_2$, and that no other transmissions with the same channel are under consideration. From this scenario, we are unable to conclude $\mathtt{R}_1 \prec_{\mathtt{HB}} \mathtt{R}_2$ since the following sequence $\mathtt{S}_1; \mathtt{R}_2; \mathtt{R}_1; \mathtt{S}_2$ is possible which is consistent $\mathtt{R}_1 \prec_{\mathtt{HB}} \mathtt{S}_2$ but yet violate $\mathtt{R}_1 \prec_{\mathtt{HB}} \mathtt{R}_2$. Hence, [HB-CB] is unsound to use. However, we can conclude $\mathtt{S}_1 \prec_{\mathtt{HB}} \mathtt{S}_2$ since for $\mathtt{R}_1 \prec_{\mathtt{HB}} \mathtt{S}_2$ to hold, it must be the case that $\mathtt{S}_1 \prec_{\mathtt{HB}} \mathtt{R}_1$. This is because $\mathtt{R}_1 \prec_{\mathtt{HB}} \mathtt{S}_1$ is not possible, since at least one send must preceed $\mathtt{R}_1$. By transitivity of $\prec_{\mathtt{HB}}$, we can therefore conclude $\mathtt{S}_1 \prec_{\mathtt{HB}} \mathtt{S}_2$. Hence, it is sound to use [CB-HB] for proving race freedom for each pair of adjacent transmissions. □

To reason about these orderings, we propose a constraint ordering system whose syntax is depicted in Fig. 4.3a. The send and receive events are related using either a HB or a HB relation, while the ordering constraints are composed using either $\wedge$ or $\vee$. The language supports $\neg(\mathtt{E})$ to indicate that an (immutable) event has not occurred yet.

The semantics of the race-free assertions and ordering constraints is given in Fig. 4.3c, where the proof context is a set of orderings, Π, with elements from Fig. 4.3a. The implication, $\Rightarrow^*$ is solved by repeatedly applying the propagation rules ([CB-HB], [HB-HB]) from Fig. 4.3b.

Race-free protocol. For brevity of the presentation, the transmitted messages are ignored in the rest of this subsection, that is, a transmission is described as a tuple $\mathcal{R}\mathtt{ole} \times \mathcal{R}\mathtt{ole} \times \mathtt{nat} \times \mathcal{L}\mathtt{chan}$.

Definition 9 (Race-free adjacent transmissions)**.** *Given a protocol* $\mathtt{G}$*, two adjacent transmissions* $i_1 \in \mathtt{G}$ *and* $i_2 \in \mathtt{G}$ *are said to be race-free w.r.t. each other, denoted by* $\mathtt{RF}(i_1, i_2)$*, only when they are in a HB relation:*

$$\forall i_1, i_2 \in \mathtt{G} \cdot (\mathtt{Adj}(i_1, i_2) \Rightarrow i_1 \prec_{\mathtt{HB}} i_2)$$

where $i_1 \prec_{\mathtt{HB}} i_2$ *stands for:*

$$\mathtt{send}(i_1) \prec_{\mathtt{HB}} \mathtt{send}(i_2) \wedge \mathtt{recv}(i_1) \prec_{\mathtt{HB}} \mathtt{recv}(i_2).$$

Definition 10 (Race-free protocol)**.** *A protocol* $\mathtt{G}$ *is race-free, denoted by* $\mathtt{RF}(\mathtt{G})$*, when all the linked transmissions are in a HB relation:*

$$\forall i_1, i_2 \in \mathtt{G} \cdot (\mathtt{Adj}^+(i_1, i_2) \Rightarrow i_1 \prec_{\mathtt{HB}} i_2).$$

Theorem 1 (Race-free protocol)**.** *A protocol is race-free if every pair of adjacent transmissions is race free:*

$$(\forall i_1, i_2 \in \mathtt{G} \Rightarrow \mathtt{RF}(i_1, i_2)) \Rightarrow \mathtt{RF}(\mathtt{G}).$$

Proof: Using inductive proving on the definition of linked transmissions and the transitivity of HB we can show that :

$$\dfrac{\dfrac{\dfrac{\texttt{true}}{\forall i_1, i_2 \in \texttt{G} \cdot \texttt{Adj}(i_1, i_2) \Rightarrow i_1 \prec_{\texttt{HB}} i_2}\,[\text{PREM}] \quad \dfrac{\dfrac{\texttt{true}}{\forall i \cdot \texttt{Adj}(i, i') \Rightarrow i \prec_{\texttt{HB}} i'}\,[\text{PREM}] \quad \dfrac{\texttt{true}}{\forall i \cdot \texttt{Adj}^+(i', i) \Rightarrow i' \prec_{\texttt{HB}} i}\,[\text{IND}]}{\forall i_1, i_2 \in \texttt{G} \cdot \exists\, i' \cdot \texttt{Adj}(i_1, i') \wedge \texttt{Adj}^+(i', i_2) \Rightarrow i_1 \prec_{\texttt{HB}} i_2}}{\forall i_1, i_2 \in \texttt{G} \cdot \texttt{Adj}(i_1, i_2) \vee \exists\, i' \cdot \texttt{Adj}(i_1, i') \wedge \texttt{Adj}^+(i', i_2) \Rightarrow i_1 \prec_{\texttt{HB}} i_2}}{\forall i_1, i_2 \in \texttt{G} \cdot (\texttt{Adj}^+(i_1, i_2) \Rightarrow i_1 \prec_{\texttt{HB}} i_2)}\,[\text{Def. 10}]$$

□

As mentioned before, we are assuming a hybrid communication system, where both message passing and other explicit synchronization mechanisms are used. Therefore, a simple analysis to check whether the protocol is linear is insufficient. Instead of analyzing the protocol for channels which race, we designed an algorithm - **Algorithm 1** - which inserts the race-free constraints as explicit proof obligations within the protocol. We rely on the program verification to detect when these constraints are not satisfied.

Algorithm 1: Decorates a protocol with ordering assumptions, and race-free guards

input : G - a global multi-party protocol

output: G′ - refined G

1 $\mathcal{S} \leftarrow$ `collect`(G)

2 G′ $\leftarrow$ `addGuards`(G,$\mathcal{S}^{\ominus}$)

3 G′ $\leftarrow$ `addAssumptions`(G′,$\mathcal{S}^{\oplus}$)

4 **return** G′

The algorithm is pretty straightforward: it first collects the necessary ordering assumptions and race-free guards, and then inserts them within the global protocol using a simple scheme guided by the unique transmission identifier. The methods used for insertion are not described here for limit of space, but also because they do not represent any theoretical or technical interest. As an intuition though, the insertion follows these principles: $\oplus(\texttt{P}^{(i)})$ is inserted immediately after transmission i; $\oplus(\texttt{S}^{(i)} \prec_{\texttt{CB}} \texttt{R}^{(i)})$ is inserted immediately after transmission i; $\oplus(\texttt{P}_1^{(i_1)} \prec_{\texttt{HB}} \texttt{P}_2^{(i_2)})$ and $\ominus(\texttt{P}_1^{(i_1)} \prec_{\texttt{HB}} \texttt{P}_2^{(i_2)})$ are inserted immediately after transmission i_2. According to **Algorithm 1** the assumptions are added to the protocol after inserting the guards, therefore, in the refined protocol, the assumptions precede the guards relative to the same transmission.

The `collect` method derives the ordering relations necessary for the refinement of a user-defined protocol into a race-free protocol. In a top-down approach, `collect` compiles protocol summaries for each compositional sub-protocol, until each sub-protocol is reduced to a single

$$
\begin{array}{lll}
\textit{Base Element} & \texttt{BForm a} & ::= \texttt{a} \mid (\texttt{BForm a}) * (\texttt{BForm a}) \\
\textit{Border Element} & \texttt{EForm a} & ::= \bot \mid \texttt{BForm a} \mid (\texttt{EForm a}) \vee (\texttt{EForm a}) \\
\textit{Border Event} & \beta^{\texttt{E}} & ::= \texttt{EForm P}^{(i)} \\
\textit{Border Transmission} & \beta^{\texttt{T}} & ::= \texttt{EForm P} \xrightarrow{i} \texttt{P : c}
\end{array}
$$

$$
\textit{(Operation Map)}\ \texttt{RMap} \stackrel{\text{def}}{=} \mathcal{R}\texttt{ole} \rightarrow \beta^{\texttt{E}} \qquad \textit{(Transmission Map)}\ \texttt{CMap} \stackrel{\text{def}}{=} \mathcal{L}\texttt{chan} \rightarrow \beta^{\texttt{T}}
$$

$$
\textit{(Boundary)}\ \texttt{Border} \stackrel{\text{def}}{=} \texttt{RMap} \times \texttt{CMap}
$$

Figure 4.4: Elements of the Boundary Summary.

transmission. It then gradually merges each such summary to obtain the global view of the all orderings within the initial protocol.

Protocol Summary. The summary of a protocol G is a tuple $\mathcal{S}{=}\langle \texttt{B}, \texttt{F}, \texttt{A}^{\oplus}, \texttt{A}^{\ominus} \rangle$, where B is the left boundary, also called a *backtier*, mapping roles to their first occurrence and channels to their first transmissions within G, F is the right boundary, also called *frontier*, mapping roles to their last occurrence and channels to their last transmissions within G, $\texttt{A}^{\oplus}$ is a set of ordering assumptions which reflect the implicit synchronization ordering relations, and $\texttt{A}^{\ominus}$ is a set of ordering proof obligations needed to ensure the race-freedom of G. We denote by $\mathcal{S}^{\texttt{bt}}$, $\mathcal{S}^{\texttt{ft}}$, $\mathcal{S}^{\oplus}$ and $\mathcal{S}^{\ominus}$ the elements of the summary. The $\mathcal{S}^{\oplus}$ and $\mathcal{S}^{\ominus}$ sets are the result of fusing together the communication summaries of each compositional protocol.

Protocol Boundary. Each boundary, whether left or right, is a pair of maps $\langle \texttt{K}, \Gamma \rangle$, where K relates protocol roles to events and Γ relates channels to transmission. The fact that a map is not defined for a particular input is indicated by $\bot$. The constituent elements of these maps are formally described in Fig. 4.4, where $\beta^{\texttt{E}}$ is used to populate the events map - K, and the $\beta^{\texttt{T}}$ elements populate the transmissions map - Γ. We denote by F.K and F.Γ the elements of a boundary F.

Finally, the decomposition of the protocol into sub-protocols and the building of summaries is recursively defined as below, where the helper functions, $\texttt{h}_1$, $\texttt{h}_2$ and $\texttt{h}_3$, merge the summaries specific to each protocol composition operator:

$$\begin{aligned}
\texttt{collect}(\texttt{G}_1;\texttt{G}_2) &\stackrel{\text{def}}{=} \texttt{h}_1(\texttt{collect}(\texttt{G}_1),\texttt{collect}(\texttt{G}_2))\\
\texttt{collect}(\texttt{G}_1 * \texttt{G}_2) &\stackrel{\text{def}}{=} \texttt{h}_2(\texttt{collect}(\texttt{G}_1),\texttt{collect}(\texttt{G}_2))\\
\texttt{collect}(\texttt{G}_1 \vee \texttt{G}_2) &\stackrel{\text{def}}{=} \texttt{h}_3(\texttt{collect}(\texttt{G}_1),\texttt{collect}(\texttt{G}_2))\\
\texttt{collect}(\texttt{S}\xrightarrow{i}\texttt{R} : \texttt{c}) &\stackrel{\text{def}}{=} \langle\langle \texttt{K},\Gamma\rangle,\ \langle \texttt{K},\Gamma\rangle,\ \{\oplus(\texttt{S}\xrightarrow{i}\texttt{R})\},\ \emptyset\rangle\\
&\quad \text{where } \texttt{K}=(\texttt{S} : \texttt{S}^{(i)}, \texttt{R} : \texttt{R}^{(i)}) \text{ and } \Gamma=(\texttt{c} : \texttt{S}\xrightarrow{i}\texttt{R})
\end{aligned}$$

$$\begin{aligned}
\texttt{h}_1(\mathcal{S}_1,\mathcal{S}_2) &\stackrel{\text{def}}{=} \langle \mathcal{S}_1^{\texttt{bt}} \lfloor;\rfloor \mathcal{S}_2^{\texttt{bt}}, \mathcal{S}_2^{\texttt{ft}} \lfloor;\rfloor \mathcal{S}_1^{\texttt{ft}}, \mathcal{S}_1^{\oplus} \cup \mathcal{S}_2^{\oplus} \cup \texttt{A}_3^{\oplus}, \mathcal{S}_1^{\ominus} \cup \mathcal{S}_2^{\ominus} \cup \texttt{A}_3^{\ominus}\rangle\\
&\quad \text{where } \langle \texttt{A}_3^{\oplus}, \texttt{A}_3^{\ominus}\rangle \ = \ \texttt{merge_adjacent}(\mathcal{S}_1^{\texttt{ft}}, \mathcal{S}_2^{\texttt{bt}})\\
\texttt{h}_2(\mathcal{S}_1,\mathcal{S}_2) &\stackrel{\text{def}}{=} \langle \mathcal{S}_1^{\texttt{bt}} \lfloor *\rfloor \mathcal{S}_2^{\texttt{bt}}, \mathcal{S}_1^{\texttt{ft}} \lfloor *\rfloor \mathcal{S}_2^{\texttt{ft}}, \mathcal{S}_1^{\oplus} \cup \mathcal{S}_2^{\oplus}, \mathcal{S}_1^{\ominus} \cup \mathcal{S}_2^{\ominus}\rangle\\
\texttt{h}_3(\mathcal{S}_1,\mathcal{S}_2) &\stackrel{\text{def}}{=} \langle \mathcal{S}_1^{\texttt{bt}} \lfloor \vee\rfloor \mathcal{S}_2^{\texttt{bt}}, \mathcal{S}_1^{\texttt{ft}} \lfloor \vee\rfloor \mathcal{S}_2^{\texttt{ft}}, \mathcal{S}_1^{\oplus} \cup \mathcal{S}_2^{\oplus}, \mathcal{S}_1^{\ominus} \cup \mathcal{S}_2^{\ominus}\rangle
\end{aligned}$$

where the $\lfloor \texttt{op} \rfloor$ operator (over-loaded over boundaries and maps' elements) represents the fusion specific to each protocol connector:

$$\begin{aligned}
\langle \texttt{K}_1,\Gamma_1\rangle \lfloor \texttt{op}\rfloor \langle \texttt{K}_2,\Gamma_2\rangle &\stackrel{\text{def}}{=} \langle \texttt{K},\Gamma\rangle\\
\text{where } \texttt{K}&=\texttt{K}_1[\texttt{P} : \texttt{K}_1(\texttt{P}) \lfloor \texttt{op}\rfloor \texttt{K}_2(\texttt{P})]_{\forall \texttt{P}\in \texttt{dom}(\texttt{K}_2)}\\
\Gamma&=\Gamma_1[\texttt{c} : \Gamma_1(\texttt{c}) \lfloor \texttt{op}\rfloor \Gamma_2(\texttt{c})]_{\forall \texttt{c}\in \texttt{dom}(\Gamma_2)}
\end{aligned}$$

$$\beta_1 \lfloor;\rfloor \beta_2 \stackrel{\text{def}}{=} \begin{cases} \beta_2 & \text{if } \beta_1 = \bot \\ \beta_1 & \text{otherwise} \end{cases} \qquad \beta_1 \lfloor *\rfloor \beta_2 \stackrel{\text{def}}{=} \begin{cases} \beta_1 & \text{if } \beta_2 = \bot \\ \beta_2 & \text{if } \beta_1 = \bot \\ \beta_1 * \beta_2 & \text{otherwise} \end{cases}$$

$$\beta_1 \lfloor \vee\rfloor \beta_2 \stackrel{\text{def}}{=} \beta_1 \vee \beta_2$$

$\texttt{K}[\texttt{P} : \beta^{\texttt{E}}]$ denotes an update to $\texttt{K}$, such that the value corresponding to $\texttt{P}$ is updated to $\beta^{\texttt{E}}$, even if $\texttt{K}$ was previously not defined on $\texttt{P}$. Similarly, $\Gamma[\texttt{c} : \beta^{\texttt{T}}]$ indicates an update to Γ such that $\texttt{c}$ is mapped to $\beta^{\texttt{T}}$.

In the case of backtier fusion for sequence, $\mathcal{S}_1^{\texttt{bt}} \lfloor;\rfloor \mathcal{S}_2^{\texttt{bt}}$ ensures that the resulted backtier reflects all the first transmissions for each channel employed by either $\texttt{G}_1$ or $\texttt{G}_2$, whichever first, and all possible first events for each role in the sequence. Dually, $\mathcal{S}_2^{\texttt{ft}} \lfloor;\rfloor \mathcal{S}_1^{\texttt{ft}}$, ensures all last transmission and events with respect to the considered sequence will be captured by the newly derived summary.

One of the key points of this phase is captured by the merge of adjacent boundaries, $\mathcal{S}_1^{\texttt{ft}}$ and $\mathcal{S}_2^{\texttt{bt}}$, respectively, merge which generates a set of assumptions, $\texttt{A}_3^{\oplus}$, and a set of proof obligations,

$\mathtt{A}_3^{\ominus}$. The assumptions capture the happens-before relation between adjacent events, namely the last event of $\mathtt{G}_1$ and the first event of $\mathtt{G}_2$ with respect to a particular role. Similarly, the guards capture the race-free conditions for all the adjacent transmissions sharing a common channel. Formally this merge is defined by the `merge_adjacent` function:

$$\begin{aligned}\mathtt{merge\_adjacent}(\langle \mathtt{K}_1, \Gamma_1\rangle, \langle \mathtt{K}_2, \Gamma_2\rangle) &\stackrel{\text{def}}{=} \langle \mathtt{K}_1 \between \mathtt{K}_2, \Gamma_1 \between \Gamma_2\rangle \\ \text{where } \mathtt{Map}_1 \between \mathtt{Map}_2 &= \bigcup_{\mathtt{Key}\in\mathtt{Keys}} \mathtt{merge}(\mathtt{Map}_1(\mathtt{Key}), \mathtt{Map}_2(\mathtt{Key})) \\ \text{and} \qquad \mathtt{Keys} &= \mathtt{dom}(\mathtt{Map}_1) \cup \mathtt{dom}(\mathtt{Map}_2).\end{aligned}$$

In the following, the recursive function `merge` is overloaded such that it can cater to both event merging (first base-case definition) and transmission merging (second base-case definition):

$$\begin{aligned}\mathtt{merge}(\beta_1 * \beta_2,\, \beta) &\stackrel{\text{def}}{=} \mathtt{merge}(\beta_1,\, \beta) \cup \mathtt{merge}(\beta_2,\, \beta) \\ \mathtt{merge}(\beta,\, \beta_1 * \beta_2) &\stackrel{\text{def}}{=} \mathtt{merge}(\beta,\, \beta_1) \cup \mathtt{merge}(\beta,\, \beta_2) \\ \mathtt{merge}(\beta_1 \vee \beta_2,\, \beta) &\stackrel{\text{def}}{=} \mathtt{merge}(\beta_1,\, \beta) \cup \mathtt{merge}(\beta_2,\, \beta) \\ \mathtt{merge}(\beta,\, \beta_1 \vee \beta_2) &\stackrel{\text{def}}{=} \mathtt{merge}(\beta,\, \beta_1) \cup \mathtt{merge}(\beta,\, \beta_2) \\ \mathtt{merge}(\mathtt{P}^{(i_1)},\, \mathtt{P}^{(i_2)}) &\stackrel{\text{def}}{=} \oplus(\mathtt{P}^{(i_1)} \prec_{\mathtt{HB}} \mathtt{P}^{(i_2)}) \\ \mathtt{merge}(i_1,\, i_2) &\stackrel{\text{def}}{=} \ominus(i_1 \prec_{\mathtt{HB}} i_2)\end{aligned}$$

4.3 Local Projection

Based on the communication interface, but also on the verifier's requirements, the projection of the global protocol to local specifications could go through a couple of automatic projection phases before being used by the verification process. This way, the projection could describe how each party is contributing to the communication, or it could be more granular describing how each communication instrument is used with respect to a communicating party.

Projection Language. Fig. 4.5 describes the two kinds of specification mentioned above. The per party specification language is depicted in Fig. 4.5a. Here, each send and receive specification name the communication instrument `c` along with a message `v` described by a formula Δ. In the per channel specifications, Fig. 4.5b, the communication instrument is implicit. The congruence of all the compound terms described in Sec. 4.2 holds for the projected languages as well, with the exception of sequential commutativity since the disjointness conditions for

Local protocol	Υ ::=	L ::=	Z ::=
Send	$\mathtt{c}^i!\mathtt{v}\cdot\Delta$	$^i!\mathtt{v}\cdot\Delta$	
Receive	$\mid \mathtt{c}^i?\mathtt{v}\cdot\Delta$	$\mid {}^i?\mathtt{v}\cdot\Delta$	
Transm.			$\mathtt{P}_1 \xrightarrow{i} \mathtt{P}_2 : \mathtt{v}\cdot\Delta$
HO variable	$\mid \mathtt{V}$	$\mid \mathtt{V}$	
Concurrency	$\mid \Upsilon * \Upsilon$		
Choice	$\mid \Upsilon \vee \Upsilon$	$\mid \mathtt{L} \vee \mathtt{L}$	$\mid \mathtt{Z} \vee \mathtt{Z}$
Sequence	$\mid \Upsilon;\Upsilon$	$\mid \mathtt{L};\mathtt{L}$	$\mid \mathtt{Z};\mathtt{Z}$
Guard	$\mid \ominus(\Psi)$	$\mid \ominus(\Psi)$	$\mid \ominus(\Psi)$
Assumption	$\mid \oplus(\Psi)$	$\mid \oplus(\Psi)$	$\mid \oplus(\Psi)$
	(a) Per party	(b) Per endpoint	(c) Per channel

Figure 4.5: The Projection Language

the latter do not hold (e.g. either the peer or the channel are implicitly the same for the entire projected specification). For brevity, we will often omit to explicitly mention the unique label identifier of transmission events $\mathtt{c}^i!\mathtt{v}\cdot\Delta$, $\mathtt{c}^i?\mathtt{v}\cdot\Delta$, $^i!\mathtt{v}\cdot\Delta$ and $^i?\mathtt{v}\cdot\Delta$.

Automatic projection. Using different projection granularities should not permit event reorderings (modulo $*$ composed events).

Proposition 1 (Projection Fidelity)**.** *The projection to a decomposed specification, such as global protocol to per party, or per party to per channel, does not alter the communication pattern specified before projection.*

To support the above proposition, we have designed a set of projection-rules, described in Fig. 4.6. We support three different projection granularities: (a) the per-party projection which offers a local view of the communication w.r.t. each communicating party, (b) the per-endpoint projection to capture the specification of each communicating party w.r.t. a specific channel (also called logical endpoint from now on), and finally (c) a per-channel projection which captures the communication w.r.t. a single channel across multiple parties. The last projection rules are only needed in designing the semantics of the target language and they are being used to model the communication channels. Therefore we postpone further discussion about Fig. 4.6c to Chapter 6, and focus on the first two kinds of projections.

Fig. 4.6a depicting the per party projection rules is quite self-explanatory, with the exception of the last rule, that of the guard projection. This rule introduces one important aspect of collaborative proving, namely that a happens-before relation between two events, say $\mathtt{P}_1^{(i_1)} \prec_{\mathtt{HB}} \mathtt{P}_2^{(i_2)}$, needs to be proved to hold by the peer which triggers the later event, $\mathtt{P}_2$ in this case, but can

safely be assumed to hold by all the other peers which rely on $\mathtt{P}_2$ to perform the proving, hence the projected guard order versus the assumed one. Take note that the peers which assume the said happens-before relation may use this assumption at any program point, irrespective of the proving outcome. This is sound because if the peer which needs to prove the guard succeeds to do so, then the assumption was correct and it does not lead to any bogus outcome. However, should the peer which needs to prove the guard fail to do so, then the verification of the corresponding thread fails, and by the proof rule of parallel composition the whole program verification fails as well, and therefore the effect of making a wrong assumption is cancelled by this global verification fail.

As expected, the per endpoint projection rules, Fig. 4.6b, strips the channel information from the per party specifications, since it will be implicitly available. Furthermore, inserting a guard $\ominus(\mathtt{P}^{(i)})$ between adjacent transmissions on different channels with a common sender ensures that the order of events at the sender's site is accurately inherited from the corresponding per party specification across different channels. To emphasize this behavior we consider the following sequence of receiving events captured by a per-party specification, $(\mathtt{G})\!\upharpoonright_{\mathtt{P}}$:

$$
\begin{array}{l}
(\mathtt{G})\!\upharpoonright_{\mathtt{P}} : \quad \mathtt{c}_1?\mathtt{v}\cdot\Delta_1 \;;\oplus(\mathtt{P}^{(1)});\quad \mathtt{c}_2?\mathtt{v}\cdot\Delta_2;\quad \oplus(\mathtt{P}^{(2)});\, \mathtt{c}_2?\mathtt{v}\cdot\Delta_3;\oplus(\mathtt{P}^{(3)});\quad \mathtt{c}_1?\mathtt{v}\cdot\Delta_4 \;\ldots \\
\hline
(\mathtt{G})\!\upharpoonright_{\mathtt{P},\mathtt{c}_1} : \mathtt{c}_1?\mathtt{v}\cdot\Delta_1 \;;\oplus(\mathtt{P}^{(1)});\qquad\qquad\quad ;\ominus(\mathtt{P}^{(2)});\qquad\quad ;\ominus(\mathtt{P}^{(3)});\quad \boxed{\mathtt{c}_1?\mathtt{v}\cdot\Delta_4}\ldots \\
(\mathtt{G})\!\upharpoonright_{\mathtt{P},\mathtt{c}_2} : \qquad\qquad ;\ominus(\mathtt{P}^{(1)});\quad \boxed{\mathtt{c}_2?\mathtt{v}\cdot\Delta_2} ;\oplus(\mathtt{P}^{(2)});\mathtt{c}_2?\mathtt{v}\cdot\Delta_3;\oplus(\mathtt{P}^{(3)});\qquad\qquad \ldots
\end{array}
$$

The above local specification snapshot highlights how local fidelity is secured: the events marked with red boxes are guarded by their immediately preceding events, since they are handled by different channels. A subsequent refinement removes redundant guards, grayed in the example above, since adjacent same channel events need to guard only the last event on the considered channel.

Given the congruence of global protocols and local specifications, the projection is an isomorphism due to the unique labelling and ordering relations carefully inserted after each transmission. Given two protocols $\mathtt{G}_1$ and $\mathtt{G}_2$, with $\mathtt{P}_1..\mathtt{P}_n \in \mathtt{G}_1$ and $\neg(\exists \mathtt{P} \in \mathtt{G}_2 \cdot \mathtt{P} \notin \{\mathtt{P}_1..\mathtt{P}_n\})$, and $\mathtt{c}_1..\mathtt{c}_m \in \mathtt{G}_1$ and $\neg(\exists \mathtt{c} \in \mathtt{G}_2 \cdot \mathtt{c} \notin \{\mathtt{c}_1..\mathtt{c}_m\})$:

$$\mathtt{G}_1 \equiv \mathtt{G}_2 \;\Leftrightarrow\; (\{(\mathtt{G}_1)\!\upharpoonright_{\mathtt{P}_j}\}_{\mathtt{j}=1..\mathtt{n}} \;\equiv\; \{(\mathtt{G}_2)\!\upharpoonright_{\mathtt{P}_j}\}_{\mathtt{j}=1..\mathtt{n}}).$$

This condition holds even for the more granular specifications:

$$\mathtt{G}_1 \equiv \mathtt{G}_2 \;\Leftrightarrow\; (\{(\mathtt{G}_1)\!\upharpoonright_{\mathtt{P}_j,\mathtt{c}_k}\}_{\mathtt{j}=1..\mathtt{n},\mathtt{k}=1..\mathtt{m}} \;\equiv\; \{(\mathtt{G}_2)\!\upharpoonright_{\mathtt{P}_j,\mathtt{c}_k}\}_{\mathtt{j}=1..\mathtt{n},\mathtt{k}=1..\mathtt{m}}).$$

$$
\begin{array}{lcl}
(S \xrightarrow{i} \mathtt{R} : \mathtt{c}\langle \Delta \rangle)\lfloor_{\mathtt{P}} & := & \begin{cases} \mathtt{c}^{i}!\mathtt{v} \cdot \Delta & \text{if } \mathtt{P}{=}S \\ \mathtt{c}^{i}?\mathtt{v} \cdot \Delta & \text{if } \mathtt{P}{=}\mathtt{R} \\ \mathtt{emp} & \text{otherwise} \end{cases} \\
(\mathtt{G}_1 * \mathtt{G}_2)\lfloor_{\mathtt{P}} & := & (\mathtt{G}_1)\lfloor_{\mathtt{P}} * (\mathtt{G}_2)\lfloor_{\mathtt{P}} \\
(\mathtt{G}_1 \vee \mathtt{G}_2)\lfloor_{\mathtt{P}} & := & (\mathtt{G}_1)\lfloor_{\mathtt{P}} \vee (\mathtt{G}_2)\lfloor_{\mathtt{P}} \\
(\mathtt{G}_1 ; \mathtt{G}_2)\lfloor_{\mathtt{P}} & := & (\mathtt{G}_1)\lfloor_{\mathtt{P}} ; (\mathtt{G}_2)\lfloor_{\mathtt{P}} \\
(\oplus(\mathtt{P}_1^{(i)}))\lfloor_{\mathtt{P}} & := & \begin{cases} \oplus(\mathtt{P}^{(i)}) & \text{if } \mathtt{P}{=}\mathtt{P}_1 \\ \mathtt{emp} & \text{otherwise} \end{cases} \\
(\ominus(\mathtt{P}_1^{(i_1)} \prec_{\mathtt{HB}} \mathtt{P}_2^{(i_2)}))\lfloor_{\mathtt{P}} & := & \begin{cases} \ominus(\mathtt{P}_1^{(i_1)} \prec_{\mathtt{HB}} \mathtt{P}_2^{(i_2)}) & \text{if } \mathtt{P}{=}\mathtt{P}_2 \\ \oplus(\mathtt{P}_1^{(i_1)} \prec_{\mathtt{HB}} \mathtt{P}_2^{(i_2)}) & \text{otherwise} \end{cases}
\end{array}
$$

(a) global spec → per party spec

$$
\begin{array}{lcl}
({\mathtt{c}_1}^{i}!\mathtt{v} \cdot \Delta)\lfloor_{\mathtt{c}} & := & \begin{cases} {}^{i}!\mathtt{v} \cdot \Delta & \text{if } \mathtt{c}{=}\mathtt{c}_1 \\ \mathtt{emp} & \text{otherwise} \end{cases} \\
({\mathtt{c}_1}^{i}?\mathtt{v} \cdot \Delta)\lfloor_{\mathtt{c}} & := & \begin{cases} {}^{i}?\mathtt{v} \cdot \Delta & \text{if } \mathtt{c}{=}\mathtt{c}_1 \\ \mathtt{emp} & \text{otherwise} \end{cases} \\
(\Upsilon_1 * \Upsilon_2)\lfloor_{\mathtt{c}} & := & \begin{cases} (\Upsilon_{\mathtt{j}})\lfloor_{\mathtt{c}} & \text{if } \mathtt{c} \in \Upsilon_{\mathtt{j}}, \mathtt{j}{=}1 \text{ or } 2 \\ \mathtt{emp} & \text{otherwise} \end{cases} \\
(\Upsilon_1 \vee \Upsilon_2)\lfloor_{\mathtt{c}} & := & (\Upsilon_1)\lfloor_{\mathtt{c}} \vee (\Upsilon_2)\lfloor_{\mathtt{c}} \\
(\Upsilon_1 ; \Upsilon_2)\lfloor_{\mathtt{c}} & := & (\Upsilon_1)\lfloor_{\mathtt{c}} ; (\Upsilon_2)\lfloor_{\mathtt{c}} \\
(\oplus(\mathtt{P}^{(i)}))\lfloor_{\mathtt{c}} & := & \begin{cases} \oplus(\mathtt{P}^{(i)}) & \text{if } \mathtt{c} \in i \\ \ominus(\mathtt{P}^{(i)}) & \text{otherwise} \end{cases} \\
(\ominus(\mathtt{P}_1^{(i_1)} \prec_{\mathtt{HB}} \mathtt{P}_2^{(i_2)}))\lfloor_{\mathtt{c}} & := & \begin{cases} \ominus(\mathtt{P}_1^{(i_1)} \prec_{\mathtt{HB}} \mathtt{P}_2^{(i_2)}) & \text{if } \mathtt{c} \in i_2 \\ \mathtt{emp} & \text{otherwise} \end{cases} \\
(\oplus(\mathtt{P}_1^{(i_1)} \prec_{\mathtt{HB}} \mathtt{P}_2^{(i_2)}))\lfloor_{\mathtt{c}} & := & \begin{cases} \oplus(\mathtt{P}_1^{(i_1)} \prec_{\mathtt{HB}} \mathtt{P}_2^{(i_2)}) & \text{if } \mathtt{c} \in i_2 \\ \mathtt{emp} & \text{otherwise} \end{cases}
\end{array}
$$

(b) per party spec → per endpoint spec

$$
\begin{array}{lcl}
(S \xrightarrow{i} \mathtt{R} : \mathtt{c}\langle \mathtt{v} \cdot \Delta \rangle)\lfloor_{\mathtt{c}} & := & S \xrightarrow{i} \mathtt{R} : \mathtt{v} \cdot \Delta \\
(\mathtt{G}_1 * \mathtt{G}_2)\lfloor_{\mathtt{c}} & := & \begin{cases} (\mathtt{G}_1)\lfloor_{\mathtt{c}} & \text{if } \mathtt{c} \in \mathtt{G}_1 \\ (\mathtt{G}_2)\lfloor_{\mathtt{c}} & \text{if } \mathtt{c} \in \mathtt{G}_2 \\ \mathtt{emp} & \text{otherwise} \end{cases} \\
(\mathtt{G}_1 \vee \mathtt{G}_2)\lfloor_{\mathtt{c}} & := & (\mathtt{G}_1)\lfloor_{\mathtt{c}} \vee (\mathtt{G}_2)\lfloor_{\mathtt{c}} \\
(\mathtt{G}_1 ; \mathtt{G}_2)\lfloor_{\mathtt{c}} & := & (\mathtt{G}_1)\lfloor_{\mathtt{c}} ; (\mathtt{G}_2)\lfloor_{\mathtt{c}} \\
(\oplus(\mathtt{P}_1^{(i)}))\lfloor_{\mathtt{c}} & := & \begin{cases} \oplus(\mathtt{P}^{(i)}) & \text{if } \mathtt{c} \in i \\ \ominus(\mathtt{P}^{(i)}) & \text{if } \mathtt{c} \notin i \end{cases} \\
(\ominus(\mathtt{P}_1^{(i_1)} \prec_{\mathtt{HB}} \mathtt{P}_2^{(i_2)}))\lfloor_{\mathtt{c}} & := & \begin{cases} \ominus(\mathtt{P}_1^{(i_1)} \prec_{\mathtt{HB}} \mathtt{P}_2^{(i_2)}) & \text{if } \mathtt{c} \in i_2 \\ \mathtt{emp} & \text{otherwise} \end{cases}
\end{array}
$$

(c) global spec → per channel spec

$$
(\oplus(\mathtt{P}_1^{(i)} \prec_{\mathtt{CB}} \mathtt{P}_2^{(i)}))\lfloor_{\mathtt{All}} := \oplus(\mathtt{P}_1^{(i)} \prec_{\mathtt{CB}} \mathtt{P}_2^{(i)}) \qquad (\oplus(\mathtt{P}_1^{(i_1)} \prec_{\mathtt{HB}} \mathtt{P}_2^{(i_2)}))\lfloor_{\mathtt{All}} := \oplus(\mathtt{P}_1^{(i_1)} \prec_{\mathtt{HB}} \mathtt{P}_2^{(i_2)})
$$

(d) global spec → shared spec

Figure 4.6: Projection Rules

And lastly, as discussed in Sec. 4.1, the communicated-before and happened-before assumptions are projected into a shared store, so that each party can benefit from them (Fig. 4.6d).

4.4 Nondeterminism

The logic proposed this far allows for the communication designer to express concurrent communication via the parallel composition $*$ as depicted in Fig. 4.1. The support for concurrent protocols allows: (i) for a more relaxed communication implementation where the order of transmissions is not strict, (ii) for a more performant underlying implementation via a finer concurrency. However, the usage of $*$ in designing concurrent protocols is restricted to the well-formedness property of Def. 4 which forbids concurrent channel usages in order to avoid possible nondeterminism. This section argues however, that nondeterminism is not only desirable, but also required in some specific communication-based applications, such as distributed computation where the distribution of a computation or the collection of the computation result need not follow a specific order, yet it is consumed on the same channel.

For example, let us consider the following data structure `tnode` used for building a binary tree and its logical description `tree`:

```
struct tnode { struct tnode *left; struct tnode *right; };
```

$$\texttt{tree}(\texttt{root}) \triangleq \texttt{emp} \wedge \texttt{root} = \texttt{null} \ \vee \ \exists\, \texttt{l}, \texttt{r} \cdot \texttt{root} \mapsto \texttt{tnode}\langle \texttt{l}, \texttt{r} \rangle * \texttt{tree}(\texttt{l}) * \texttt{tree}(\texttt{r}).$$

Let us also assume that a binary tree of a considerable size needs to be traversed to update each of its nodes according to some compute-intensive function `foo`. We also assume that (i) this update is insensitive to the order in which the nodes are modified. Moreover, this special tree (ii) allows for its left and right subtree to be arbitrarily swapped as long as the child-parent structure is maintained. If a party `A` owns the tree it could then engage two helper parties, say $\texttt{H}_1$ and $\texttt{H}_2$ waiting for requests from `A` on the same channel $\texttt{c}_1$, to run `foo` and return the result to `A` over channel $\texttt{c}_2$. A protocol for such a computation could then be described as:

$$\begin{aligned} \texttt{G}_1 \ \triangleq \ & (\texttt{A} \xrightarrow{1} \texttt{H}_1 : \texttt{c}_1\langle \texttt{v}_1 \cdot \texttt{tree}(\texttt{v}_1) \rangle * \texttt{A} \xrightarrow{2} \texttt{H}_2 : \texttt{c}_1\langle \texttt{v}_2 \cdot \texttt{tree}(\texttt{v}_2) \rangle) \ ; \\ & (\texttt{H}_1 \xrightarrow{3} \texttt{A} : \texttt{c}_2\langle \texttt{v}_3 \cdot \texttt{tree}(\texttt{v}_3) \rangle * \texttt{H}_2 \xrightarrow{4} \texttt{A} : \texttt{c}_2\langle \texttt{v}_4 \cdot \texttt{tree}(\texttt{v}_4) \rangle). \end{aligned}$$

and a corresponding implementation might be shaped as:

A	H_1	H_2
...	...	...
send(c1,x->left);	datat *t1,*t2;	datat *t1,*t2;
send(c1,x->right);	t1 = receive(c1);	t1 = receive(c1);
x->left = receive(c2);	t2 = foo(t1);	t2 = foo(t1);
x->right = receive(c2);	send(c2,t2);	send(c2,t2);

It is worth noting some clear and some more salient points of such a communication configuration. First, the protocol allows two race conditions which are made explicit in the implementation: one when A is sending the subtrees references towards H_1 and H_2, and the second race condition is when A receives the result of the computations. However, this kind of race should be acceptable since, from A's perspective, distinguishing between the two helpers is irrelevant - as per the tree's property (ii). Moreover, the order in which the references of the two updated subtrees are received is not important according to assumption (i) stated earlier.

Next, it might be easily observed that such a configuration is only acceptable provided that the messages which A sends are abstractly the same, and so are the received messages. A formal definition of abstractly similar messages could be translated into:

Definition 11 (Indistinguishable Messages)**.** *Given a global protocol* G*, two messages captured by this protocol, say* $v_1 \cdot \Delta_1$ *and* $v_2 \cdot \Delta_2$*, are said to be abstractly indistinguishable, denoted by* $v_1 \cdot \Delta_1 \dashv\vdash v_2 \cdot \Delta_2$*, when the following relation holds:*

$$\frac{\Delta_1 \vdash^{\mathtt{emp}}_{\{\}} [v_1/v_2]\Delta_2 \qquad \Delta_2 \vdash^{\mathtt{emp}}_{\{\}} [v_2/v_1]\Delta_1}{v_1 \cdot \Delta_1 \dashv\vdash v_2 \cdot \Delta_2}$$

Communication configurations such as the one exemplified above and its corresponding protocol fail to meet the well-formedness condition of deterministic protocols and deterministic communication in general. However, there are a number of such scenarios in the area of distributed computation where nondeterminism is useful in gaining computation performance: distributed rendering in computer graphics, computational biology, statistics, big data processing, and so on. In the subsequent we relax the definition of well-formed concurrency to also cope with nondeterminism.

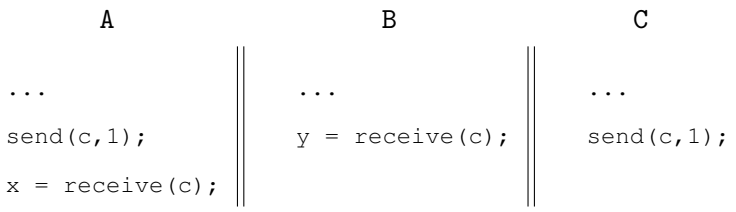
Definition 12 (Well-Formed Nondeterminism)**.** *A protocol specification* $G_1 * G_2$ *is said to be well-formed with respect to* $*$ *if and only if (i) all the messages transmitted over shared channels are abstractly indistinguishable and moreover, (ii) the specification* $G_1 * G_2$ *contains no explicit*

or implicit self-transmissions. The well-formedness is denoted by $\texttt{well-formed}(\texttt{G}_1 * \texttt{G}_2)$ *which is recursively defined in Fig. 4.7.*

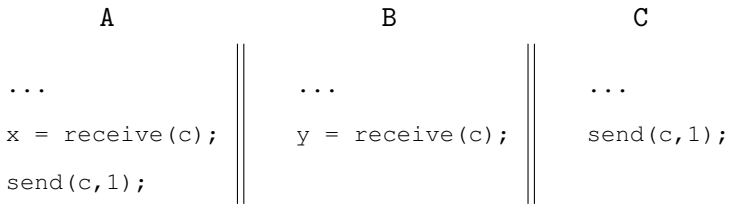
The second constraint of the well-formed nondeterminism refers to those transmissions which are reflexive or might lead to implicit self-communication scenarios. Reflexive transmissions are, however, disallowed in the current multiparty communication logic, as described in Sec. 4.2.1, therefore a protocol contains only irreflexive transmissions. As for the implicit self-transmissions, they are avoided by requiring for the two protocols composed by $*$ to be compatible, or, as formally described in Fig. 4.7, the following relation should hold: $\texttt{G}_1 \bowtie_{\texttt{p}} \texttt{G}_2$. To get a better feeling of why this constraint is enforced for well-formed concurrency let us consider the following nondeterministic communication protocol:

$$\texttt{G}_2 \triangleq \texttt{A} \xrightarrow{1} \texttt{B} : \texttt{c}\langle \texttt{v} \cdot \Delta \rangle * \texttt{C} \xrightarrow{2} \texttt{A} : \texttt{c}\langle \texttt{v} \cdot \Delta \rangle$$

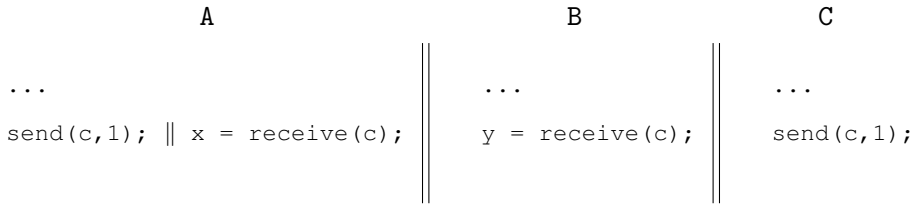
Implementation variations for protocol $\texttt{G}_2$ might include the following two configurations:



(a) Implementation 1



(b) Implementation 2



(c) Implementation 3

Even if all three of the above are seemingly legitimate implementations of $\texttt{G}_2$, with A and C transmitting 1 to B and A, respectively, a more careful analysis of Implementation 2 - (b) - uncovers a potential deadlock: B and A are both waiting to receive a message from c. Should the receive in B be consumed prior to that in A (we stress that data-race freedom is not enforced

$$\frac{\forall i_1 \in \mathtt{G}_1, i_2 \in \mathtt{G}_2 \Rightarrow \mathtt{compatibT}(i_1, i_2)}{\mathtt{compatibP}(\mathtt{G}_1, \mathtt{G}_2)} \qquad \frac{\mathtt{chan}(i_1){=}\mathtt{chan}(i_2) \Rightarrow \mathtt{msg}(i_1)\dashv\vdash\mathtt{msg}(i_2)}{\mathtt{compatibT}(i_1, i_2)}$$

$$\frac{\mathtt{compatibP}(\mathtt{G}_1, \mathtt{G}_2) \quad \mathtt{compatibP}(\mathtt{G}_1, \mathtt{G}_1) \quad \mathtt{compatibP}(\mathtt{G}_2, \mathtt{G}_2)}{\mathtt{compatib}(\mathtt{G}_1, \mathtt{G}_2)}$$

$$\frac{\mathtt{chan}(i_1){=}\mathtt{chan}(i_2) \Rightarrow \neg(\mathtt{send}(i_1){=}\mathtt{recv}(i_2)) \wedge \neg(\mathtt{recv}(i_1){=}\mathtt{send}(i_2))}{i_1 \bowtie i_2}$$

$$\frac{\forall i_1 \in \mathtt{G}_1, i_2 \in \mathtt{G}_2 \Rightarrow i_1 \bowtie i_2}{\mathtt{G}_1 \bowtie_{\mathtt{p}} \mathtt{G}_2}$$

$$\frac{\mathtt{well\text{-}formed}(\mathtt{G}')}{\mathtt{well\text{-}formed}(\mathtt{S} \xrightarrow{i} \mathtt{R} : \mathtt{c}\langle \mathtt{v} \cdot \Delta \rangle; \mathtt{G}')} \qquad \frac{\mathtt{well\text{-}formed}(\mathtt{G}_1) \quad \mathtt{well\text{-}formed}(\mathtt{G}_2)}{\mathtt{well\text{-}formed}(\mathtt{G}_1 \vee \mathtt{G}_2)}$$

$$\frac{\mathtt{G}_1 \bowtie_{\mathtt{p}} \mathtt{G}_2 \quad \mathtt{compatib}(\mathtt{G}_1, \mathtt{G}_2) \quad \mathtt{well\text{-}formed}(\mathtt{G}_1) \quad \mathtt{well\text{-}formed}(\mathtt{G}_2)}{\mathtt{well\text{-}formed}(\mathtt{G}_1 * \mathtt{G}_2)}$$

Figure 4.7: Well-Formed Concurrency with Nondeterminism

for transmissions composed with $*$ since it would defeat the purpose of allowing concurrent composition of protocols in the first place), it then leads to a self-communication scenario where A remains blocked since its corresponding send is scheduled in the program code to occur after the blocking receive. To avoid such situations it is preferred to disallow direct or indirect self-transmissions like the one exemplified above, hence the (ii) condition in the definition of well-formed nondeterministic protocols. This condition is formally described via the binary relation between two protocols:

Proposition 2. *The parallel composition of protocol* $\mathtt{G}_1$ *with protocol* $\mathtt{G}_2$ *forms no implicit self-transmission if* $\mathtt{G}_1 \bowtie_{\mathtt{p}} \mathtt{G}_2$ *holds.*

Proof: The proof is straightforward using the definition of $\bowtie_{\mathtt{p}}$ in Fig. 4.7 and it relies on the fact that the set of senders and that of receivers of all the transmissions sharing the same channel are disjoint. □

A more relaxed nondeterminism. A further relaxation of the nondeterminism well-formedness condidtion is possible and even required by certain communication scenarios where a higher level of precision reasoning is required.

Suppose a nondeterministic communication protocol like below:

$$\mathtt{G}_3 \triangleq \mathtt{B} \xrightarrow{1} \mathtt{A} : \mathtt{c}\langle \mathtt{v} \cdot 0 \leq \mathtt{v} < 10 \rangle * \mathtt{C} \xrightarrow{2} \mathtt{A} : \mathtt{c}\langle \mathtt{v} \cdot 10 \leq \mathtt{v} \leq 100 \rangle.$$

with a possible implementation of party `A` as follows:

```
1    x = receive(c);
2    y = receive(c);
3    z = x + y;
4    assert 10≤z<110;
```

Despite of the nondeterminism of the receiving party, where `x` and `y` could store the values received from `B` and `C`, or vice-versa, the assertion at line `4` should still hold regardless of the message receiving order since the operation used over the received messages is commutative.

That being said, it should be possible to capture concurrent transmissions over a shared channel whose messages are in fact described by abstractly different formulas. Constraint (i) of well-formed nondeterminism in Def. 12 could be the relaxed to allow distinct messages, with the proviso that it should not break type safety:

Definition 13 (Well-Formed Nondeterminism - 2)**.** *A protocol specification* $\mathtt{G}_1 * \mathtt{G}_2$ *is said to be well-formed with respect to* $*$ *if and only if (i) all the messages transmitted over shared channels have the same type and moreover, (ii) the specification* $\mathtt{G}_1 * \mathtt{G}_2$ *contains no explicit or implicit self-transmissions.*

Though more expressive than previously formalized protocols, this kind of nondeterminisim however, would fit better in a framework with possibilities for commutative properties checking [51]. Commutative properties are beyond the scope of this thesis, therefore the proof system described in the next chapter assumes only well-formed nondeterminism as per Def. 12.

4.5 Variations of the Communication Model

We chose to illustrate the multiparty session logic proposed in this thesis in the context of an asynchronous communication model with shared channels. On the one hand, this choice is motivated by the widely usage of asynchronous communication in real world applications. On the other hand, sharing channels among multiple participants poses significant challenges [27, 47, 35, 88] in providing communication safety and correctness guarantees.

But the currently proposed session logic is not limited to just asynchronous communication with shared channel, that is the communication model described in Sec. 2.6. Having to reason on top of a *synchronous* communication instead of an asynchronous one, simply involves

adapting the constraint propagation rules in Fig. 4.3b to describe a synchronous behavior. The propagation rules introduced so far characterize the program order of events - [HB-HB] - and the blocking behavior of the receive operations - [CB-HB]. To also capture for blocking send operations we introduce the following propagation rule:

$$\mathtt{E}_1 \prec_{\mathtt{CB}} \mathtt{E}_2 \wedge \mathtt{E}_1 \prec_{\mathtt{HB}} \mathtt{E}_3 \;\Rightarrow\; \mathtt{E}_2 \prec_{\mathtt{HB}} \mathtt{E}_3 \quad \text{[sync-CB-HB]}$$

Note that the semantic of the assertions does not change, and neither does the protocol refinement nor the local projection rules.

Let us look at a simple protocol to illustrate the difference in behavior between the two communication models, where for brevity we omit the message information:

$$\mathtt{G}_1 \triangleq \mathtt{A} \xrightarrow{1} \mathtt{B} : \mathtt{c}; \mathtt{B} \xrightarrow{2} \mathtt{A} : \mathtt{c}.$$

refined as below:

$$\begin{aligned}\mathtt{G}_1 \triangleq\; & \mathtt{A} \xrightarrow{1} \mathtt{B} : \mathtt{c}; \oplus(\mathtt{A}^{(1)}); \oplus(\mathtt{B}^{(1)}); \oplus(\mathtt{A}^{(1)} \prec_{\mathtt{CB}} \mathtt{B}^{(1)}); \\ & \mathtt{B} \xrightarrow{2} \mathtt{A} : \mathtt{c}; \oplus(\mathtt{B}^{(2)}); \oplus(\mathtt{A}^{(2)}); \oplus(\mathtt{B}^{(2)} \prec_{\mathtt{CB}} \mathtt{A}^{(2)}); \oplus(\mathtt{A}^{(1)} \prec_{\mathtt{HB}} \mathtt{A}^{(2)}); \oplus(\mathtt{B}^{(1)} \prec_{\mathtt{HB}} \mathtt{B}^{(2)}); \\ & \ominus(\mathtt{A}^{(1)} \prec_{\mathtt{HB}} \mathtt{B}^{(2)}); \ominus(\mathtt{B}^{(1)} \prec_{\mathtt{HB}} \mathtt{A}^{(2)}).\end{aligned}$$

Without some form of explicit synchronization in the actual implementation of $\mathtt{G}_1$, the guard $\ominus(\mathtt{B}^{(1)} \prec_{\mathtt{HB}} \mathtt{A}^{(2)})$ can never be proved to hold in an asynchronous communication, that is using just [HB-HB] and [CB-HB] propagation rules:

$$\Pi \vDash_{\mathtt{RF}} \mathtt{A}^{(1)} \prec_{\mathtt{HB}} \mathtt{B}^{(2)}.$$
$$\Pi \nvDash_{\mathtt{RF}} \mathtt{B}^{(1)} \prec_{\mathtt{HB}} \mathtt{A}^{(2)}.$$

where $\Pi = \{\mathtt{A}^{(1)} \prec_{\mathtt{CB}} \mathtt{B}^{(1)}, \mathtt{B}^{(2)} \prec_{\mathtt{CB}} \mathtt{A}^{(2)}, \mathtt{A}^{(1)} \prec_{\mathtt{HB}} \mathtt{A}^{(2)}, \mathtt{B}^{(1)} \prec_{\mathtt{HB}} \mathtt{B}^{(2)}\}$. The fact that one of the guards fails to hold, correctly warns over a possible communication race in an asynchronous setting.

If we assume a synchronous communication and allow the [sync-CB-HB] to be fired where applicable, it would lead to:

$$\frac{\dfrac{\mathtt{true}}{\Pi \cup \{\mathtt{B}^{(1)} \prec_{\mathtt{HB}} \mathtt{A}^{(2)}\} \vDash_{\mathtt{RF}} \mathtt{B}^{(1)} \prec_{\mathtt{HB}} \mathtt{A}^{(2)}}}{\Pi \vDash_{\mathtt{RF}} \mathtt{B}^{(1)} \prec_{\mathtt{HB}} \mathtt{A}^{(2)}} \text{[sync-CB-HB]}$$

which is obviously correct in a synchronous setting since sending is also blocking and therefore $\mathtt{G}_1$ contains no race.

Another natural variation of the general communication model initially proposed in this dissertation, assumes a linear usage of the communication channels. In this model each channel is shared between exactly two parties. This kind of model poses no challenge w.r.t. race detection. Since no channel is shared, then the refined protocol will contain no guard. The currently proposed logic could cope with such a model without any further extension.

This section informally discussed the fact that the currently proposed logic is not confined to a single communication model, pointing out the ease of adjusting the theory to other models. However, the subsequent chapters which formally introduce the verification of protocols and its semantics continue to assume the more general asynchronous communication model with shared channels, as per Sec. 2.6. Varying the communication model would not affect the verification rules presented next, but it would influence the discussion on deadlock detection and the soundness proof of the present theory. For brevity, from now on, we defer from discussing these issues w.r.t. other communication models.

Chapter 5

AUTOMATIC VERIFICATION OF COMMUNICATING PROGRAMS

5.1 Forward Verification

The user provides the global protocol which is then automatically refined according to the methodology described in Sec. 4.2. The refined protocol is automatically projected onto a per party specification, followed by a per channel endpoint basis as described in Sec. 4.3. Using such a modular approach where we provide a specification for each channel endpoint adds natural support for delegation, where a channel (as well as its specification) could be delegated to a third party. These communication specifications are made available in the program abstract state using a combination of ghost assertions and release lemmas (detailed in the subsequent). The verification could then automatically check whether a certain implementation follows the global protocol, after it had first bound the *program elements* (threads and channel endpoints) to the *logical ones* (peers and channels).

Language. Analogous to the dyadic session logic , we use a similar concurrent programming language with that in Fig. 2.1 to experiment with the proposed multiparty logic. The language support for asynchronous message passing with explicit synchronization is modified as below to tackle the communication specification proposed in Chapter 4:

$$
\begin{array}{lll}
(s,h,o) \vDash \pi & \text{iff} & [\![\pi]\!]_s = \texttt{true} \\
(s,h,o) \vDash \texttt{emp} & \text{iff} & \texttt{dom}(h) = \emptyset \\
(s,h,o) \vDash \texttt{emp}_{\texttt{sh}} & \text{iff} & \texttt{dom}(h) = \emptyset \wedge \texttt{dom}(s) = \emptyset \\
(s,h,o) \vDash \texttt{v} \mapsto \texttt{d}\langle \texttt{v}_1, .., \texttt{v}_n \rangle & \text{iff} & \texttt{struct d}\{\texttt{t}_1\ \texttt{f}_1; ..; \texttt{t}_n\ \texttt{f}_n\} \in \mathcal{P} \text{ and} \\
 & & \quad h = [s(\texttt{v}) \mapsto \texttt{d}[\texttt{f}_1 \mapsto s(\texttt{v}_1), .., \texttt{f}_n \mapsto s(\texttt{v}_n)]] \\
(s,h,o) \vDash p(\texttt{v}^*) & \text{iff} & p(\texttt{w}^*) \equiv \Phi \in \mathcal{P} \text{ and } (s,h,o) \vDash [\texttt{v}^*/\texttt{w}^*]\Phi \\
(s,h,o) \vDash \kappa_1 * \kappa_2 & \text{iff} & \exists\, h_1, h_2 \cdot h = h_1 \uplus h_2 \text{ and } (s,h_1,o) \vDash \kappa_1 \text{ and } (s,h_2,o) \vDash \kappa_2 \\
(s,h,o) \vDash \exists\, \texttt{v}^* \cdot \kappa \wedge \pi \wedge \Psi & \text{iff} & \exists \nu^* \cdot ((s[(\mapsto\nu)^*], h, o) \vDash \kappa \text{ and } (s[(\texttt{v}\mapsto\nu)^*], h, o) \vDash \pi) \\
 & & \quad \text{and } o \vDash_{\texttt{RF}} \Psi \\
(s,h,o) \vDash \Delta_1 \vee \Delta_2 & \text{iff} & (s,h,o) \vDash \Delta_1 \text{ or } (s,h,o) \vDash \Delta_2
\end{array}
$$

$$
\texttt{State} \triangleq \texttt{Stack} \times \texttt{Heap} \times \Pi \qquad \texttt{Stack} \triangleq \texttt{Var} \to \texttt{Val} \cup \texttt{Loc} \qquad \texttt{Heap} \triangleq \texttt{Loc} \rightharpoonup_{\texttt{fin}} \texttt{DVal}
$$

$$
h_1 \perp h_2 \;\Leftrightarrow\; \texttt{dom}(h_1) \cap \texttt{dom}(h_2) = \emptyset
$$

$$
h = h_1 \uplus h_2 \;\Leftrightarrow\; (h_1 \perp h_2) \wedge (\texttt{dom}(h) = \texttt{dom}(h_1) \cup \texttt{dom}(h_2))
$$

$$
\forall \sigma, \sigma_1, \sigma_2 \in \texttt{State} \cdot \sigma = \sigma_1 \uplus \sigma_2 \;\Leftrightarrow\; (\sigma = (s,h,o) \wedge \sigma_1 = (s,h_1,o) \wedge \sigma_2 = (s,h_2,o) \wedge h = h_1 \uplus h_2)
$$

Figure 5.1: The Semantics of the State Assertions

$$
\begin{array}{lll}
\textit{Expressions} & e & ::= \ ... \\
 & & \quad | \ \texttt{open}\ ()\ \texttt{with}\ (\texttt{c}, \{\texttt{P}_1, .., \texttt{P}_n\}) \mid \texttt{close}(\tilde{c}) \\
 & & \quad | \ \texttt{send}(\tilde{c}, \texttt{v}) \mid \texttt{recv}(\tilde{c}) \mid \texttt{notifyAll}(\texttt{E}) \mid \texttt{wait}(\texttt{E})
\end{array}
$$

Concurrent Separation Logics. The assertion semantic model introduced in Sec. 2.2 is enhanced to that of Fig. 5.1. Our approach of layered abstractions permits us to build on top of the traditional storage model for heap manipulating programs with a minimal, yet important extension to account for the race-free assertions. Therefore, we define the program state as the triple comprising a stack $\texttt{s}\in\texttt{Stack}$ which is a total mapping from local and logical variables `Var` to primitive values `Val` or memory locations `Loc`; a heap $\texttt{h}\in\texttt{Heap}$ which is a finite partial mapping from memory locations to data structures stored in the heap, `DVal`; and an ordering relations store $o\in\Pi$ which is a set of assumed events and events relations.

The semantics of the state assertions in Fig. 5.1 are similar to those of Sec. 2.2, with the exception of the conjunction with the order relations which is evaluated in the relations store o. The evaluation of the heap-independent expressions, π, built up from variables and constants is standard, and we therefore leave it unspecified.

Verification. To check whether a user program follows the stipulated communication scenario, a traditional analysis would need to reason about the program's behaviour using the operational

$$[\textbf{OPEN}]$$
$$\vdash \{\ \texttt{init(c)}\ \}\ \texttt{open()}\ \texttt{with}\ (\texttt{c}, \texttt{P}^*)\ \{\ \texttt{opened(c, P}^*\texttt{, res)}\}$$

$$[\textbf{CLOSE}]$$
$$\vdash \{\ \texttt{empty}(\tilde{\texttt{c}})\}\ \texttt{close}(\tilde{\texttt{c}})\ \{\ \texttt{emp}\ \}$$

$$[\textbf{SEND}]$$
$$\frac{\texttt{inv} \triangleq \texttt{Peer(P)} \wedge \texttt{opened(c, P}^*\texttt{,}\ \tilde{\texttt{c}}) \wedge \texttt{P}\in\texttt{P}^*}{\vdash \{\mathcal{C}(\texttt{c, P, !v}\cdot\texttt{V(v);L})*\texttt{V(x)}*\texttt{inv}\}\ \texttt{send}(\tilde{\texttt{c}}, \texttt{x})\ \{\mathcal{C}(\texttt{c, P, L})*\texttt{inv}\}}$$

$$[\textbf{RECV}]$$
$$\frac{\texttt{inv} \triangleq \texttt{Peer(P)} \wedge \texttt{opened(c, P}^*\texttt{,}\ \tilde{\texttt{c}}) \wedge \texttt{P}\in\texttt{P}^*}{\vdash \{\mathcal{C}(\texttt{c, P, ?v}\cdot\texttt{V(v);L})*\texttt{inv}\}\ \texttt{recv}(\tilde{\texttt{c}})\ \{\mathcal{C}(\texttt{c, P, L})*\texttt{V(res)}*\texttt{inv}\}}$$

Figure 5.2: Communication Primitives

$$\begin{array}{lll}
\texttt{G}(\{\texttt{P}_1..\texttt{P}_n\}, \texttt{c}^*) & \mapsto & \texttt{Party}(\texttt{P}_1, \texttt{c}^*, (\texttt{G})\downarrow_{\texttt{P}_1}) * ... * \texttt{Party}(\texttt{P}_n, \texttt{c}^*, (\texttt{G})\downarrow_{\texttt{P}_n}) * \texttt{initall}(\texttt{c}^*). \\
\texttt{Party}(\texttt{P}, \{\texttt{c}_1..\texttt{c}_m\}, (\texttt{G})\downarrow_{\texttt{P}}) & \mapsto & \mathcal{C}(\texttt{c}_1, \texttt{P}, (\texttt{G})\downarrow_{\texttt{P,c}_1}) * ... * \mathcal{C}(\texttt{c}_m, \texttt{P}, (\texttt{G})\downarrow_{\texttt{P,c}_m}) * \texttt{bind}(\texttt{P}, \{\texttt{c}_1..\texttt{c}_m\}). \\
\texttt{initall}(\{\texttt{c}_1..\texttt{c}_m\}) & \mapsto & \texttt{init}(\texttt{c}_1) * ... * \texttt{init}(\texttt{c}_m).
\end{array}$$

(a) Splitting lemmas

$$\begin{array}{lll}
[\textbf{EMP-C}] & \mathcal{C}(\texttt{c}, \texttt{P}_1, \texttt{emp}) * ... * \mathcal{C}(\texttt{c}, \texttt{P}_n, \texttt{emp}) \wedge \texttt{opened}(\texttt{c}, \{\texttt{P}_1..\texttt{P}_n\}, \tilde{\texttt{c}}) \mapsto \texttt{empty}(\tilde{\texttt{c}}). \\
[\textbf{EMP-P}] & \mathcal{C}(\texttt{c}_1, \texttt{P}, \texttt{emp}) * ... * \mathcal{C}(\texttt{c}_m, \texttt{P}, \texttt{emp}) * \texttt{bind}(\texttt{P}, \{\texttt{c}_1..\texttt{c}_m\}) \mapsto \texttt{Party}(\texttt{P}, \texttt{c}^*, \texttt{emp}).
\end{array}$$

(b) Joining lemmas

$$\begin{array}{lll}
[\textbf{L+}] & \mathcal{C}(\texttt{c}, \texttt{P}, \oplus(\Psi); \texttt{L}) & \mapsto \mathcal{C}(\texttt{c}, \texttt{P}, \texttt{L}) \wedge \Psi. \\
[\textbf{L-}] & \mathcal{C}(\texttt{c}, \texttt{P}, \ominus(\Psi); \texttt{L}) \wedge \Psi & \mapsto \mathcal{C}(\texttt{c}, \texttt{P}, \texttt{L}).
\end{array}$$

(c) Lemmas to handle orders

Figure 5.3: Lemmas for Specification Manipulation

semantics of the primitives' implementation. Since our goal is to emphasize on the benefits of implementing a protocol guided communication, rather than deciding the correctness of the primitives machinery, we adopt a specification strategy using abstract predicates [81, 30] to formally describe the behavior of the program's primitives using Hoare triples. Provided that the primitives respect their abstract specification, developers could then choose alternative communication libraries, without the need to re-construct the correctness proof of their underlying program. The proposed abstract specifications are introduced in the subsequent.

Abstract Specification. We define a set of abstract predicates to support session specification of different granularity. Some of these predicates have been progressively introduced across this thesis, but for brevity we have omitted certain details. We resume their presentation here with

more details:

$\texttt{Party}(\texttt{P}, \texttt{c}^*, \Upsilon)$	associates a local protocol projection Υ to its corresponding party P and the set of channels $\texttt{c}^*$ used by P to communicate with its peers;
$\texttt{Peer}(\texttt{P})$	flow-sensitively tracks the executing party, since the execution of parties can either be in parallel or sequentialized;
$\mathcal{C}(\texttt{c}, \texttt{P}, \texttt{L})$	associates an endpoint specification L to its corresponding party P and channel c;
$\texttt{Common}((\texttt{G})\downarrow_{\texttt{All}})$	comprises the ordering assumptions shared among all the parties;
$\texttt{initall}(\texttt{c}^*)/$	hold only when the specifications corresponding to logical channels $\texttt{c}^*/\texttt{c}$
$\texttt{init}(\texttt{c})$	are available (have been released into the abstract program state - Fig. 5.3a);
$\texttt{bind}(\texttt{P}, \texttt{c}^*)$	binds a party P to all the channels $\texttt{c}^*$ it uses;
$\texttt{opened}(\texttt{c}, \texttt{P}^*, \tilde{\texttt{c}})$	binds a program channel $\tilde{\texttt{c}}$ to a logical one c and to the peers sharing $\tilde{\texttt{c}}$;
$\texttt{empty}(\tilde{\texttt{c}})$	holds only when all the transmissions on $\tilde{\texttt{c}}$ have been consumed (Fig. 5.3b).

To cater for each verification phase, the session specifications with the required granularity are made available in the program's abstract state via the splitting lemmas in Fig. 5.3a. The joining rules in Fig. 5.3b restore the granularity of the specification post verification, to indicate that the expected communication is fully consumed.

The channel creation described by the [OPEN] triple in Fig. 5.2, associates a logical channel c to the program channel res returned by open. The user can also specify a set of logical peers $\texttt{P}^*$ sharing the same program channel res. If there are multiple program channels implementing the same logical channel, these should be shared among distinct peers. Also, it is not allowed for the same program channel to implement multiple logical channel, since the race-freedom analysis makes the basic assumption that the logical channels are distinct. If this assumption is broken, then the communication might contain undetected races:

$$\texttt{opened}(\texttt{c}, \texttt{P}_1^*, \tilde{\texttt{c}}_1) * \texttt{opened}(\texttt{c}, \texttt{P}_2^*, \tilde{\texttt{c}}_2) \wedge \texttt{P}_1^* \cap \texttt{P}_2^* \neq \emptyset \wedge \tilde{\texttt{c}}_1 \neq \tilde{\texttt{c}}_2 \;\; \mapsto \texttt{false}$$

$$\texttt{opened}(\texttt{c}_1, \texttt{P}_1^*, \tilde{\texttt{c}}) * \texttt{opened}(\texttt{c}_2, \texttt{P}_2^*, \tilde{\texttt{c}}) \wedge \texttt{c}_1 \neq \texttt{c}_2 \;\; \mapsto \texttt{false}$$

The closing of a channel is described by the [CLOSE] triple which requires for all the endpoints corresponding to the program channel $\tilde{\texttt{c}}$ to have completed their communication share, $\texttt{empty}(\tilde{\texttt{c}})$.

To support send and receive operations, we decorate the corresponding methods with dual

generic specifications. The precondition of [SEND] ensures that indeed a send operation is expected, $!\mathtt{v} \cdot \mathtt{V}(\mathtt{v})$, where the message $\mathtt{v}$ to be transmitted is described by a higher-order relation over $\mathtt{v}$. Should this be confirmed, to ensure memory safety, the verifier also checks whether the program state indeed owns the message to be transmitted and that it adheres to the properties described by the freshly discovered relation, $\mathtt{V}(\mathtt{x})$. Dually, [RECV] ensures that the receiving state gains the ownership of the transmitted message. Both specifications guarantee that the transmission is consumed by the expected party, $\mathtt{Peer}(\mathtt{P})$. The rest of the proof rules are standard [5], we therefore omit their details in this presentation.

The proof obligations generated while checking each expression are discharged to a Separation Logic solver in the form of entailment checks, detailed in the subsequent sub-section.

Entailment. Traditionally, the logical entailment between formulae written in the symbolic heap fragment of separation logic is expressed as follows: $\Delta_{\mathtt{a}} \vdash \Delta_{\mathtt{c}} * \Delta_{\mathtt{r}}$, where Δ_r comprises those residual resources described by the antecedent Δ_a, but not by the consequent Δ_c. Intuitively, a valid entailment suggests that the resource models described by Δ_a are sufficient to conclude the availability of those described by Δ_c.

Since the proposed logic is tailored to support reasoning about communication primitives with generic protocol specifications, the entailment should also be able to interpret and instantiate such generic specifications. Therefore we equip the entailment checker to reason about formulae which contain second-order variables. Consequently, the proposed entailment is designed to support the instantiation of such variables. However, the instantiation might not be unique, so we collect the candidate instantiations in a set of residual states. The entailment has thus the following form: $\Delta_{\mathtt{a}} \vdash^{\kappa}_{\mathtt{Q}} \Delta_{\mathtt{c}} \rightsquigarrow S$ (shorthand for $\bigcup_{\Delta_{\mathtt{r}} \in S} (\kappa * \Delta_{\mathtt{a}} \vdash \exists \mathtt{Q} \cdot (\kappa * \Delta_{\mathtt{c}}) * \Delta_{\mathtt{r}})$), where S is the set of possible residual states, and κ is the matched heap. Note that S is derived and its size should be of at least 1 in order to consider the entailment as valid. Using the model in Fig. 5.1, the fact that $\Delta_{\mathtt{a}}$ entails $\Delta_{\mathtt{c}}$ with residue S is semantically described by:

$(s,h,o) \vDash \Delta_{\mathtt{a}} \vdash \Delta_{\mathtt{c}} \rightsquigarrow S$ iff $(s,h,o) \vDash \Delta_{\mathtt{a}} \Rightarrow (|\mathtt{S}| \geq 1$ and $\exists h_{\mathtt{c}}, h_{\mathtt{r}} \in \mathtt{Heap} \cdot (h_{\mathtt{c}} \perp h_{\mathtt{r}}$ and $h{=}h_{\mathtt{c}} \uplus h_{\mathtt{r}}$ and $(s,h_{\mathtt{c}},o) \vDash \Delta_{\mathtt{c}}$ and $\forall \Delta_{\mathtt{r}} \in S \Rightarrow (s,h_{\mathtt{r}},o) \vDash \Delta_{\mathtt{r}}))$.

The non-session related rules used for the manipulation of general resource predicates are adapted from [20], a work that introduces a Separation Logic solver for user-defined inductive predicates. The entailment rules needed to accommodate session reasoning are formally described in Fig. 5.4 and informally in the subsequent:

$$\frac{\begin{array}{c}[\textbf{ENT-CHAN-MATCH}]\\ \Delta_a \Rightarrow c_1{=}c_2 \wedge P_1{=}P_2 \qquad \mathcal{C}(c_1,P_1,L_a) \vdash^{\kappa}_{Q} \mathcal{C}(c_2,P_2,L_c) \rightsquigarrow S_1 \\ S_2 = \{\pi^e_i \mid \pi^e_i{\in}S_1 \text{ and } \mathtt{SAT}(\Delta_a{\wedge}\pi^e_i) \text{ and } \mathtt{SAT}(\Delta_c{\wedge}\pi^e_i)\} \qquad \bigvee_{\pi^e \in S_2} (\Delta_a \wedge \pi^e) \vdash^{\kappa * \mathcal{C}(c_1,P_1,L_a)}_{Q} \Delta_c \rightsquigarrow S\end{array}}{\mathcal{C}(\tilde{c}_1,P_1,L_a) * \Delta_a \vdash^{\kappa}_{Q} \mathcal{C}(\tilde{c}_2,P_2,L_c) * \Delta_c \rightsquigarrow S}$$

$$\frac{[\textbf{ENT-CHAN}]\quad L_a \vdash^{\kappa}_{Q} L_c \rightsquigarrow S' \qquad S{=}\{\pi^e_i \mid \pi^e_i{\in}S'\}}{\mathcal{C}(c,P,L_a) \vdash^{\kappa}_{Q} \mathcal{C}(c,P,L_c) \rightsquigarrow S} \qquad \frac{[\textbf{ENT-RHS-PVAR}]\quad S{=}\{\mathtt{emp}{\wedge}V{=}L_a\}}{L_a \vdash^{\kappa}_{Q} V \rightsquigarrow S}$$

$$\frac{[\textbf{ENT-SEQ}]\quad \Box_a \vdash^{\kappa}_{Q} \Box_c \rightsquigarrow S_1 \qquad L_a \vdash^{\kappa}_{Q} L_c \rightsquigarrow S_2 \qquad \text{where } \Box ::= ?v\cdot\Delta \mid !v\cdot\Delta}{\Box_a;L_a \vdash^{\kappa}_{Q} \Box_c;L_c \rightsquigarrow \{\mathtt{emp}{\wedge}\pi_1{\wedge}\pi_2 \mid \pi_1{\in}S_1 \text{ and } \pi_2{\in}S_2\}}$$

$$\frac{[\textbf{ENT-RECV}]\quad \text{fresh } w \qquad \rho{=}[w/v_1,w/v_2] \qquad \rho(\Delta_a) \vdash^{\kappa}_{Q-\{v_1,v_2\}} \rho(\Delta_c) \rightsquigarrow S' \qquad S{=}\{\pi^e_i \mid \pi^e_i{\in}S'\}}{?v_1\cdot\Delta_a \vdash^{\kappa}_{Q} ?v_2\cdot\Delta_c \rightsquigarrow S}$$

$$\frac{[\textbf{ENT-SEND}]\quad \text{fresh } w \qquad \rho{=}[w/v_1,w/v_2] \qquad \rho(\Delta_c) \vdash^{\kappa}_{Q-\{v_1,v_2\}} \rho(\Delta_a) \rightsquigarrow S' \qquad S{=}\{\pi^e_i \mid \pi^e_i{\in}S'\}}{!v_1\cdot\Delta_a \vdash^{\kappa}_{Q} !v_2\cdot\Delta_c \rightsquigarrow S}$$

$$\frac{[\textbf{ENT-LHS-OR}]\quad L_i;L_a \vdash^{\kappa}_{Q} L_c \rightsquigarrow S_i \qquad S = \{\bigvee_i \Delta_i \mid \Delta_i{\in}S_i\}}{(\bigvee_i L_i);L_a \vdash^{\kappa}_{Q} L_c \rightsquigarrow S} \qquad \frac{[\textbf{ENT-RHS-OR}]\quad L_a \vdash^{\kappa}_{Q} L_i;L_c \rightsquigarrow S_i \qquad S = \bigcup S_i}{L_a \vdash^{\kappa}_{Q} (\bigvee_i L_i);L_c \rightsquigarrow S}$$

$$\frac{[\textbf{ENT-LHS-HOV}]\quad V \notin \mathrm{fv}(\Delta_c) \qquad \mathtt{SAT}(\Delta_c) \qquad \text{fresh } w \qquad S{=}\{\mathtt{emp}{\wedge}V(w){=}[w/v]\Delta_c\}}{V(v) \vdash^{\kappa}_{Q} \Delta_c \rightsquigarrow S}$$

$$\frac{[\textbf{ENT-RHS-HOV}]\quad V \notin \mathrm{fv}(\Delta_a) \qquad \Delta_a \vdash^{\kappa}_{Q} \Delta_c \rightsquigarrow S' \qquad \text{fresh } w \qquad S{=}\{\mathtt{emp}{\wedge}V(w){=}[w/v]\Delta_i \mid \Delta_i{\in}S'\}}{\Delta_a \vdash^{\kappa}_{Q} V(v) * \Delta_c \rightsquigarrow S}$$

π^e is a shorthand for $\mathtt{emp}{\wedge}\pi$, $\mathtt{fv}(\Delta)$ returns all free variables in Δ, and `fresh` denotes a fresh variable.

Figure 5.4: Selected Entailment Rules.

[ENT-CHAN-MATCH] indicates that channel matches are solved prior to other resources. Moreover, a channel match is only fired provided that the current proving context is able to derive the fact that the $\mathcal{C}$ predicates refer to the same logical endpoint ($c_1{=}c_2 \wedge P_1{=}P_2$).

Lastly, the residue derived after a channel match is filtered such that it only keeps meaningful instantiations, namely those instantiations not contradicting the antecedent or consequent, hence the need for `SAT` check.

[ENT–CHAN] two representations of channel endpoints match only when the consequent's corresponding communication specification follows deductively from the antecedent's specification. Furthermore, only classic entailment is considered, that is to say that the residue should not contain any resource reference, but only pure formulae.

[ENT–RHS–PVAR] the instantiation of higher-order variables is mostly useful when checking the call of generic methods, and is constrained to simple consequent comprising solely the higher order variable which is to be instantiated to a communication specification.

[ENT–SEQ] the communication specifications composed using strict sequencing conjunction are progressively examined by checking the head transmissions subsumption prior to its tail.

[ENT–RECV] checks for covariant subsumption of the communication patterns.

[ENT–SEND] the dual of [ENT–RECV], enforces contravarinat subsumption check since the information should only flow from a stronger constraint towards a weaker one.

[ENT–LHS–OR] assuming well-formed choices, an entailment with disjunctive antecedent holds only when the consequent follows deductively from each LHS disjunct.

[ENT–RHS–OR] an entailment with disjunctive consequent holds only when at least one of the consequent's disjuncts follows from the antecedent.

[ENT–LHS–HOV] a higher-order variable with first order variables dependencies in the context of spatial consequent, can only be soundly instantiated when the antecedent comprises solely the higher order variable.

[ENT–RHS–HOV] the instantiation of higher-order variables from the consequent is more loose than that of antecedent instantiation, since it allows a spatial context along with the higher-order variable. The instantiation behaves as a container for the residual states S'.

Considering the example below, a context expecting to read an integer greater than or equal to 1 could engage a channel designed with a more relaxed specification **(a)**. However, a context

expecting to transmit an integer greater than or equal to 1 should only be allowed to engage a more specialized channel, such as one designed to solely transmit 1 **(b)**:

$$\dfrac{\dfrac{\mathtt{v}_1{\geq}1 \vdash [\mathtt{v}_1/\mathtt{v}_2]\mathtt{v}_2{\geq}0}{?\mathtt{v}_1 \cdot \mathtt{v}_1{\geq}1 \vdash ?\mathtt{v}_2 \cdot \mathtt{v}_2{\geq}0}\ \mathtt{ENT{-}RECV}}{\mathcal{C}(\mathtt{c}, \mathtt{P}, ?\mathtt{v}_1 \cdot \mathtt{v}_1{\geq}1) \vdash \mathcal{C}(\mathtt{c}, \mathtt{P}, ?\mathtt{v}_2 \cdot \mathtt{v}_2{\geq}0)}\ \mathtt{ENT{-}CHAN}$$

(a)

$$\dfrac{\dfrac{[\mathtt{v}_1/\mathtt{v}_2]\mathtt{v}_2{=}1 \vdash \mathtt{v}_1{\geq}1}{!\mathtt{v}_1 \cdot \mathtt{v}_1{\geq}1 \vdash !\mathtt{v}_2 \cdot \mathtt{v}_2{=}1}\ \mathtt{ENT{-}SEND}}{\mathcal{C}(\mathtt{c}, \mathtt{P}, !\mathtt{v}_1 \cdot \mathtt{v}_1{\geq}1) \vdash \mathcal{C}(\mathtt{c}, \mathtt{P}, !\mathtt{v}_2 \cdot \mathtt{v}_2{=}1)}\ \mathtt{ENT{-}CHAN}$$

(b)

5.2 Explicit Synchronization

Depending on the communication context, and on the instrument used for communication we could opt amongst a few explicit synchronization mechanisms, such as `CountDownLatch`, `wait − notifyAll`, `barriers`, etc. The choice of these mechanisms are orthogonal to our approach. We have experimented some examples with both `CountDownLatch` and `wait − notifyAll` and obtained similar results. We chose to formalize the latter since its simplicity suffice for our running example.

The creation of a conditional-variable for `wait`, [CREATE] in Fig. 5.5, releases the specification to verify the calls of `notifyAll` and `wait`, respectively, with respect to the session logic orderings.

A call to notify is safe only if the triggering event has occurred already, [NOTIFY−ALL]. In other words, the caller's state should contain the triggering event information. [WAIT] on the other hand, releases an ordering assumption conditioned by the send/receive event which is protected by the current `wait`. The condition has a double meaning in this context: (i) the protected event should have not occurred before a call to `wait`, and (ii) the ordering is released to the state only after proving that the event indeed occurred, facilitated by the *Wait lemma* in Fig. 5.5.

The ease of detecting a `wait−notifyAll` deadlock (within a single synchronization object) is a bonus offered by our logic, since it is simply reduced to checking whether there is any context in which a call to `wait` terminated without a corresponding `notify` call. Formally, this is captured by:

$$\frac{[\textbf{CREATE}]\quad \mathtt{V} = \bigwedge_{\mathtt{j}\in\{2..\mathtt{n}\}} \oplus(\mathtt{E_j} \Rightarrow \mathtt{E_1} \prec_{\mathtt{HB}} \mathtt{E_j})}{\vdash \{\mathtt{emp}\}\ \mathbf{create()\ with\ E_1, \overline{E_2..E_n}}\ \{\mathtt{NOTIFY}(\mathtt{E_1}, \ominus(\mathtt{E_1})) * \mathtt{WAIT}(\overline{\mathtt{E_2..E_n}}, \mathtt{V})\}}$$

$$[\textbf{NOTIFY-ALL}]\quad \vdash \{\mathtt{NOTIFY}(\mathtt{E}, \ominus(\mathtt{E})) \wedge \mathtt{E}\}\ \mathbf{notifyAll(E)}\ \{\mathtt{NOTIFY}(\mathtt{E}, \mathtt{emp})\}$$

$$\frac{[\textbf{WAIT}]\quad \mathtt{V^{rel}} = \oplus(\mathtt{E} \Rightarrow \mathtt{E_1} \prec_{\mathtt{HB}} \mathtt{E})}{\vdash \{\mathtt{WAIT}(\mathtt{E}, \mathtt{V^{rel}}) \wedge \neg(\mathtt{E})\}\ \mathbf{wait(E)}\ \{\mathtt{WAIT}(\mathtt{E}, \mathtt{emp}) * \mathtt{V^{rel}}\}}$$

$$[\textbf{Wait}]\quad \oplus(\mathtt{E_2} \Rightarrow \mathtt{E_1} \prec_{\mathtt{HB}} \mathtt{E_2}) \wedge \mathtt{E_2} \Rightarrow \mathtt{E_1} \prec_{\mathtt{HB}} \mathtt{E_2}$$

$$[\textbf{Distribute-waits}]\quad \mathtt{WAIT}(\overline{\mathtt{E_2..E_n}}, \bigwedge_{\mathtt{j}\in\{2..\mathtt{n}\}} \Psi_\mathtt{j}) \Rightarrow \bigwedge_{\mathtt{j}\in\{2..\mathtt{n}\}} \mathtt{WAIT}(\overline{\mathtt{E_2..E_n}}, \Psi_\mathtt{j})$$

Figure 5.5: Synchronization Primitives for `wait-notifyAll`

$$\mathtt{NOTIFY}(\mathtt{E}, \ominus(\mathtt{E})) * \mathtt{WAIT}(\mathtt{E}, \mathtt{emp}) \ \mapsto\ \bot_{deadlock}$$

For more general deadlocks across multiple synchronization objects, we will need to build *waits-for* graphs amongst these objects and detect cycles, where possible, using lemmas similar to the above. For simplicity, these issues are ignored in the current presentation.

5.3 Deadlock Detection

This section continues the discussion over the inter-channel deadlock introduced in Sec. 3.6, lifting this issue to the context of multiparty communication and proposing a precise method to detect or prove the absence of such deadlocks. To start let us consider two communicating scenarios as described by the following global protocols:

$$\mathtt{H_1}(\mathtt{A}, \mathtt{B}, \mathtt{c_1}, \mathtt{c_2},) \triangleq \mathtt{A} \xrightarrow{1} \mathtt{B} : \mathtt{c_1}\ ;\ \mathtt{B} \xrightarrow{2} \mathtt{A} : \mathtt{c_2}.$$
$$\mathtt{H_2}(\mathtt{A}, \mathtt{B}, \mathtt{c_1}, \mathtt{c_2},) \triangleq \mathtt{A} \xrightarrow{1} \mathtt{B} : \mathtt{c_1} * \mathtt{B} \xrightarrow{2} \mathtt{A} : \mathtt{c_2}.$$

$\mathtt{H_1}$ and $\mathtt{H_2}$ both refer to the same transmissions, however the composition of these transmissions differs: the former imposes strict sequencing, while the latter assumes arbitrary transmission order. Consider next the four implementation configurations for parties `A` and `B` as depicted in Fig. 5.6, where Implementation 2 in Fig. 5.6b contains a deadlock. The attempts to verify these implementations against the endpoint projections of $\mathtt{H_1}$ fail, except for Implementation 3 which respects the strict sequencing required by $\mathtt{H_1}$. $\mathtt{H_1}$ therefore avoids the deadlock scenario of Fig. 5.6b by imposing strict sequencing between its transmission. However, due to the use of concurrent composition $*$ in its specification, all of the four implementations verify against the

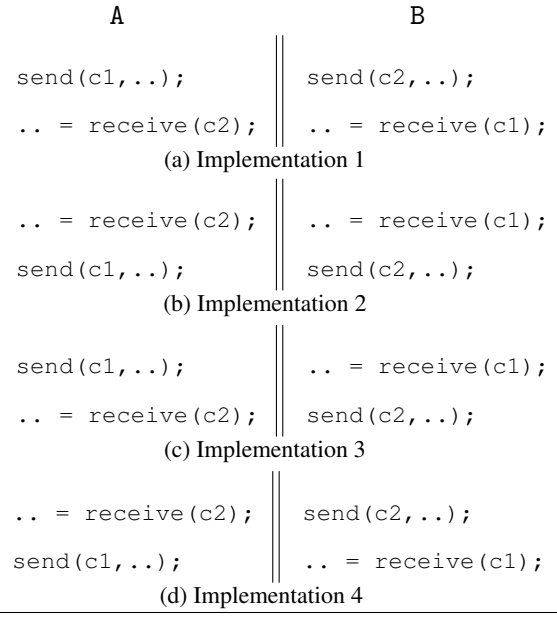


Figure 5.6: Communication Implementations

projections of the more relaxed protocol $\mathtt{H}_2$. That is to say that protocols such as $\mathtt{H}_2$ are prone to inter-deadlock scenarios and therefore their implementation needs to be checked accordingly.

To detect such deadlocks we propose a variant of the wait-for graphs where the vertices are represented by channels instead of processes as in the classic implementation of wait-for graphs. Traditionally, wait-for graphs are obtained from resource-allocation graphs [89] by removing the resource nodes and contracting the edges corresponding to the removed nodes. More precisely, for a traditional such graph, an edge from some node $\mathtt{P}_1$ to node $\mathtt{P}_2$ signifies that the process corresponding to $\mathtt{P}_1$ is waiting for the process corresponding to $\mathtt{P}_2$ to release a resource that the process of $\mathtt{P}_1$ requires. In other words, if the resource allocation graph contains a node $\mathtt{R}$ corresponding to some resource, and two edges $(\mathtt{P}_1, \mathtt{R})$ and $(\mathtt{R}, \mathtt{P}_2)$ then its corresponding wait-for graph contains the $(\mathtt{P}_1, \mathtt{P}_2)$ edge, denoted by $\mathtt{P}_1 \rightarrow \mathtt{P}_2$ and read "$\mathtt{P}_1$ waits for $\mathtt{P}_2$ ".

This chapter proposes a new kind of wait-for graph based on transmissions constructed w.r.t. an expression $\mathtt{e}$ and a protocol $\mathtt{G}$ which describes the transmissions of $\mathtt{e}$. This wait-for graph is represented by a directed graph whose nodes are the receive and send operations of $\mathtt{e}$ abstracted as a pair of the form $\tilde{\mathtt{c}}^i$. $\tilde{\mathtt{c}}$ represents the program channel used for consuming the corresponding operation, and $i \in \mathtt{G}$ is the unique label identifier which binds the considered program transmission from $\mathtt{e}$ to its corresponding logical transmission in $\mathtt{G}$.

Since the receive operation is blocking while send in non-blocking, a natural way to add

edges to a channel-based wait-for graph is to link the send operations to prior blocking receives. An edge $(\tilde{c}_1^{i_1}, \tilde{c}_2^{i_2})$, also written $\tilde{c}_1^{i_1} \rightarrow \tilde{c}_2^{i_2}$, could be read in this context as "the send on $\tilde{c}_1$ corresponding to transmission i_1 is waiting for the receive on $\tilde{c}_2$ corresponding to transmission i_2 to be consumed". To this purpose, and according to the brief introduction of deadlock detection in Sec. 3.6, an *accumulation predicate* $\texttt{WFG}(\texttt{R}, \texttt{W})$ tracks the prior blocking receives via the R set argument and the accumulated wait-for graph with W.

Since deadlocks cannot be locally detected, it is necessary to accumulate dependency information across all communicating threads, making thus the $\texttt{WFG}(\texttt{R}, \texttt{W})$ shareable. To ensure that the accumulated information is complete, $\texttt{WFG}(\texttt{R}, \texttt{W})$ is treated as a mutable resource initiated with a full permission [4] which is then split across all threads. The information is considered complete at the threads joint points when $\texttt{WFG}(\texttt{R}, \texttt{W})@1$, that is when the resource gains full permission again. Different from the usual fractional permission usage, where a partial permission allows only read access, in this context mutation/update over $\texttt{WFG}(\texttt{R}, \texttt{W})@\texttt{f}$ is allowed even for partial permission, $@\texttt{f}<1$, since the fractions are solely used to ensure complete wait-for graph when attached to accumulation predicates.

Adopting this thesis' usual style of manipulating abstract predicates via carefully designed lemmas (applied according to the lemma mechanism introduced in [72]), the following set of lemmas are crafted for being used with $\texttt{WFG}(\texttt{R}, \texttt{W})@\texttt{f}$ predicates:

$$[\underline{\textbf{ACC}}] \qquad \texttt{WFG}(\texttt{R}_1, \texttt{W}_1)@\texttt{f}_1 * \texttt{WFG}(\texttt{R}_2, \texttt{W}_2)@\texttt{f}_2 \mapsto \texttt{WFG}(\texttt{R}_1 \cup \texttt{R}_2, \texttt{W}_1 \cup \texttt{W}_2)@\texttt{f}_1 + \texttt{f}_2.$$

$$[\underline{\textbf{DEADLOCK}}] \quad \texttt{WFG}(\texttt{R}, \texttt{W})@\texttt{f} \wedge \texttt{is\_cycle}(\texttt{W}) \mapsto \bot_{deadlock}.$$

$$[\underline{\textbf{RESET}}] \qquad \texttt{WFG}(\texttt{R}, \texttt{W})@1 \wedge \neg(\texttt{is\_cycle}(\texttt{W})) \mapsto \texttt{WFG}(\{\}, \{\})@1.$$

where $\texttt{is\_cycle}(\texttt{W})$ detects the cycles in W and is straightforwardly defined as:

$$\frac{\exists(\tilde{c}_1^{i_1}, \tilde{c}_2^{i_2}), (\tilde{c}_3^{i_3}, \tilde{c}_4^{i_4}) \in \texttt{W} \cdot \tilde{c}_1{=}\tilde{c}_4 \wedge \tilde{c}_2{=}\tilde{c}_3 \wedge i_1{=}i_4 \wedge i_2{=}i_3}{\texttt{is\_cycle}(\texttt{W})}$$

Some of the proof rules described earlier in Fig. 5.2 are now enhanced as per Fig. 5.7 such that they account for the newly introduced accumulation predicate. [**WFG–PAR**] uniformly distributes the accumulation predicate between $\texttt{e}_1$ and $\texttt{e}_2$. [**WFG–SEND**] accumulates in the wait-for graph W the dependencies between the current transmission i on channel $\tilde{c}$ and all the previous blocking receives. [**WFG–RECV**] simply records the current receive event. All the other proof rules remain unchanged treating WFG as a resource predicate. The conditional poses

$$\frac{(\mathtt{fv}(\Delta_1) \cup \mathtt{fv}(\mathtt{S}_1)) \cap \mathtt{modif}(\mathtt{e}_2) = \emptyset \qquad (\mathtt{fv}(\Delta_2) \cup \mathtt{fv}(\mathtt{S}_2)) \cap \mathtt{modif}(\mathtt{e}_1) = \emptyset \quad \vdash \{\Delta_1 * \mathtt{WFG}(\mathtt{R},\mathtt{W})@\mathtt{f}/2\}\ \mathtt{e}_1\ \{\mathtt{S}_1 * \mathtt{WFG}(\mathtt{R}_1,\mathtt{W}_1)@\mathtt{f}/2\} \quad \vdash \{\Delta_2 * \mathtt{WFG}(\mathtt{R},\mathtt{W})@\mathtt{f}/2\}\ \mathtt{e}_2\ \{\mathtt{S}_2 * \mathtt{WFG}(\mathtt{R}_2,\mathtt{W}_2)@\mathtt{f}/2\}}{\vdash \{\Delta_1 * \Delta_2 * \mathtt{WFG}(\mathtt{R},\mathtt{W})@\mathtt{f}\}\ \mathtt{e}_1 || \mathtt{e}_2\ \{\mathtt{S}_1 * \mathtt{S}_2 * \mathtt{WFG}(\mathtt{R}_1,\mathtt{W}_1)@\mathtt{f}/2 * \mathtt{WFG}(\mathtt{R}_2,\mathtt{W}_2)@\mathtt{f}/2\}}\ [\textbf{WFG–PAR}]$$

$$\frac{\mathtt{WFG}_{\mathtt{pre}} \triangleq \mathtt{WFG}(\mathtt{R},\mathtt{W})@\mathtt{f} \qquad \mathtt{WFG}_{\mathtt{post}} \triangleq \mathtt{WFG}(\mathtt{R},\mathtt{W} \cup \{(\tilde{\mathtt{c}}^i, \tilde{\mathtt{c}}_0^{i_0}) | \tilde{\mathtt{c}}_0^{i_0} \in \mathtt{R}\})@\mathtt{f} \quad \mathtt{inv} \triangleq \mathtt{Peer}(\mathtt{P}) * \mathtt{opened}(\mathtt{c},\mathtt{P}^*,\tilde{\mathtt{c}}) \wedge \mathtt{P} \in \mathtt{P}^*}{\vdash \{\mathcal{C}(\mathtt{c},\mathtt{P},{}^i!\mathtt{v}\cdot\mathtt{V}(\mathtt{v});\mathtt{L}) * \mathtt{V}(\mathtt{x}) * \mathtt{WFG}_{\mathtt{pre}} * \mathtt{inv}\}\ \mathtt{send}(\tilde{\mathtt{c}},\mathtt{x})\ \{\mathcal{C}(\mathtt{c},\mathtt{P},\mathtt{L}) * \mathtt{WFG}_{\mathtt{post}} * \mathtt{inv}\}}\ [\textbf{WFG–SEND}]$$

$$\frac{\mathtt{WFG}_{\mathtt{pre}} \triangleq \mathtt{WFG}(\mathtt{R},\mathtt{W})@\mathtt{f} \qquad \mathtt{WFG}_{\mathtt{post}} \triangleq \mathtt{WFG}(\mathtt{R} \cup \{\tilde{\mathtt{c}}^i\},\mathtt{W})@\mathtt{f} \quad \mathtt{inv} \triangleq \mathtt{Peer}(\mathtt{P}) * \mathtt{opened}(\mathtt{c},\mathtt{P}^*,\tilde{\mathtt{c}}) \wedge \mathtt{P} \in \mathtt{P}^*}{\vdash \{\mathcal{C}(\mathtt{c},\mathtt{P},{}^i?\mathtt{v}\cdot\mathtt{V}(\mathtt{v});\mathtt{L}) * \mathtt{WFG}_{\mathtt{pre}} * \mathtt{inv}\}\ \mathtt{recv}(\tilde{\mathtt{c}})\ \{\mathcal{C}(\mathtt{c},\mathtt{P},\mathtt{L}) * \mathtt{V}(\mathtt{res}) * \mathtt{WFG}_{\mathtt{post}} * \mathtt{inv}\}}\ [\textbf{WFG–RECV}]$$

Figure 5.7: Enhanced Communication Primitives with Specs for Wait-For Graphs

no difficulty in accurately detecting a deadlock if one exists, or prove its absence since the proof is designed to track the program state path-sensitively.

Let us next explore how this deadlock detection actually gels for the communication configurations of Fig. 5.6 and the protocol described by $\mathtt{H}_2$ defined at the beginning of this section. Consider the following program which executes the code for parties A and B, w.r.t. to the problematic implementation, namely Implementation 2 (for brevity the abstract states omit most of the details related to communication and resources, and solely focus on the details related to WFG):

```
1   c1 = open() with (c1,{A,B});
2   c2 = open() with (c2,{A,B});
3   // WFG({},{})@1 * opened(c1,{A,B},c1) * opened(c2,{A,B},c2) * ...
4   // Peer(A) * WFG({},{})@0.5                  || // Peer(B) * WFG({},{})@0.5
5   .. = receive(c2);                            || .. = receive(c1);
6   // Peer(A) * WFG({c2^2},{})@0.5              || // Peer(B) * WFG({c1^1},{})@0.5
7   send(c1,..);                                 || send(c2,..);
8   // Peer(A) * WFG({c2^2},{c1^1 → c2^2})@0.5   || // Peer(B) * WFG({c1^1},{c2^2 → c1^1})@0.5
9   // WFG({c1^1,c2^2},{c1^1 → c2^2, c2^2 → c1^1})@1 * ...
10  // verification FAILS: deadlock detected via [DEADLOCK]
```

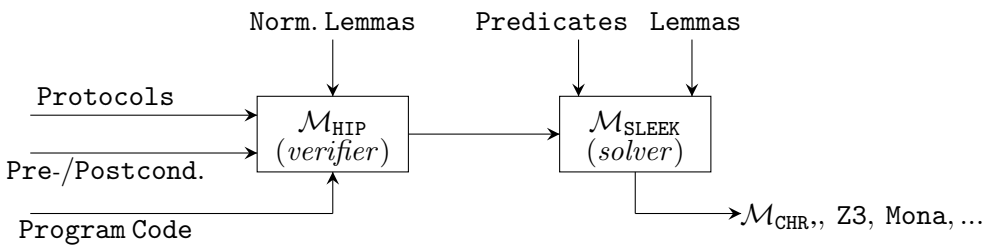


Figure 5.8: Mercurius Overview.

It is easy to notice that if lines 5 and 7 get replaced by the configuration in Fig. 5.6c for example, the verification of thread join is successful: the program state after line 7 would include $\texttt{Peer(A)} * \texttt{WFG}(\{\texttt{c2}^2\}, \{\})@0.5$ and $\texttt{Peer(B)} * \texttt{WFG}(\{\texttt{c1}^1\}, \{\texttt{c2}^2 \rightarrow \texttt{c1}^1\})@0.5$, respectively, and the abstract state post thread join should contain $\texttt{WFG}(\{\texttt{c1}^1, \texttt{c2}^2\}, \{\texttt{c2}^2 \rightarrow \texttt{c1}^1\})@1$ - an instance of the waits-for graph with no cycle.

The combination of transmission identifier and program channel in constructing the wait-for graph, unambiguously identifies each send or receive operations even in the context in which a channel is used for multiple operations within the same thread.

5.4 Implementation

We have developed $\mathcal{M}$ercurius, a prototype implementation for the proposed logic. $\mathcal{M}$ercurius is written in OCaml and integrated with the HIP/SLEEK verification framework [20]. Fig. 5.8 draws its main components: the user provided inputs - program code, method specifications and protocols, the lemmas specific to the communication logic, the verifier - $\mathcal{M}_{\texttt{HIP}}$, the entailment solver - $\mathcal{M}_{\texttt{SLEEK}}$, the user defined predicates and lemmas, $\mathcal{M}_{\texttt{CHR}}$ - a constraint handling system written in CHR, and the selected off the shelf provers Z3 and Mona for arithmetic and bag theory, respectively.

The program verifier $\mathcal{M}_{\texttt{HIP}}$ takes as input the program code to be certified along with the protocols corresponding to the considered program. Each method definition inside the program is expected to be decorated with a set of pre- and postconditions. The verifier maintains stores containing the program's method definitions, data structures, predicates definitions (the protocols are special kind of predicates whose definitions are communication patterns written using the syntax of Fig. 4.1), and lemmas.

Each method is verified independently of each other, as follows: $\mathcal{M}_{\texttt{HIP}}$ assumes the method's precondition, and abstractly executes the method's body according to the rules in [20]. If each sub-expression of the method's body is successfully verified, $\mathcal{M}_{\texttt{HIP}}$ uses the final abstract state to check if the postcondition provided by the user also holds. If the result is positive, the method is considered safe, otherwise it fails. In addition to the verification rules already implemented in HIP, $\mathcal{M}_{\texttt{HIP}}$ adds support for the verification of the communication primitives introduced in Sec. 5.1. But it does so, not by implementing extra verification rules, but by equipping the existing framework with a series of higher-order predicates carefully manipulated in the program's abstract states by the normalization lemmas in Fig. 5.3. This approach is highly modular, since it allows us to change the details of the underlying theories or the granularity of the verification by manipulating the existing predicates and lemmas, avoiding therefore the need of adding new proof rules or changing the existing ones.

During the verification of each expression, $\mathcal{M}_{\texttt{HIP}}$ makes regular calls to $\mathcal{M}_{\texttt{SLEEK}}$ - the specialized entailment checker. $\mathcal{M}_{\texttt{SLEEK}}$ is an enhancement of SLEEK which mainly adds support for higher-order predicates and higher-order variables. Having robust higher-order mechanisms for entailment checking, allows for a straightforward encoding of the entailment rules in Fig. 5.4. In turn, $\mathcal{M}_{\texttt{SLEEK}}$ discharges the proof obligations for known theories to Z3 and Mona. For dealing with the ordering relations and associated constraint system introduced in Fig. 4.3 we have developed $\mathcal{M}_{\texttt{CHR}}$, a custom designed constraint handling system written in CHR.

$\mathcal{M}$ercurius can be tested online via a web interface which contains further details of its usage and a set of case studies:

$$\texttt{http://loris-5.d2.comp.nus.edu.sg/Mercurius}.$$

We next continue this section with examples which highlight how each of the three components work.

Constraint Handling Rules for the Ordering System. The ordering relations introduced in Sec. 4.2.3 translate into the following CHR relations:

$\mathcal{M}_{\texttt{CHR}}$	Example	Ordering-constraint lang.
ev/1	ev(E)	E
hb/2	hb(E_1, E_2)	$E_1 \prec_{HB} E_2$
cb/2	cb(E_1, E_2)	$E_1 \prec_{CB} E_2$
snot/1	snot(E), snot($E_1 \prec_{HB} E_2$)	–

$\mathcal{M}_{\texttt{CHR}}$ comprises a total of 27 simplification and propagation rules. Below we only provide selected rules:

```
cb-hb   % cb(A,B),hb(B,C)        ==> hb(A,C).
hb-hb   % hb(A,B),hb(B,C)        ==> hb(A,C).
irefl   % hbp(A,A)               <=> false.
neg1    % hb(A,B),snot(hb(A,B)) <=> false.
neg2    % ev(A),snot(ev(A))     <=> false.
neg3    % snot(A;B)             <=> snot(A),snot(B).
neg4    % snot(snot(A))         <=> id(A).
neg5    % snot((A,B))           <=> snot(A);snot(B).
...
```

An entailment query between ordering relations is then translated into an unsatisfiability problem as exemplified below:

(1) $E_1 \prec_{HB} E_2 \wedge E_2 \prec_{HB} E_3 \vdash E_3 \prec_{HB} E_4$.

$\mathcal{M}_{\texttt{CHR}}$:
```
hb(EV1,EV2),hb(EV2,EV3),snot(hb(EV3,EV4)).
hb(EV1,EV2),hb(EV2,EV3),snot(hb(EV3,EV4)),hb(EV1,EV3).
```

$\mathcal{M}_{\texttt{CHR}}$ formula is SAT, therefore (1) is INVALID.

(2) $E_1 \prec_{HB} E_2 \wedge E_2 \prec_{HB} E_3 \vdash E_1 \prec_{HB} E_3$.

$\mathcal{M}_{\texttt{CHR}}$:
```
hb(EV1,EV2),hb(EV2,EV3),snot(hb(EV1,EV3)).
false.
```

$\mathcal{M}_{\texttt{CHR}}$ formula is UNSAT, therefore (2) is VALID.

(3) $E_1 \prec_{CB} E_2 \wedge E_2 \prec_{HB} E_3 \vdash E_1 \prec_{HB} E_3$.

$\mathcal{M}_{\texttt{CHR}}$:
```
cb(EV1,EV2),hb(EV2,EV3),snot(hb(EV1,EV3)).
false.
```

$\mathcal{M}_{\texttt{CHR}}$ formula is UNSAT, therefore (3) is VALID.

Entailment Checker.

Entailment 1. Check channel specification subsumption:

$$\dfrac{\dfrac{\dfrac{\dfrac{\texttt{true}}{\mathtt{v}_2{=}1 \vdash^{\kappa}_{\mathtt{Q}\cup\{\mathtt{v}_1,\mathtt{v}_2\}} [\mathtt{v}_2/\mathtt{v}_1]\mathtt{v}_1{=}1 \rightsquigarrow S_1}\,\text{ENT-SEND}}{!\mathtt{v}_1\cdot\mathtt{v}_1{=}1 \vdash^{\kappa}_{\mathtt{Q}} !\mathtt{v}_2\cdot\mathtt{v}_2{=}1 \rightsquigarrow S_1}\,\text{ENT-SEND} \quad \dfrac{\dfrac{\dots}{?2 \vdash^{\kappa}_{\mathtt{Q}} ?2 \rightsquigarrow S_2} \quad \dfrac{\texttt{fail}}{?3 \vdash^{\kappa}_{\mathtt{Q}} \mathtt{emp} \rightsquigarrow \emptyset}\,\text{ENT-SEQ}}{?2;?3 \vdash^{\kappa}_{\mathtt{Q}} ?2 \rightsquigarrow \emptyset}\,\text{ENT-SEQ}}{!1 \vdash^{\kappa}_{\mathtt{Q}} !1 \rightsquigarrow S_1}\,\text{ENT-SEQ}}{!1;?2;?3 \vdash^{\kappa}_{\mathtt{Q}} !1;?2 \rightsquigarrow \emptyset}\,\text{ENT-SEQ}}{\mathcal{C}(\tilde{\mathtt{c}},\mathtt{P},!1;?2;?3) \vdash^{\kappa}_{\mathtt{Q}} \mathcal{C}(\tilde{\mathtt{c}},\mathtt{P},!1;?2) \rightsquigarrow \emptyset}\,\text{ENT-CHAN}$$

Entailment 2. Check channel specification subsumption with order assumption:

$$\dfrac{\dfrac{\dfrac{\dots}{\mathcal{C}(\tilde{\mathtt{c}},\mathtt{P},!1) \vdash^{\kappa}_{\mathtt{Q}} \mathcal{C}(\tilde{\mathtt{c}},\mathtt{P},!1) \rightsquigarrow \{\mathtt{emp}\}} \quad \dfrac{\texttt{true}}{\mathtt{E1}{\prec_{\mathtt{HB}}}\mathtt{E2} \vdash^{\kappa * \mathcal{C}(\tilde{\mathtt{c}},\mathtt{P},!1)}_{\mathtt{Q}} \mathtt{emp} \rightsquigarrow \{\mathtt{emp}\}}\,\text{ENT-CHAN}}{\mathcal{C}(\tilde{\mathtt{c}},\mathtt{P},!1) \wedge \mathtt{E1}{\prec_{\mathtt{HB}}}\mathtt{E2} \vdash^{\kappa}_{\mathtt{Q}} \mathcal{C}(\tilde{\mathtt{c}},\mathtt{P},!1) \rightsquigarrow \{\mathtt{emp}\}}\,\text{L+}}{\mathcal{C}(\tilde{\mathtt{c}},\mathtt{P},\oplus(\mathtt{E1}{\prec_{\mathtt{HB}}}\mathtt{E2});!1) \vdash^{\kappa}_{\mathtt{Q}} \mathcal{C}(\tilde{\mathtt{c}},\mathtt{P},!1) \rightsquigarrow \{\mathtt{emp}\}}\,\text{L+}$$

Entailment 3. Check channel specification subsumption with guard:

$$
\dfrac{
\dfrac{\dfrac{\cdots}{\mathcal{C}(\tilde{c}_2,\mathtt{P},!2) \wedge \mathtt{E1}{\prec_{\mathtt{HB}}}\mathtt{E2} \wedge \mathtt{E2}{\prec_{\mathtt{HB}}}\mathtt{E3} \vdash^{\kappa}_{\mathtt{Q}} \mathcal{C}(\tilde{c}_2,\mathtt{P},!2) \rightsquigarrow \{\mathtt{E1}{\prec_{\mathtt{HB}}}\mathtt{E2} \wedge \pi_1\}}}{\mathcal{C}(\tilde{c}_2,\mathtt{P},\oplus(\mathtt{E1}{\prec_{\mathtt{HB}}}\mathtt{E2});!2) \wedge \mathtt{E2}{\prec_{\mathtt{HB}}}\mathtt{E3} \vdash^{\kappa}_{\mathtt{Q}} \mathcal{C}(\tilde{c}_2,\mathtt{P},!2) \rightsquigarrow \{\mathtt{E1}{\prec_{\mathtt{HB}}}\mathtt{E2} \wedge \pi_1\}}\ \mathtt{L+}
\qquad
\dfrac{\mathcal{C}(\tilde{c}_1,\mathtt{P},!1) \wedge \mathtt{E2}{\prec_{\mathtt{HB}}}\mathtt{E3} \wedge \mathtt{E1}{\prec_{\mathtt{HB}}}\mathtt{E2} \wedge \mathtt{E1}{\prec_{\mathtt{HB}}}\mathtt{E3} \wedge \pi_1 \vdash^{\kappa*\mathcal{C}(\tilde{c}_2,\mathtt{P},!2)}_{\mathtt{Q}} \mathcal{C}(\tilde{c}_1,\mathtt{P},!1) \rightsquigarrow \{\mathtt{emp}\}}{\mathcal{C}(\tilde{c}_1,\mathtt{P},\ominus(\mathtt{E1}{\prec_{\mathtt{HB}}}\mathtt{E3});!1) \wedge \mathtt{E2}{\prec_{\mathtt{HB}}}\mathtt{E3} \wedge \mathtt{E1}{\prec_{\mathtt{HB}}}\mathtt{E2} \wedge \pi_1 \vdash^{\kappa*\mathcal{C}(\tilde{c}_2,\mathtt{P},!2)}_{\mathtt{Q}} \mathcal{C}(\tilde{c}_1,\mathtt{P},!1) \rightsquigarrow \{\mathtt{emp}\}}\ \mathtt{L-}
}{
\mathcal{C}(\tilde{c}_1,\mathtt{P},\ominus(\mathtt{E1}{\prec_{\mathtt{HB}}}\mathtt{E3});!1) * \mathcal{C}(\tilde{c}_2,\mathtt{P},\oplus(\mathtt{E1}{\prec_{\mathtt{HB}}}\mathtt{E2});!2) \wedge \mathtt{E2}{\prec_{\mathtt{HB}}}\mathtt{E3} \vdash^{\kappa}_{\mathtt{Q}} \mathcal{C}(\tilde{c}_1,\mathtt{P},!1) * \mathcal{C}(\tilde{c}_2,\mathtt{P},!2) \rightsquigarrow \{\mathtt{emp}\}
}\ \mathtt{ENT-CHAN-MATCH}
$$

$$
\dfrac{
\dfrac{\dfrac{\dfrac{\dfrac{\mathtt{true}}{\mathtt{v}_2{=}1 \vdash^{\kappa}_{\mathtt{Q}\cup\{\mathtt{v}_1,\mathtt{v}_2\}} [\mathtt{v}_2/\mathtt{v}_1]\mathtt{v}_1{=}1 \rightsquigarrow \{\exists \mathtt{v}_2 \cdot \mathtt{emp} \wedge \mathtt{v}_2{=}1\}}}{!\mathtt{v}_1 \cdot \mathtt{v}_1{=}1 \vdash^{\kappa}_{\mathtt{Q}} !\mathtt{v}_2 \cdot \mathtt{v}_2{=}1 \rightsquigarrow \{\exists \mathtt{v}_2 \cdot \mathtt{emp} \wedge \mathtt{v}_2{=}1\}}\ \mathtt{ENT-SEND}}{!1 \vdash^{\kappa*\mathcal{C}(\tilde{c}_2,\mathtt{P},!2)}_{\mathtt{Q}} !1 \rightsquigarrow \{\exists \mathtt{v}_2 \cdot \mathtt{emp} \wedge \mathtt{v}_2{=}1\}}\ \mathtt{ENT-SEND}}{\mathcal{C}(\tilde{c}_1,\mathtt{P},!1) \vdash^{\kappa*\mathcal{C}(\tilde{c}_2,\mathtt{P},!2)}_{\mathtt{Q}} \mathcal{C}(\tilde{c}_1,\mathtt{P},!1) \rightsquigarrow \{\exists \mathtt{v}_2 \cdot \mathtt{emp} \wedge \mathtt{v}_2{=}1\}}\ \mathtt{ENT-CHAN}
\qquad
\dfrac{\mathtt{true}}{\mathtt{emp} \wedge \mathtt{v}_2{=}1 \wedge \mathtt{E2}{\prec_{\mathtt{HB}}}\mathtt{E3} \wedge \mathtt{E1}{\prec_{\mathtt{HB}}}\mathtt{E2} \wedge \mathtt{E1}{\prec_{\mathtt{HB}}}\mathtt{E3} \vdash^{\kappa*\mathcal{C}(\tilde{c}_2,\mathtt{P},!2)*\mathcal{C}(\tilde{c}_1,\mathtt{P},!1)}_{\mathtt{Q}\cup\{\mathtt{v}_2\}} \mathtt{emp} \rightsquigarrow \{\mathtt{emp}\}}
}{
\mathcal{C}(\tilde{c}_1,\mathtt{P},!1) \wedge \mathtt{E2}{\prec_{\mathtt{HB}}}\mathtt{E3} \wedge \mathtt{E1}{\prec_{\mathtt{HB}}}\mathtt{E2} \wedge \mathtt{E1}{\prec_{\mathtt{HB}}}\mathtt{E3} \vdash^{\kappa*\mathcal{C}(\tilde{c}_2,\mathtt{P},!2)}_{\mathtt{Q}} \mathcal{C}(\tilde{c}_1,\mathtt{P},!1) \rightsquigarrow \{\mathtt{emp}\}
}\ \mathtt{ENT-CHAN-MATCH}
$$

Entailment 4. Check channel specification subsumption with higher-order instantiation:

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{\dfrac{\dfrac{\dfrac{\mathtt{fresh}\ \mathtt{w} \quad S_1{=}\{\mathtt{emp} \wedge \mathtt{V}_1(\mathtt{w}){:=}(\mathtt{w}{=}1)\}}{\mathtt{v}_1{=}1 \vdash^{\kappa}_{\mathtt{Q}} \mathtt{V}_1(\mathtt{v}_1) \rightsquigarrow S_1}\ \mathtt{ENT-RHS-HOV}}{\mathtt{v}_1{=}1 \vdash^{\kappa}_{\mathtt{Q}} [\mathtt{v}_1/\mathtt{v}]\mathtt{V}_1(\mathtt{v}) \rightsquigarrow S_1}}{?\mathtt{v}_1 \cdot \mathtt{v}_1{=}1 \vdash^{\kappa}_{\mathtt{Q}} ?\mathtt{v} \cdot \mathtt{V}_1(\mathtt{v}) \rightsquigarrow S_1}\ \mathtt{ENT-RECV}}{?1 \vdash^{\kappa}_{\mathtt{Q}} ?\mathtt{v} \cdot \mathtt{V}_1(\mathtt{v}) \rightsquigarrow S_1}
\quad
\dfrac{S_2{=}\{\mathtt{emp} \wedge \mathtt{V}_2 = ?2;?3\}}{?2;?3 \vdash^{\kappa}_{\mathtt{Q}} \mathtt{V}_2 \rightsquigarrow S_2}\ \mathtt{ENT-PVAR}
\quad
S_3 = \{\mathtt{emp}{\wedge}\pi_1{\wedge}\pi_2 \mid \pi_1{\in}S_1 \text{ and } \pi_2{\in}S_2\}
}{?1;?2;?3 \vdash^{\kappa}_{\mathtt{Q}} ?\mathtt{v} \cdot \mathtt{V}_1(\mathtt{v});\mathtt{V}_2 \rightsquigarrow S_3}\ \mathtt{ENT-SEQ}
}{\mathcal{C}(\tilde{c},\mathtt{P},?1;?2;?3) \vdash^{\kappa}_{\mathtt{Q}} \mathcal{C}(\tilde{c},\mathtt{P},?\mathtt{v} \cdot \mathtt{V}_1(\mathtt{v});\mathtt{V}_2) \rightsquigarrow S_3}\ \mathtt{ENT-CHAN}
\qquad
\dfrac{S_4 = S_3}{S_3 \wedge \mathtt{x}{=}1 \vdash^{\kappa*\mathcal{C}(\tilde{c},\mathtt{P},?1;?2;?3)}_{\mathtt{Q}} \mathtt{V}_1(\mathtt{x}) \rightsquigarrow S_4}\ \mathtt{ENT-SEQ}
}{
\mathcal{C}(\tilde{c},\mathtt{P},?1;?2;?3) \wedge \mathtt{x}{=}1 \vdash^{\kappa}_{\mathtt{Q}} \mathcal{C}(\tilde{c},\mathtt{P},?\mathtt{v} \cdot \mathtt{V}_1(\mathtt{v});\mathtt{V}_2) * \mathtt{V}_1(\mathtt{x}) \rightsquigarrow S_4
}\ \mathtt{ENT-CHAN-MATCH}
$$

```
assume(Peer(I));                 ||  assume(Peer(P));   ||  assume(Peer(DB));
send(c,"EVO~2.5~SATA~III");      ||  send(c,99);        ||  String item = receive(c);
                                 ||                     ||  int    price = receive(c);
```

(a) Implementation 1 (buggy communication)

```
assume(Peer(I));                 ||  assume(Peer(P));   ||  assume(Peer(DB));
send(c,"EVO~2.5~SATA~III");      ||  wait(e);           ||  String item = receive(c);
notifyAll(e);                    ||  send(c,99);        ||  int    price = receive(c);
```

(b) Implementation 2 (safe)

Figure 5.9: Buggy vs Safe Implementation

Verifier. We next reconsider the simple example introduced in Sec. 1.1, but this time specified using the proposed session logic. $\mathtt{H}$ describes a simple protocol between three parties:

$$\mathtt{H} \triangleq \mathtt{I} \xrightarrow{1} \mathtt{DB} : \mathtt{c}_0\langle \mathtt{String} \rangle \,;\, \mathtt{P} \xrightarrow{2} \mathtt{DB} : \mathtt{c}_0\langle \mathtt{v} \cdot \mathtt{v} > 0 \rangle.$$

The automatic protocol refinement described in Sec. 4.2, transforms $\mathtt{H}$ into a race-free protocol by adding the corresponding ordering guards and assumptions:

$$\begin{aligned}\mathtt{H} \triangleq\ & \mathtt{I} \xrightarrow{1} \mathtt{DB} : \mathtt{c}_0\langle \mathtt{String} \rangle \,;\, \oplus(\mathtt{I}^{(1)}); \oplus(\mathtt{DB}^{(1)}); \oplus(\mathtt{I}^{(1)} \prec_{\mathtt{CB}} \mathtt{DB}^{(1)}) \,; \\ & \mathtt{P} \xrightarrow{2} \mathtt{DB} : \mathtt{c}_0\langle \mathtt{v} \cdot \mathtt{v} > 0 \rangle; \oplus(\mathtt{P}^{(2)}); \oplus(\mathtt{DB}^{(2)}); \oplus(\mathtt{P}^{(2)} \prec_{\mathtt{CB}} \mathtt{DB}^{(2)}); \oplus(\mathtt{DB}^{(1)} \prec_{\mathtt{HB}} \mathtt{DB}^{(2)}); \\ & \ominus(\mathtt{I}^{(1)} \prec_{\mathtt{HB}} \mathtt{P}^{(2)}); \ominus(\mathtt{DB}^{(1)} \prec_{\mathtt{HB}} \mathtt{DB}^{(2)}).\end{aligned}$$

The projections derived automatically by applying the rules in Fig. 4.6 are as follows:

$$\begin{aligned}(\mathtt{H})\!\downarrow_{\mathtt{I},\mathtt{c}_0} &:= \mathtt{!String};\ \oplus(\mathtt{I}^{(1)}); \oplus(\mathtt{DB}^{(1)} \prec_{\mathtt{HB}} \mathtt{DB}^{(2)}). \\ (\mathtt{H})\!\downarrow_{\mathtt{P},\mathtt{c}_0} &:= \mathtt{!v \cdot v > 0}; \oplus(\mathtt{P}^{(2)}); \ominus(\mathtt{I}^{(1)} \prec_{\mathtt{HB}} \mathtt{P}^{(2)}); \oplus(\mathtt{DB}^{(1)} \prec_{\mathtt{HB}} \mathtt{DB}^{(2)}). \\ (\mathtt{H})\!\downarrow_{\mathtt{DB},\mathtt{c}_0} &:= \mathtt{?String}; \oplus(\mathtt{DB}^{(1)}); \mathtt{?v \cdot v > 0}; \oplus(\mathtt{DB}^{(2)}); \oplus(\mathtt{I}^{(1)} \prec_{\mathtt{HB}} \mathtt{P}^{(2)}); \ominus(\mathtt{DB}^{(1)} \prec_{\mathtt{HB}} \mathtt{DB}^{(2)}). \\ (\mathtt{H})\!\downarrow_{\mathtt{All}} &:= \{\oplus(\mathtt{I}^{(1)} \prec_{\mathtt{CB}} \mathtt{DB}^{(1)}), \oplus(\mathtt{P}^{(2)} \prec_{\mathtt{CB}} \mathtt{DB}^{(2)}), \oplus(\mathtt{DB}^{(1)} \prec_{\mathtt{HB}} \mathtt{DB}^{(2)})\}.\end{aligned}$$

We next consider the possible implementations of Fig. 5.9 and discuss selected code proofs.

Example 1. We first plug the implementation of Fig. 5.9a into the following main code (where for convenience we emphasized the program's expected abstract states):

```
  //init(c0) * Party(I, c0, (H)↓I) * Party(P, c0, (H)↓P) * Party(DB, c0, (H)↓DB) ∧ (H)↓All
1 Channel c = open() with (c0, {I, P, DB});
  //Party(I, c0, (H)↓I) * Party(P, c0, (H)↓P) * Party(DB, c0, (H)↓DB) ∧ (H)↓All ∧ opened(c0, {I, P, DB}, c)
2 I || P || DB
  //Party(I, c0, emp) * Party(P, c0, emp) * Party(DB, c0, emp) ∧ (H)↓All ∧ opened(c0, {I, P, DB}, c)
3 close(c);
```

To highlight why the implementation in Fig. 5.9a is an unsafe implementation of H when plugged into the above main program, let us consider the abstract states of parties DB and P, since these are the parties which need to prove the race-freedom, i.e the parties which contain guards in their specifications. The proof for the code implementing I is straightforward, and therefore omitted. We start with the proof of DB:

$//\mathtt{Party}(\mathtt{DB}, \mathtt{c}_0, (\mathtt{H})\!\downarrow_{\mathtt{DB}}) \wedge (\mathtt{H})\!\downarrow_{\mathtt{All}} \wedge \mathtt{opened}(\mathtt{c}_0, \{\mathtt{I}, \mathtt{P}, \mathtt{DB}\}, \mathtt{c})$

```
1  assume(Peer(DB));
```

$/*$

$\mathtt{Party}(\mathtt{DB}, \mathtt{c}_0, (\mathtt{H})\!\downarrow_{\mathtt{DB}}) * \mathtt{Peer}(\mathtt{DB}) \wedge (\mathtt{H})\!\downarrow_{\mathtt{All}} \wedge \mathtt{opened}(\mathtt{c}_0, \{\mathtt{I}, \mathtt{P}, \mathtt{DB}\}, \mathtt{c})$

$\rangle\!\rangle\!\rangle$ fire SPLIT LEMMAS $\langle\!\langle\!\langle$

$\mathcal{C}(\mathtt{c}_0, \mathtt{DB}, (\mathtt{H})\!\downarrow_{\mathtt{DB},\mathtt{c}_0}) * \mathtt{Peer}(\mathtt{DB}) \wedge (\mathtt{H})\!\downarrow_{\mathtt{All}} \wedge \mathtt{opened}(\mathtt{c}_0, \{\mathtt{I}, \mathtt{P}, \mathtt{DB}\}, \mathtt{c}) \wedge \mathtt{bind}(\mathtt{DB}, \{\mathtt{c}_0\})$

$\rangle\!\rangle\!\rangle$ unroll $(\mathtt{H})\!\downarrow_{\mathtt{DB},\mathtt{c}_0}$ $\langle\!\langle\!\langle$

$\mathcal{C}(\mathtt{c}_0, \mathtt{DB}, ?\mathtt{String}; \oplus(\mathtt{DB}^{(1)}); ?\mathtt{v} \cdot \mathtt{v} > 0; \oplus(\mathtt{DB}^{(2)}); \oplus(\mathtt{I}^{(1)} \prec_{\mathtt{HB}} \mathtt{P}^{(2)}); \ominus(\mathtt{DB}^{(1)} \prec_{\mathtt{HB}} \mathtt{DB}^{(2)})) *$
$\mathtt{Peer}(\mathtt{DB}) \wedge (\mathtt{H})\!\downarrow_{\mathtt{All}} \wedge \mathtt{opened}(\mathtt{c}_0, \{\mathtt{I}, \mathtt{P}, \mathtt{DB}\}, \mathtt{c})$

$*/$

```
2  String item = receive(c);
```

$/*$

$\mathcal{C}(\mathtt{c}_0, \mathtt{DB}, \oplus(\mathtt{DB}^{(1)}); ?\mathtt{v} \cdot \mathtt{v} > 0; \oplus(\mathtt{DB}^{(2)}); \oplus(\mathtt{I}^{(1)} \prec_{\mathtt{HB}} \mathtt{P}^{(2)}); \ominus(\mathtt{DB}^{(1)} \prec_{\mathtt{HB}} \mathtt{DB}^{(2)})) * \mathtt{item} \mapsto \mathtt{String}\langle\_\rangle *$
$\mathtt{Peer}(\mathtt{DB}) \wedge (\mathtt{H})\!\downarrow_{\mathtt{All}} \wedge \mathtt{opened}(\mathtt{c}_0, \{\mathtt{I}, \mathtt{P}, \mathtt{DB}\}, \mathtt{c})$

$\rangle\!\rangle\!\rangle$ fire LEMMA $[\underline{\mathtt{L}+}]$ $\langle\!\langle\!\langle$

$\mathcal{C}(\mathtt{c}_0, \mathtt{DB}, ?\mathtt{v} \cdot \mathtt{v} > 0; \oplus(\mathtt{DB}^{(2)}); \oplus(\mathtt{I}^{(1)} \prec_{\mathtt{HB}} \mathtt{P}^{(2)}); \ominus(\mathtt{DB}^{(1)} \prec_{\mathtt{HB}} \mathtt{DB}^{(2)})) * \mathtt{item} \mapsto \mathtt{String}\langle\_\rangle *$
$\mathtt{Peer}(\mathtt{DB}) \wedge (\mathtt{H})\!\downarrow_{\mathtt{All}} \wedge \mathtt{opened}(\mathtt{c}_0, \{\mathtt{I}, \mathtt{P}, \mathtt{DB}\}, \mathtt{c}) \wedge \mathtt{DB}^{(1)}$

$*/$

```
3  int   price = receive(c);
```

$/*$

$\mathcal{C}(\mathtt{c}_0, \mathtt{DB}, \oplus(\mathtt{DB}^{(2)}); \oplus(\mathtt{I}^{(1)} \prec_{\mathtt{HB}} \mathtt{P}^{(2)}); \ominus(\mathtt{DB}^{(1)} \prec_{\mathtt{HB}} \mathtt{DB}^{(2)})) * \mathtt{item} \mapsto \mathtt{String}\langle\_\rangle * \mathtt{Peer}(\mathtt{DB}) \wedge (\mathtt{H})\!\downarrow_{\mathtt{All}} \wedge$
$\mathtt{opened}(\mathtt{c}_0, \{\mathtt{I}, \mathtt{P}, \mathtt{DB}\}, \mathtt{c}) \wedge \mathtt{DB}^{(1)} \wedge \mathtt{price} > 0$

$\rangle\!\rangle\!\rangle$ fire LEMMA $[\underline{\mathtt{L}+}]$ twice $\langle\!\langle\!\langle$

$\mathcal{C}(\mathtt{c}_0, \mathtt{DB}, \ominus(\mathtt{DB}^{(1)} \prec_{\mathtt{HB}} \mathtt{DB}^{(2)})) * \mathtt{item} \mapsto \mathtt{String}\langle\_\rangle * \mathtt{Peer}(\mathtt{DB}) \wedge (\mathtt{H})\!\downarrow_{\mathtt{All}} \wedge$
$\mathtt{opened}(\mathtt{c}_0, \{\mathtt{I}, \mathtt{P}, \mathtt{DB}\}, \mathtt{c}) \wedge \mathtt{DB}^{(1)} \wedge \mathtt{price} > 0 \wedge \mathtt{DB}^{(2)} \wedge \mathtt{I}^{(1)} \prec_{\mathtt{HB}} \mathtt{P}^{(2)}$

$\rangle\!\rangle\!\rangle$ fire LEMMA $[\mathtt{L}-]$ $\langle\!\langle\!\langle$

$\mathcal{C}(\mathtt{c}_0, \mathtt{DB}, \mathtt{emp}) * \mathtt{item} \mapsto \mathtt{String}\langle\_\rangle * \mathtt{Peer}(\mathtt{DB}) \wedge (\mathtt{H})\!\downarrow_{\mathtt{All}} \wedge$
$\mathtt{opened}(\mathtt{c}_0, \{\mathtt{I}, \mathtt{P}, \mathtt{DB}\}, \mathtt{c}) \wedge \mathtt{DB}^{(1)} \wedge \mathtt{price} > 0 \wedge \mathtt{DB}^{(2)} \wedge \mathtt{I}^{(1)} \prec_{\mathtt{HB}} \mathtt{P}^{(2)}$

$*/$

The code excerpt implementing DB is successfully verified against its local specification. The application of lemma $[\mathtt{L}-]$ after line 3 is valid since the global store of event orderings,

namely $(\mathtt{H})\downarrow_{\mathtt{All}}$, already contains the information that $\mathtt{DB}^{(1)} \prec_{\mathtt{HB}} \mathtt{DB}^{(2)}$, which refers to the program order.

We now move on to the problematic code of Fig. 5.9a implementing participant P:

```
//Party(P, c0, (H)↓P) ∧ (H)↓All ∧ opened(c0, {I, P, DB}, c)
1  assume(Peer(P));
   /*
     Party(P, c0, (H)↓P) ∗ Peer(P) ∧ (H)↓All ∧ opened(c0, {I, P, DB}, c)
     ⟩⟩⟩  fire SPLIT LEMMAS  ⟨⟨⟨
     C(c0, P, (H)↓P,c0) ∗ Peer(P) ∧ (H)↓All ∧ opened(c0, {I, P, DB}, c) ∧ bind(P, {c0})
     ⟩⟩⟩    unroll (H)↓P,c0    ⟨⟨⟨
     C(c0, P, !v · v > 0; ⊕(P(2)); ⊖(I(1) ≺HB P(2)); ⊕(DB(1) ≺HB DB(2))) ∗ Peer(P) ∧ (H)↓All ∧ opened(c0, {I, P, DB}, c)
   */
2  send(c, 99);
   /*
     C(c0, P, ⊕(P(2)); ⊖(I(1) ≺HB P(2)); ⊕(DB(1) ≺HB DB(2))) ∗ Peer(P) ∧ (H)↓All ∧ opened(c0, {I, P, DB}, c)
     ⟩⟩⟩   fire LEMMA [L+]   ⟨⟨⟨
     C(c0, P, ⊖(I(1) ≺HB P(2)); ⊕(DB(1) ≺HB DB(2))) ∗ Peer(P) ∧ (H)↓All ∧ opened(c0, {I, P, DB}, c) ∧ P(2)
     ⟩⟩⟩   fire LEMMA [L−]   ⟨⟨⟨    ⇒ FAIL
   */
```

It is straightforward to notice that the race-free guard of the above code does not hold. Because the verification of the code for P fails, then the whole main program fails since the three threads cannot join.

Example 2. We next plug the implementation of Fig. 5.9b into the following main code which also uses explicit synchronization We stress on the fact that the `wait-notifyAll` primitives used for this example are just proof of concept to highlight how explicit synchronization influences the communication. These primitives could be replaced by any other synchronization mechanisms with its corresponding specification, provided that they are also annotated with the necessary ordering events. The safe usage of the resources guarded by these primitives is beyond the scope of this thesis, therefore their functionality is abstracted to the point where it solely caters for communication event ordering:

```
//init(c0) * Party(I, c0, (H)|I) * Party(P, c0, (H)|P) * Party(DB, c0, (H)|DB) ∧ (H)|All
1 create () with I(1), {P(2)};
/*
  init(c0) * Party(I, c0, (H)|I) * Party(P, c0, (H)|P) * Party(DB, c0, (H)|DB)*
                                NOTIFY(I(1), ⊖(I(1))) * WAIT({P(2)}, ⊕(P(2) ⇒ I(1)≺HB P(2))) ∧ (H)|All
*/
2 Channel c = open() with (c0, {I, P, DB});
/*
  Party(I, c0, (H)|I) * Party(P, c0, (H)|P) * Party(DB, c0, (H)|DB)*
        NOTIFY(I(1), ⊖(I(1))) * WAIT({P(2)}, ⊕(P(2) ⇒ I(1)≺HB P(2))) ∧ (H)|All ∧ opened(c0, {I, P, DB}, c)
*/
3 I || P || DB
/*
  Party(I, c0, emp) * Party(P, c0, emp) * Party(DB, c0, emp)
                              NOTIFY(I(1), emp) * WAIT({P(2)}, emp) ∧ (H)|All ∧ opened(c0, {I, P, DB}, c)
*/
4 close(c);
```

The code for DB is the same for both implementations of protocol H, therefore its proof is the same as exemplified in the previous example. This highlights an important aspect of local reasoning: once the safety and functionality of a code has been proved it need not be proved again unless it is refactored or its specification changes. In the current case none of the two has changed.

The next two code excerpts represent the implementation of party I and P, respectively, as per Fig. 5.9b, with selected abstract states.

Code of I. The call at line 3 in the implementation of I is successfully verified since event $\mathtt{I}^{(1)}$ has already occurred:

$$\mathtt{NOTIFY}(\mathtt{I}^{(1)}, \ominus(\mathtt{I}^{(1)})) * \Delta \wedge \mathtt{I}^{(1)} \wedge \pi \wedge (\mathtt{E}{=}\mathtt{I}^{(1)}) \vdash^{\kappa}_{\mathtt{Q}} \mathtt{NOTIFY}(\mathtt{E}, \ominus(\mathtt{E})) \wedge \mathtt{E} \rightsquigarrow \{\Delta\}.$$

where $\Delta \wedge \pi$ stand for

$$\mathcal{C}(\mathtt{c}_0, \mathtt{I}, \mathtt{emp}) * \mathtt{Peer}(\mathtt{I}) \wedge (\mathtt{H})\lfloor_{\mathtt{All}} \wedge \mathtt{opened}(\mathtt{c}_0, \{\mathtt{I}, \mathtt{P}, \mathtt{DB}\}, \mathtt{c}) \wedge \mathtt{DB}^{(1)} \prec_{\mathtt{HB}} \mathtt{DB}^{(2)}.$$

```
//Party(I, c0, (H)|I) * NOTIFY(I(1), ⊖(I(1))) ∧ (H)|All ∧ opened(c0, {I, P, DB}, c)
1 assume(Peer(I));
/*
  Party(I, c0, (H)|I) * NOTIFY(I(1), ⊖(I(1))) * Peer(I) ∧ (H)|All ∧ opened(c0, {I, P, DB}, c)
  ⟩⟩⟩  fire SPLIT LEMMAS  ⟨⟨⟨
  C(c0, I, (H)|I,c0) * NOTIFY(I(1), ⊖(I(1))) * Peer(I) ∧ (H)|All ∧ opened(c0, {I, P, DB}, c) ∧ bind(I, {c0})
  ⟩⟩⟩  unroll (H)|I,c0  ⟨⟨⟨
  C(c0, I, !String; ⊕(I(1)); ⊕(DB(1)≺HBDB(2))) * NOTIFY(I(1), ⊖(I(1)))*Peer(I) ∧ (H)|All∧opened(c0, {I, P, DB}, c)
  */
2 send(c,"EVO~2.5~SATA~III");
/*
  C(c0, I, ⊕(I(1)); ⊕(DB(1)≺HBDB(2))) * NOTIFY(I(1), ⊖(I(1)))*Peer(I) ∧ (H)|All∧opened(c0, {I, P, DB}, c)
  ⟩⟩⟩  fire LEMMA [L+] twice  ⟨⟨⟨
  C(c0, I, emp) * NOTIFY(I(1), ⊖(I(1)))*Peer(I) ∧ (H)|All∧opened(c0, {I, P, DB}, c) ∧ I(1) ∧ DB(1)≺HBDB(2)
*/
3   notifyAll(I(1));
//C(c0, I, emp) * NOTIFY(I(1), emp)*Peer(I) ∧ (H)|All∧opened(c0, {I, P, DB}, c) ∧ I(1) ∧ DB(1)≺HBDB(2)
```

Code of P. The call to `wait` at line 2 in the implementation of P is successful since the event guarded by this call has not occurred yet:

$$\dfrac{\dfrac{S_1 = \{\pi \wedge (\mathtt{E}{=}\mathtt{P}^{(2)}) \wedge (\mathtt{V}{=}(\mathtt{P}^{(2)} \Rightarrow \mathtt{I}^{(1)}{\prec_{\mathtt{HB}}}\mathtt{P}^{(2)}))\}}{\mathtt{WAIT}(\mathtt{P}^{(2)}, \oplus(\mathtt{P}^{(2)} \Rightarrow \mathtt{I}^{(1)}{\prec_{\mathtt{HB}}}\mathtt{P}^{(2)})) \wedge \pi \wedge (\mathtt{E}{=}\mathtt{P}^{(2)}) \vdash^{\kappa}_{\mathtt{Q}} \mathtt{WAIT}(\mathtt{E},\mathtt{V}) \rightsquigarrow S_1.}\ \text{ENT-MATCH} \quad \dfrac{S = S_1 * \Delta}{\Delta * S_1 \vdash^{\kappa}_{\mathtt{Q}} \neg(\mathtt{E}) \rightsquigarrow S.}}{\mathtt{WAIT}(\mathtt{P}^{(2)}, \oplus(\mathtt{P}^{(2)} \Rightarrow \mathtt{I}^{(1)}{\prec_{\mathtt{HB}}}\mathtt{P}^{(2)})) * \Delta \wedge \pi \wedge (\mathtt{E}{=}\mathtt{P}^{(2)}) \vdash^{\kappa}_{\mathtt{Q}} \mathtt{WAIT}(\mathtt{E},\mathtt{V}) \wedge \neg(\mathtt{E}) \rightsquigarrow S.}\ \text{ENT-MATCH}$$

where $\Delta \wedge \pi$ stand for

$$\mathcal{C}(\mathtt{c}_0, \mathtt{P}, !\mathtt{v}\cdot \mathtt{v} > 0; \oplus(\mathtt{P}^{(2)}); \ominus(\mathtt{I}^{(1)}{\prec_{\mathtt{HB}}}\mathtt{P}^{(2)}); \oplus(\mathtt{DB}^{(1)}{\prec_{\mathtt{HB}}}\mathtt{DB}^{(2)})) * \mathtt{Peer}(\mathtt{P}) \wedge$$
$$\wedge(\mathtt{H})\downarrow_{\mathtt{All}} \wedge \mathtt{opened}(\mathtt{c}_0, \{\mathtt{I}, \mathtt{P}, \mathtt{DB}\}, \mathtt{c}) \wedge \mathtt{bind}(\mathtt{P}, \{\mathtt{c}_0\}).$$

Another salient point in the proof of P is the application of $[\underline{\mathtt{L}}\text{–}]$ lemma after line 3. The guard $\ominus(\mathtt{I}^{(1)}{\prec_{\mathtt{HB}}}\mathtt{P}^{(2)})$ in the specification of $\mathtt{c}_0$ holds due to teh following event order constraints subsumed by the abstract state:

$$(\mathtt{P}^{(2)} \Rightarrow \mathtt{I}^{(1)}{\prec_{\mathtt{HB}}}\mathtt{P}^{(2)}) \wedge \mathtt{P}^{(2)}.$$

$//\mathtt{Party}(\mathtt{I}, \mathtt{c}_0, (\mathtt{H})\!\downarrow_{\mathtt{I}}) * \mathtt{WAIT}(\mathtt{P}^{(2)}, \oplus(\mathtt{P}^{(2)} \Rightarrow \mathtt{I}^{(1)} \prec_{\mathtt{HB}} \mathtt{P}^{(2)})) \wedge (\mathtt{H})\!\downarrow_{\mathtt{All}} \wedge \mathtt{opened}(\mathtt{c}_0, \{\mathtt{I}, \mathtt{P}, \mathtt{DB}\}, \mathtt{c})$

1 `assume(Peer(P));`

/*

$\mathtt{Party}(\mathtt{P}, \mathtt{c}_0, (\mathtt{H})\!\downarrow_{\mathtt{P}}) * \mathtt{WAIT}(\mathtt{P}^{(2)}, \oplus(\mathtt{P}^{(2)} \Rightarrow \mathtt{I}^{(1)} \prec_{\mathtt{HB}} \mathtt{P}^{(2)})) * \mathtt{Peer}(\mathtt{P}) \wedge (\mathtt{H})\!\downarrow_{\mathtt{All}} \wedge \mathtt{opened}(\mathtt{c}_0, \{\mathtt{I}, \mathtt{P}, \mathtt{DB}\}, \mathtt{c})$

$\rangle\rangle\rangle$ fire SPLIT LEMMAS $\langle\langle\langle$

$\mathcal{C}(\mathtt{c}_0, \mathtt{P}, (\mathtt{H})\!\downarrow_{\mathtt{I},\mathtt{c}_0}) * \mathtt{WAIT}(\mathtt{P}^{(2)}, \oplus(\mathtt{P}^{(2)} \Rightarrow \mathtt{I}^{(1)} \prec_{\mathtt{HB}} \mathtt{P}^{(2)})) * \mathtt{Peer}(\mathtt{P}) \wedge$
$(\mathtt{H})\!\downarrow_{\mathtt{All}} \wedge \mathtt{opened}(\mathtt{c}_0, \{\mathtt{I}, \mathtt{P}, \mathtt{DB}\}, \mathtt{c}) \wedge \mathtt{bind}(\mathtt{P}, \{\mathtt{c}_0\})$

$\rangle\rangle\rangle$ unroll $(\mathtt{H})\!\downarrow_{\mathtt{P},\mathtt{c}_0}$ $\langle\langle\langle$

$\mathcal{C}(\mathtt{c}_0, \mathtt{P}, !\mathtt{v} \cdot \mathtt{v} > 0; \oplus(\mathtt{P}^{(2)}); \ominus(\mathtt{I}^{(1)} \prec_{\mathtt{HB}} \mathtt{P}^{(2)}); \oplus(\mathtt{DB}^{(1)} \prec_{\mathtt{HB}} \mathtt{DB}^{(2)})) * \mathtt{WAIT}(\mathtt{P}^{(2)}, \oplus(\mathtt{P}^{(2)} \Rightarrow \mathtt{I}^{(1)} \prec_{\mathtt{HB}} \mathtt{P}^{(2)})) *$
$\mathtt{Peer}(\mathtt{P}) \wedge (\mathtt{H})\!\downarrow_{\mathtt{All}} \wedge \mathtt{opened}(\mathtt{c}_0, \{\mathtt{I}, \mathtt{P}, \mathtt{DB}\}, \mathtt{c}) \wedge \mathtt{bind}(\mathtt{P}, \{\mathtt{c}_0\})$

*/

2 `wait(P(2));`

/*

$\mathcal{C}(\mathtt{c}_0, \mathtt{P}, !\mathtt{v} \cdot \mathtt{v} > 0; \oplus(\mathtt{P}^{(2)}); \ominus(\mathtt{I}^{(1)} \prec_{\mathtt{HB}} \mathtt{P}^{(2)}); \oplus(\mathtt{DB}^{(1)} \prec_{\mathtt{HB}} \mathtt{DB}^{(2)})) * \mathtt{WAIT}(\mathtt{P}^{(2)}, \mathtt{emp}) * \mathtt{Peer}(\mathtt{P}) \wedge$
$(\mathtt{H})\!\downarrow_{\mathtt{All}} \wedge \mathtt{opened}(\mathtt{c}_0, \{\mathtt{I}, \mathtt{P}, \mathtt{DB}\}, \mathtt{c}) \wedge \mathtt{bind}(\mathtt{P}, \{\mathtt{c}_0\}) \wedge (\mathtt{P}^{(2)} \Rightarrow \mathtt{I}^{(1)} \prec_{\mathtt{HB}} \mathtt{P}^{(2)})$

*/

3 `send(99);`

/*

$\mathcal{C}(\mathtt{c}_0, \mathtt{P}, \oplus(\mathtt{P}^{(2)}); \ominus(\mathtt{I}^{(1)} \prec_{\mathtt{HB}} \mathtt{P}^{(2)}); \oplus(\mathtt{DB}^{(1)} \prec_{\mathtt{HB}} \mathtt{DB}^{(2)})) * \mathtt{WAIT}(\mathtt{P}^{(2)}, \mathtt{emp}) * \mathtt{Peer}(\mathtt{P}) \wedge$
$(\mathtt{H})\!\downarrow_{\mathtt{All}} \wedge \mathtt{opened}(\mathtt{c}_0, \{\mathtt{I}, \mathtt{P}, \mathtt{DB}\}, \mathtt{c}) \wedge \mathtt{bind}(\mathtt{P}, \{\mathtt{c}_0\}) \wedge (\mathtt{P}^{(2)} \Rightarrow \mathtt{I}^{(1)} \prec_{\mathtt{HB}} \mathtt{P}^{(2)})$

$\rangle\rangle\rangle$ fire LEMMA $[\mathtt{L}+]$ $\langle\langle\langle$

$\mathcal{C}(\mathtt{c}_0, \mathtt{P}, \ominus(\mathtt{I}^{(1)} \prec_{\mathtt{HB}} \mathtt{P}^{(2)}); \oplus(\mathtt{DB}^{(1)} \prec_{\mathtt{HB}} \mathtt{DB}^{(2)})) * \mathtt{WAIT}(\mathtt{P}^{(2)}, \mathtt{emp}) * \mathtt{Peer}(\mathtt{P}) \wedge$
$(\mathtt{H})\!\downarrow_{\mathtt{All}} \wedge \mathtt{opened}(\mathtt{c}_0, \{\mathtt{I}, \mathtt{P}, \mathtt{DB}\}, \mathtt{c}) \wedge \mathtt{bind}(\mathtt{P}, \{\mathtt{c}_0\}) \wedge (\mathtt{P}^{(2)} \Rightarrow \mathtt{I}^{(1)} \prec_{\mathtt{HB}} \mathtt{P}^{(2)}) \wedge \mathtt{P}^{(2)}$

$\rangle\rangle\rangle$ fire LEMMA $[\mathtt{L}-]$ $\langle\langle\langle$

$\mathcal{C}(\mathtt{c}_0, \mathtt{P}, \oplus(\mathtt{DB}^{(1)} \prec_{\mathtt{HB}} \mathtt{DB}^{(2)})) * \mathtt{WAIT}(\mathtt{P}^{(2)}, \mathtt{emp}) * \mathtt{Peer}(\mathtt{P}) \wedge$
$(\mathtt{H})\!\downarrow_{\mathtt{All}} \wedge \mathtt{opened}(\mathtt{c}_0, \{\mathtt{I}, \mathtt{P}, \mathtt{DB}\}, \mathtt{c}) \wedge \mathtt{bind}(\mathtt{P}, \{\mathtt{c}_0\}) \wedge (\mathtt{P}^{(2)} \Rightarrow \mathtt{I}^{(1)} \prec_{\mathtt{HB}} \mathtt{P}^{(2)}) \wedge \mathtt{P}^{(2)}$

$\rangle\rangle\rangle$ fire LEMMA $[\mathtt{L}+]$ $\langle\langle\langle$

$\mathcal{C}(\mathtt{c}_0, \mathtt{P}, \mathtt{emp}) * \mathtt{WAIT}(\mathtt{P}^{(2)}, \mathtt{emp}) * \mathtt{Peer}(\mathtt{P}) \wedge$
$(\mathtt{H})\!\downarrow_{\mathtt{All}} \wedge \mathtt{opened}(\mathtt{c}_0, \{\mathtt{I}, \mathtt{P}, \mathtt{DB}\}, \mathtt{c}) \wedge \mathtt{bind}(\mathtt{P}, \{\mathtt{c}_0\}) \wedge (\mathtt{P}^{(2)} \Rightarrow \mathtt{I}^{(1)} \prec_{\mathtt{HB}} \mathtt{P}^{(2)}) \wedge \mathtt{P}^{(2)} \wedge \mathtt{DB}^{(1)} \prec_{\mathtt{HB}} \mathtt{DB}^{(2)}$

*/

Chapter 6

SOUNDNESS

6.1 Instrumented Operational Semantics

Semantic Model. As depicted in Fig. 6.1, the semantic model of the program state is now instrumented to capture the communication specification as well. The machine configuration is a pair of program state and channel configuration, where a channel configuration captures both a map from program channels to logical channels, as well as a global protocol describing the expected communication. Besides the local thread state and the current program, an instrumented state $\overline{\sigma}_{\mathtt{i}} \in \mathtt{IState}$ also accounts for the role played by the instrumented thread. A thread configuration $\mathtt{TC} \in \mathtt{TConfig}$ is thus defined as:

$$\mathtt{TC} \stackrel{\text{def}}{=} \langle \overline{\sigma}_{\mathtt{i}}, \mathtt{CC} \rangle, \text{ where } \overline{\sigma}_{\mathtt{i}} \stackrel{\text{def}}{=} \langle \sigma, \mathtt{P}, \mathtt{e} \rangle \text{ and } \mathtt{CC} \stackrel{\text{def}}{=} (\mathtt{CS}, \mathtt{G}).$$

For brevity we implicitly assume the existence of a set of function definitions in the program's environment.

Machine Config.	MC	::=	PState × CConfig
Channels Config	CConfig	::=	CStore × G
Channel Store	CStore	::=	$\mathcal{P}$chan⇀$\mathcal{C}$han × $\mathcal{R}$ole*
Program State	PState	::=	idt⇀IState
Instr. Thread State	IState	::=	State × $\mathcal{R}$ole × $\mathcal{P}$
Local State	State	::=	Stack × Heap × Π
Thread Config.	TConfig	::=	IState × CConfig

PS∈PState TC∈TConfig σ∈State $\overline{\sigma}_{\mathtt{i}}$∈IState CS∈CStore CC∈CConfig

Figure 6.1: A Semantic Model of the Core Language

Small-steps Operational Semantics. To prove the soundness of the proposed verification framework we instrument the operational semantics introduced in Sec. 2.7 with communication specifications which could then be related to the specification interpreted by the verifier and offer the necessary aid to identify communication errors in the checked programs. Since a machine cannot run such an instrumented semantics, once we prove the soundness of the verifier, we show how the instrumented semantics is correlated to the semantics of Sec. 2.7.

The small-step operational semantics are defined by the semantic rules in Fig. 6.2, Fig. 6.7 and Fig. 6.8. These semantic rules are defined using the transition relation $\hookrightarrow$ between machine configurations $\langle \mathtt{PS}, \mathtt{CC} \rangle \hookrightarrow \langle \mathtt{PS}', \mathtt{CC}' \rangle$, and between thread configurations $\langle \overline{\sigma}_{\mathtt{i}}, \mathtt{CC} \rangle \hookrightarrow \langle \overline{\sigma}'_{\mathtt{i}}, \mathtt{CC}' \rangle$, respectively. Similar to Sec. 2.7, we use $\hookrightarrow^*$ to denote the transitive closure of the transition relation $\hookrightarrow$.

Fig. 6.2 describes the machine reduction rules:

[iOP–MACHINE] similar to the semantics introduced in Sec. 2.7, the machine quantifies over all kinds of schedulers and propagates the communication effect produced by the current thread to subsequent threads via the channel configuration CC.

[iOP–PAR] splits the heap into disjoint sub-heaps corresponding to each freshly created threads, $\sigma{=}\sigma_1 \uplus \sigma_2$, without affecting the channels' states.

[iOP–JOIN] if both threads have finished their execution, their corresponding resources are transferred to the parent thread, whose local state is update to capture the disjoint union of the two threads states. Since the channels are a shared resource across all threads, each thread sees the same view of the channel store. Should a thread fault, as per [iOP–JOIN–ERR1] or [iOP–JOIN–ERR2], then their fault is propagated through the whole program.

We next distinguish between the possible instantiations of `error`, namely those reduction faults resulted as a consequence of communication errors:

PROT_ERR indicates that the current reduction refers to a transmission which is not expected within the communication protocol.

RACE_ERR is raised when the $\mathtt{S}_{\mathtt{safe}}$ or $\mathtt{R}_{\mathtt{safe}}$ identify a sending or receiving race condition.

RES_ERR is raised when the communicating thread refers to resources it does not owe.

$$\frac{\langle \overline{\sigma}_{\mathtt{i}}, \mathtt{CC}\rangle \hookrightarrow^* \langle \overline{\sigma}'_{\mathtt{i}}, \mathtt{CC}'\rangle}{\langle \mathtt{PS}[t \mapsto \overline{\sigma}_{\mathtt{i}}], \mathtt{CC}\rangle \hookrightarrow \langle \mathtt{PS}[t \mapsto \overline{\sigma}'_{\mathtt{i}}], \mathtt{CC}'\rangle} \; [\mathbf{iOP\text{–}MACHINE}]$$

$$\frac{\overline{\sigma}_{\mathtt{i}} = \langle \sigma, \mathtt{P}, (\mathtt{e}_1 || \mathtt{e}_2)\rangle \quad \textit{fresh}\ t_1, t_2 \quad \overline{\sigma}'_{\mathtt{i}} := \langle \mathtt{emp}_{\mathtt{sh}}, \mathtt{P}, (\mathtt{join}\ t_1\ t_2)\rangle \quad \sigma = \sigma_1 \uplus \sigma_2 \quad \mathtt{PS}' := \mathtt{PS}[t \mapsto \overline{\sigma}'_{\mathtt{i}}][t_1 \mapsto \langle \sigma_1, \mathtt{P}, \mathtt{e}_1\rangle][t_2 \mapsto \langle \sigma_2, \mathtt{P}, \mathtt{e}_2\rangle]}{\langle \mathtt{PS}[t \mapsto \overline{\sigma}_{\mathtt{i}}], \mathtt{CC}\rangle \hookrightarrow \langle \mathtt{PS}', \mathtt{CC}\rangle} \; [\mathbf{iOP\text{–}PAR}]$$

$$\frac{\overline{\sigma}_{\mathtt{i}} = \langle \sigma, \mathtt{P}, \mathtt{join}\ t_1\ t_2\rangle \quad \mathtt{PS} = \mathtt{PS}'[t_1 \mapsto \langle \sigma_1, \_, \mathtt{skip}\rangle][t_2 \mapsto \langle \sigma_2, \_, \mathtt{skip}\rangle] \quad \sigma' := \sigma \uplus \sigma_1 \uplus \sigma_2}{\langle \mathtt{PS}[t \mapsto \overline{\sigma}_{\mathtt{i}}], \mathtt{CC}\rangle \hookrightarrow \langle \mathtt{PS}'[t \mapsto \langle \sigma', \mathtt{P}, \mathtt{skip}\rangle], \mathtt{CC}\rangle} \; [\mathbf{iOP\text{–}JOIN}]$$

$$\frac{\overline{\sigma}_{\mathtt{i}} = \langle \_, \_, \mathtt{join}\ t_1\ t_2\rangle \quad \mathtt{PS} = \mathtt{PS}'[t_1 \mapsto \langle \_, \_, \mathtt{skip}\rangle][t_2 \mapsto \langle \_, \_, \mathtt{e}\rangle]}{\langle \mathtt{PS}[t \mapsto \overline{\sigma}_{\mathtt{i}}], \mathtt{CC}\rangle \hookrightarrow \langle \mathtt{PS}[t \mapsto \overline{\sigma}_{\mathtt{i}}], \mathtt{CC}\rangle} \; [\mathbf{iOP\text{–}JOIN\text{–}SKIP1}]$$

$$\frac{\overline{\sigma}_{\mathtt{i}} = \langle \_, \_, \mathtt{join}\ t_1\ t_2\rangle \quad \mathtt{PS} = \mathtt{PS}'[t_1 \mapsto \langle \_, \_, \mathtt{e}\rangle][t_2 \mapsto \langle \_, \_, \mathtt{skip}\rangle]}{\langle \mathtt{PS}[t \mapsto \overline{\sigma}_{\mathtt{i}}], \mathtt{CC}\rangle \hookrightarrow \langle \mathtt{PS}[t \mapsto \overline{\sigma}_{\mathtt{i}}], \mathtt{CC}\rangle} \; [\mathbf{iOP\text{–}JOIN\text{–}SKIP2}]$$

$$\frac{\overline{\sigma}_{\mathtt{i}} = \langle \_, \_, \mathtt{join}\ t_1\ t_2\rangle \quad \mathtt{PS} = \mathtt{PS}'[t_1 \mapsto \langle \_, \_, \mathtt{e}\rangle][t_2 \mapsto \mathtt{error}]}{\langle \mathtt{PS}[t \mapsto \overline{\sigma}_{\mathtt{i}}], \mathtt{CC}\rangle \hookrightarrow \mathtt{error}} \; [\mathbf{iOP\text{–}JOIN\text{–}ERR1}]$$

$$\frac{\overline{\sigma}_{\mathtt{i}} = \langle \_, \_, \mathtt{join}\ t_1\ t_2\rangle \quad \mathtt{PS} = \mathtt{PS}'[t_1 \mapsto \mathtt{error}][t_2 \mapsto \langle \_, \_, \mathtt{e}\rangle]}{\langle \mathtt{PS}[t \mapsto \overline{\sigma}_{\mathtt{i}}], \mathtt{CC}\rangle \hookrightarrow \mathtt{error}} \; [\mathbf{iOP\text{–}JOIN\text{–}ERR2}]$$

Figure 6.2: Instrumented Semantic Rules: Machine Reduction

$$
\begin{array}{ll}
\texttt{EMPTY}(\texttt{P}^*, (\texttt{S} \xrightarrow{i} \texttt{R} : \_)) & \triangleq \forall \texttt{P} \in \texttt{P}^* \Rightarrow \texttt{P} \notin \{\texttt{S}, \texttt{R}\} \\
\texttt{EMPTY}(\texttt{P}^*, \texttt{Z}_1; \texttt{Z}_2) & \triangleq \texttt{EMPTY}(\texttt{P}^*, \texttt{Z}_1) \wedge \texttt{EMPTY}(\texttt{P}^*, \texttt{Z}_2) \\
\texttt{EMPTY}(\texttt{P}^*, \texttt{Z}_1 \vee \texttt{Z}_2) & \triangleq \texttt{EMPTY}(\texttt{P}^*, \texttt{Z}_1) \vee \texttt{EMPTY}(\texttt{P}^*, \texttt{Z}_2)
\end{array}
$$

Figure 6.3: Safety Check: Leak-free

$$
\frac{\texttt{CC} = \langle \texttt{CS}, \texttt{G} \rangle \qquad \texttt{CC}' := \langle \texttt{CS}[\texttt{res} \mapsto (\texttt{c}, \texttt{P}^*)], \texttt{G} \rangle}{\langle \langle \sigma, \_, \texttt{open}() \texttt{ with } (\texttt{c}, \texttt{P}^*) \rangle, \texttt{CC} \rangle \hookrightarrow \langle \langle \sigma, \_, \texttt{skip} \rangle, \texttt{CC}' \rangle} \; [\textbf{iOP-OPEN}]
$$

$$
\frac{\texttt{CC} = \langle \texttt{CS}'[\tilde{\texttt{c}} \mapsto (\texttt{c}, \texttt{P}^*)], \texttt{G} \rangle \qquad \texttt{CC}' := \langle \texttt{CS}', \texttt{G} \rangle \qquad \texttt{EMPTY}((\texttt{G})\lfloor_{\texttt{c}}, \texttt{P}^*)}{\langle \langle \sigma, \_, \texttt{close}(\tilde{\texttt{c}}) \rangle, \texttt{CC} \rangle \hookrightarrow \langle \langle \sigma', \_, \texttt{skip} \rangle, \texttt{CC}' \rangle} \; [\textbf{iOP-CLOSE}]
$$

$$
\frac{\texttt{CC} = \langle \texttt{CS}'[\tilde{\texttt{c}} \mapsto (\texttt{c}, \texttt{P}^*)], \texttt{G} \rangle \qquad \neg(\texttt{EMPTY}((\texttt{G})\lfloor_{\texttt{c}}, \texttt{P}^*))}{\langle \langle \sigma, \_, \texttt{close}(\tilde{\texttt{c}}) \rangle, \texttt{CC} \rangle \hookrightarrow \texttt{LEAK\_ERR}} \; [\textbf{iOP-CLOSE-ELEAK}]
$$

Figure 6.4: Instrumented Semantic Rules: Channel Manipulation

LEAK_ERR indicates a possible data leak towards unintended recipients.

The expressions which manipulate the channels are given a semantic via the rules in Fig. 6.4:

[iOP-OPEN] opens a new channel mapped to the logical channel c and used by the logical peers $\texttt{P}^*$ within the protocol G.

[iOP-CLOSE] removes the logical channel and peers dependency for the program channel $\tilde{\texttt{c}}$, but not before checking that there are no further transmissions expected to be consumed over the corresponding logical channel, c. The check is done via a specially crafted predicate EMPTY, defined in Fig. 6.3 which receives as arguments the logical peers using $\tilde{\texttt{c}}$ and the channel projection which describes the communication over c. The predicate holds only when there are no more references to $\texttt{P}^*$ in the projection of G w.r.t. c, that is in $(\texttt{G})\lfloor_{\texttt{c}}$. The special case of communication choice checks that at least one of the choices contains no reference $\texttt{P}^*$. If this check fails, the reduction also faults with a LEAK_ERR indicating that there are peers who are still expected to send or receive messages via c.

To test for protocol conformance we define a predicate, $\texttt{FID}(\texttt{c}, \texttt{P}, \texttt{G}) \triangleq (\texttt{safe}, m, \texttt{G}')$, which checks whether the current transmission is correctly captured within the global protocol. This

$$
\begin{array}{ll}
\texttt{FID}(\texttt{c},\texttt{P},(\texttt{S}\xrightarrow{i}\texttt{R}:\texttt{c}_0\langle\texttt{v}\cdot\Delta\rangle;\texttt{G}^\texttt{t})) & \triangleq\ (\texttt{SAFE},\ \texttt{v}\cdot\Delta,\ \texttt{HOLE}\xrightarrow{i}\texttt{R}:\texttt{c}_0\langle\texttt{v}\cdot\Delta\rangle;\texttt{G}^\texttt{t}) \text{ when } \texttt{c}_0{=}\texttt{c}\ \wedge\ \texttt{S}{=}\texttt{P} \\
\texttt{FID}(\texttt{c},\texttt{P},(\texttt{S}\xrightarrow{i}\texttt{R}:\texttt{c}_0\langle\texttt{v}\cdot\Delta\rangle;\texttt{G}^\texttt{t})) & \triangleq\ (\texttt{SAFE},\ \texttt{v}\cdot\Delta,\ \texttt{S}\xrightarrow{i}\texttt{HOLE}:\texttt{c}_0\langle\texttt{v}\cdot\Delta\rangle;\texttt{G}^\texttt{t}) \text{ when } \texttt{c}_0{=}\texttt{c}\ \wedge\ \texttt{R}{=}\texttt{P} \\
\texttt{FID}(\texttt{c},\texttt{P},(\texttt{S}\xrightarrow{i}\texttt{R}:\texttt{c}_0\langle\texttt{v}\cdot\Delta\rangle;\texttt{G}^\texttt{t})) & \triangleq\ (\texttt{ANS},\ m,\ \texttt{S}\xrightarrow{i}\texttt{R}:\texttt{c}_0\langle\texttt{v}\cdot\Delta\rangle;\texttt{G}') \text{ when } \texttt{P}{\notin}\{\texttt{S},\ \texttt{R}\}\ \wedge\ \texttt{c}{\neq}\texttt{c}_0 \\
 & \qquad\qquad \wedge\ (\texttt{ANS},\ m,\ \texttt{G}'){=}\texttt{FID}(\texttt{c},\texttt{P},\texttt{G}^\texttt{t}) \\
\texttt{FID}(\texttt{c},\texttt{P},(\texttt{G}_1\vee\texttt{G}_2);\texttt{G}^\texttt{t}) & \triangleq\ (\texttt{ANS},\ m,\ (\texttt{G}'\vee\texttt{G}_2);\texttt{G}^\texttt{t}) \text{ when } (\texttt{ANS},\ m,\ \texttt{G}'){=}\texttt{FID}(\texttt{c},\texttt{P},\texttt{G}_1) \\
 & \qquad\qquad \wedge\ (\texttt{FAIL},\ \_,\ \_){=}\texttt{FID}(\texttt{c},\texttt{P},\texttt{G}_2) \\
 & \text{or } (\texttt{ANS},\ m,\ (\texttt{G}_1\vee\texttt{G}');\texttt{G}^\texttt{t}) \text{ when } (\texttt{FAIL},\ \_,\ \_){=}\texttt{FID}(\texttt{c},\texttt{P},\texttt{G}_1) \\
 & \qquad\qquad \wedge\ (\texttt{ANS},\ m,\ \texttt{G}'){=}\texttt{FID}(\texttt{c},\texttt{P},\texttt{G}_2) \\
\texttt{FID}(\texttt{c},\texttt{P},(\texttt{G}_1*\texttt{G}_2);\texttt{G}^\texttt{t}) & \triangleq\ (\texttt{ANS},\ m,\ (\texttt{G}'*\texttt{G}_2);\texttt{G}^\texttt{t}) \text{ when } (\texttt{ANS},\ m,\ \texttt{G}'){=}\texttt{FID}(\texttt{c},\texttt{P},\texttt{G}_1) \\
 & \qquad\qquad \wedge\ (\texttt{FAIL},\ \_,\ \_){=}\texttt{FID}(\texttt{c},\texttt{P},\texttt{G}_2) \\
 & \text{or } (\texttt{ANS},\ m,\ (\texttt{G}_1*\texttt{G}');\texttt{G}^\texttt{t}) \text{ when } (\texttt{FAIL},\ \_,\ \_){=}\texttt{FID}(\texttt{c},\texttt{P},\texttt{G}_1) \\
 & \qquad\qquad \wedge\ (\texttt{ANS},\ m,\ \texttt{G}'){=}\texttt{FID}(\texttt{c},\texttt{P},\texttt{G}_2)
\end{array}
$$

Figure 6.5: Safety Check: Protocol Conformance

predicate takes as input a logical channel $\texttt{c}$ which corresponds to the channel which performs the current transmission, a logical party $\texttt{P}$ corresponding to the thread which is performing the transmission and the current state of the communication protocol $\texttt{G}$. The predicate then encapsulates the safety of the transmission, $\texttt{safe} \in \{\texttt{SAFE}, \texttt{FAIL}\}$ as well as the logical description of the transferred message and an updated communication protocol. To capture that a transmission has been consumed, the corresponding transmission is replaced by a $\texttt{HOLE}$ in the updated global protocol. The $\texttt{FID}$ predicate is inductively defined over the structure of protocol $\texttt{G}$, as depicted in Fig. 6.5: the first two cases correspond to a send and receive, respectively, while the third case permits a transmission to overtake the head of the protocol should the current transmission be consumed on a different sender/receiver and channel than expected. The cases for choice and parallel communication emphasize the non-deterministic character of this approach where the current transmission can safely be consumed only by strictly one branch.

To force the machine to fault when it encounters a communication race, we define the predicates $\texttt{S}_{\texttt{safe}}$ and $\texttt{R}_{\texttt{safe}}$ to check for send or receive race, respectively. The predicates expect a logical peer - corresponding to the role played by the current thread - and a channel specification as arguments. The channel specification is obtained by dynamically projecting the global protocol onto a logical channel as per the projection rules described in Fig. 4.6a. The holes of the global protocol are ignored during the projection. The predicates are then recursively defined on the structure of the channel specification. Since sending is asynchronous, $\texttt{S}_{\texttt{safe}}$ also ensures that the program order and the communication protocol agree on the order of transmissions ($\texttt{P}{\neq}\texttt{R}$ in

$$
\begin{array}{ll}
\mathtt{S}_{\mathtt{safe}}(\mathtt{P}, (\mathtt{S} \xrightarrow{i} \mathtt{R} : \mathtt{v}{\cdot}\Delta; \mathtt{Z}^{\mathtt{t}})) & \triangleq \ \mathtt{SAFE} \text{ when } \mathtt{P}{=}\mathtt{S} \\
\mathtt{S}_{\mathtt{safe}}(\mathtt{P}, (\mathtt{HOLE} \xrightarrow{i} \mathtt{R} : \mathtt{v}{\cdot}\Delta; \mathtt{Z}^{\mathtt{t}})) & \triangleq \ \mathtt{S}_{\mathtt{safe}}(\mathtt{P}, \mathtt{Z}^{\mathtt{t}}) \text{ when } \mathtt{P}{\neq}\mathtt{R} \\
\mathtt{S}_{\mathtt{safe}}(\mathtt{P}, ((\mathtt{Z}_1 \vee \mathtt{Z}_2); \mathtt{Z}^{\mathtt{t}})) & \triangleq \ \mathtt{S}_{\mathtt{safe}}(\mathtt{P}, \mathtt{Z}_1) \text{ when } \mathtt{FAIL}{=}\mathtt{S}_{\mathtt{safe}}(\mathtt{P}, \mathtt{Z}_2) \\
 & \quad \text{or } \ \mathtt{S}_{\mathtt{safe}}(\mathtt{P}, \mathtt{Z}_2) \text{ when } \mathtt{FAIL}{=}\mathtt{S}_{\mathtt{safe}}(\mathtt{P}, \mathtt{Z}_1) \\
 & \\
\mathtt{R}_{\mathtt{safe}}(\mathtt{P}, (\mathtt{HOLE} \xrightarrow{i} \mathtt{R} : \mathtt{v}{\cdot}\Delta; \mathtt{Z}^{\mathtt{t}})) & \triangleq \ \mathtt{SAFE} \text{ when } \mathtt{P}{=}\mathtt{R} \\
\mathtt{R}_{\mathtt{safe}}(\mathtt{P}, (\mathtt{S} \xrightarrow{i} \mathtt{R} : \mathtt{v}{\cdot}\Delta; \mathtt{Z}^{\mathtt{t}})) & \triangleq \ \mathtt{BLOCK} \text{ when } \mathtt{P}{=}\mathtt{R} \\
\mathtt{R}_{\mathtt{safe}}(\mathtt{P}, (\mathtt{HOLE} \xrightarrow{i} \mathtt{HOLE} : \_; \mathtt{Z}^{\mathtt{t}})) & \triangleq \ \mathtt{R}_{\mathtt{safe}}(\mathtt{P}, \mathtt{Z}^{\mathtt{t}}) \\
\mathtt{R}_{\mathtt{safe}}(\mathtt{P}, ((\mathtt{Z}_1 \vee \mathtt{Z}_2); \mathtt{Z}^{\mathtt{t}})) & \triangleq \ \mathtt{R}_{\mathtt{safe}}(\mathtt{P}, \mathtt{Z}_1) \text{ when } \mathtt{FAIL}{=}\mathtt{R}_{\mathtt{safe}}(\mathtt{P}, \mathtt{Z}_2) \\
 & \quad \text{or } \ \mathtt{R}_{\mathtt{safe}}(\mathtt{P}, \mathtt{Z}_2) \text{ when } \mathtt{FAIL}{=}\mathtt{R}_{\mathtt{safe}}(\mathtt{P}, \mathtt{Z}_1)
\end{array}
$$

Figure 6.6: Safety Checks: Race-free

the second case). $\mathtt{S}_{\mathtt{safe}}$ is defined in terms of $\mathtt{safe} \in \{\mathtt{SAFE}, \mathtt{FAIL}\}$, while $\mathtt{R}_{\mathtt{safe}}$ also captures the blocking behavior of the receive, $\mathtt{safe} \in \{\mathtt{SAFE}, \mathtt{FAIL}, \mathtt{BLOCK}\}$.

Any case which is not explicitly captured by FID, $\mathtt{S}_{\mathtt{safe}}$ and $\mathtt{R}_{\mathtt{safe}}$ is deemed to fail.

The communication related expressions are given a semantic via the rules in Fig. 6.7:

[iOP–SEND] a fault-free send suggests that there is a map from the program channel $\tilde{\mathtt{c}}$ used for the send operation to a logical channel c and a set of logical peers $\mathtt{P}^*$ using it. Furthermore, the role P played by the current thread exists within the set of logical peers that $\tilde{\mathtt{c}}$ is mapped to: $\mathtt{P} \in \mathtt{P}^*$. Moreover, this operation should create no race, $\mathtt{SAFE} = \mathtt{S}_{\mathtt{safe}}(\mathtt{P}, (\mathtt{G})\lfloor_{\mathtt{c}})$, w.r.t. the channel specification, and it should also be in agreement with the global protocol, $(\mathtt{SAFE}, \mathtt{v}'{\cdot}\Delta, \mathtt{G}') = \mathtt{FID}(\mathtt{G}, \mathtt{c}, \mathtt{P})$. The transmitted message referenced by v in the program code (and by $\mathtt{v}'$ in the global protocol) is described by the logical formula Δ and it should be owned by the thread who is performing the transmission: $\sigma_{\mathtt{v}} \vDash [\mathtt{v}/\mathtt{v}']\Delta$, where the local thread state is described by $\sigma{=}\sigma' \uplus \sigma_{\mathtt{v}}$, and updated to the remaining state σ' after losing ownership of the transmitted message. If $\mathtt{S}_{\mathtt{safe}}$ does not hold the reduction faults with a RACE_ERR error, if FID does not hold it faults with PROT_ERR and finally, if the ownership of the transmitted resource cannot be verified, it faults with RES_ERR as per [iOP–SEND–ERES].

[iOP–RECV] similar to a send operation, a fault-free receive suggests that there is a map from the program channel $\tilde{\mathtt{c}}$ used for fetching messages to a logical channel c, and a set of logical peers $\mathtt{P}^*$ which share $\tilde{\mathtt{c}}$. The check that the current thread's role P indeed exists within the set of logical peers sharing $\tilde{\mathtt{c}}$ is an explicit premise of this reduction rule:

$\mathtt{P} \in \mathtt{P}^*$. If this transmission does not race with another receive, and it is also specified by $\mathtt{G}$, then the current thread state is updated to account for the received message $\sigma' := \sigma \uplus \sigma_{\mathtt{res}}$, where σ' is the updated state, σ is the current state and $\sigma_{\mathtt{res}}$ is the witness state indicating that the current thread gained ownership over the received message. If $\mathtt{R}_{\mathtt{safe}}$ does not hold then the reduction faults with `RACE_ERR`, and if `FID` fails then the reduction faults with `PROT_ERR`. If the message to be read has not yet been transmitted, the thread does not progress, remaining in the same state until the corresponding send is consumed.

Finally, the [iOP−ASSERT−PEER] in Fig. 6.8 updates the configuration of a thread to indicate the role it plays.

6.2 Soundness

This section aims to prove the soundness of the proposed multiparty solution with respect to the given small-step operational semantics by proving progress and preservation. But before proving soundness, it is worth discussing the interference and locality issues addressed by the concurrency formalism approaches in general, and by separation logic based approaches in particular.

Interference. We stress on the fact that, for clarity, the current thesis highlights the effects explicit synchronization has strictly over the communication. The effects that explicit synchronization has over the local heap are orthogonal issue tackled by works such as [5, 84, 32]. This explains the semantic choice for the programming model of Fig. 6.1 which separates the resources owned by a thread from those owned by a communication channel, and where the environment interference only affects the state of communication and not the local state.

Locality. Assuming that each message is characterized by a precise formula, each time a thread performs a read it acquires the resource ownership of exactly that heap portion needed to satisfy the formula corresponding to the received message. Similarly, on sending a message the thread releases a resource, i.e. it transfers the ownership of exactly that heap portion determined by the message formula. Since the formulae describing the messages are precise, a transmission can only modify the state of the local heap in one way, releasing or acquiring the resource which is being transmitted, or in other words there is only one possible local transmission which is safe, as highlighted in Fig. 6.7.

Moreover, a parallel computation $\mathtt{e}_1 \| \mathtt{e}_2$ can be decomposed into the local computations $\mathtt{e}_1$

$$\frac{(\mathtt{CS},\mathtt{G})=\mathtt{CC} \quad \mathtt{c},\mathtt{P}^* = \mathtt{CS}(\tilde{\mathtt{c}}) \quad \mathtt{P}\in\mathtt{P}^* \quad \mathtt{SAFE} = \mathtt{S}_{\mathtt{safe}}(\mathtt{P},(\mathtt{G})\lfloor_{\mathtt{c}}) \quad (\mathtt{SAFE},\mathtt{v}{\cdot}\Delta,\mathtt{G}') = \mathtt{FID}(\mathtt{G},\mathtt{c},\mathtt{P}) \quad \mathtt{CC}' := (\mathtt{CS},\mathtt{G}') \quad \sigma{=}\sigma'{\uplus}\sigma_{\mathtt{x}} \quad \sigma_{\mathtt{x}} \vDash [\mathtt{x}/\mathtt{v}]\Delta}{\langle\langle\sigma,\mathtt{P},\mathtt{send}(\tilde{\mathtt{c}},\mathtt{x})\rangle,\mathtt{CC}\rangle \hookrightarrow \langle\langle\sigma',\mathtt{P},\mathtt{skip}\rangle,\mathtt{CC}'\rangle} \;[\mathbf{iOP{-}SEND}]$$

$$\frac{(\mathtt{CS},\mathtt{G})=\mathtt{CC} \quad \mathtt{c},\mathtt{P}^* = \mathtt{CS}(\tilde{\mathtt{c}}) \quad \mathtt{P}\in\mathtt{P}^* \quad \mathtt{FAIL} = \mathtt{S}_{\mathtt{safe}}(\mathtt{P},(\mathtt{G})\lfloor_{\mathtt{c}})}{\langle\langle\sigma,\mathtt{P},\mathtt{send}(\tilde{\mathtt{c}},\mathtt{x})\rangle,\mathtt{CC}\rangle \hookrightarrow \mathtt{RACE\_ERR}} \;[\mathbf{iOP{-}SEND{-}ERACE}]$$

$$\frac{(\mathtt{CS},\mathtt{G})=\mathtt{CC} \quad \mathtt{c},\mathtt{P}^* = \mathtt{CS}(\tilde{\mathtt{c}}) \quad \mathtt{P}\in\mathtt{P}^* \quad (\mathtt{FAIL},\_,\_) = \mathtt{FID}(\mathtt{G},\mathtt{c},\mathtt{P})}{\langle\langle\sigma,\mathtt{P},\mathtt{send}(\tilde{\mathtt{c}},\mathtt{x})\rangle,\mathtt{CC}\rangle \hookrightarrow \mathtt{PROT\_ERR}} \;[\mathbf{iOP{-}SEND{-}EPROT}]$$

$$\frac{(\mathtt{CS},\mathtt{G})=\mathtt{CC} \quad \mathtt{c},\mathtt{P}^* = \mathtt{CS}(\tilde{\mathtt{c}}) \quad \mathtt{P}\in\mathtt{P}^* \quad \mathtt{SAFE} = \mathtt{S}_{\mathtt{safe}}(\mathtt{P},(\mathtt{G})\lfloor_{\mathtt{c}}) \quad (\mathtt{SAFE},\mathtt{v}{\cdot}\Delta,\_) = \mathtt{FID}(\mathtt{G},\mathtt{c},\mathtt{P}) \quad \neg(\exists\sigma_{\mathtt{x}},\sigma'\cdot(\sigma{=}\sigma'{\uplus}\sigma_{\mathtt{x}})\wedge(\sigma_{\mathtt{x}}\vDash[\mathtt{x}/\mathtt{v}]\Delta))}{\langle\langle\sigma,\mathtt{P},\mathtt{send}(\tilde{\mathtt{c}},\mathtt{x})\rangle,\mathtt{CC}\rangle \hookrightarrow \mathtt{RES\_ERR}} \;[\mathbf{iOP{-}SEND{-}ERES}]$$

$$\frac{(\mathtt{CS},\mathtt{G})=\mathtt{CC} \quad \mathtt{c},\mathtt{P}^* = \mathtt{CS}(\tilde{\mathtt{c}}) \quad \mathtt{P}\in\mathtt{P}^* \quad \mathtt{SAFE} = \mathtt{R}_{\mathtt{safe}}(\mathtt{P},(\mathtt{G})\lfloor_{\mathtt{c}}) \quad (\mathtt{SAFE},\mathtt{v}{\cdot}\Delta,\mathtt{G}') = \mathtt{FID}(\mathtt{G},\mathtt{c},\mathtt{P}) \quad \mathtt{CC}' := (\mathtt{CS},\mathtt{G}') \quad \sigma' {:=} \sigma{\uplus}\sigma_{\mathtt{res}} \quad \sigma_{\mathtt{res}} \vDash [\mathtt{res}/\mathtt{v}]\Delta}{\langle\langle\sigma,\mathtt{P},\mathtt{recv}(\tilde{\mathtt{c}})\rangle,\mathtt{CC}\rangle \hookrightarrow \langle\langle\sigma',\mathtt{P},\mathtt{skip}\rangle,\mathtt{CC}'\rangle} \;[\mathbf{iOP{-}RECV}]$$

$$\frac{(\mathtt{CS},\mathtt{G})=\mathtt{CC} \quad \mathtt{c},\mathtt{P}^* = \mathtt{CS}(\tilde{\mathtt{c}}) \quad \mathtt{P}\in\mathtt{P}^* \quad \mathtt{FAIL} = \mathtt{R}_{\mathtt{safe}}(\mathtt{P},(\mathtt{G})\lfloor_{\mathtt{c}})}{\langle\langle\sigma,\mathtt{P},\mathtt{recv}(\tilde{\mathtt{c}})\rangle,\mathtt{CC}\rangle \hookrightarrow \mathtt{RACE\_ERR}} \;[\mathbf{iOP{-}RECV{-}ERACE}]$$

$$\frac{(\mathtt{CS},\mathtt{G})=\mathtt{CC} \quad \mathtt{c},\mathtt{P}^* = \mathtt{CS}(\tilde{\mathtt{c}}) \quad \mathtt{P}\in\mathtt{P}^* \quad (\mathtt{FAIL},\_,\_) = \mathtt{FID}(\mathtt{G},\mathtt{c},\mathtt{P})}{\langle\langle\sigma,\mathtt{P},\mathtt{recv}(\tilde{\mathtt{c}})\rangle,\mathtt{CC}\rangle \hookrightarrow \mathtt{PROT\_ERR}} \;[\mathbf{iOP{-}RECV{-}EPROT}]$$

$$\frac{(\mathtt{CS},\mathtt{G})=\mathtt{CC} \quad \mathtt{c},\mathtt{P}^* = \mathtt{CS}(\tilde{\mathtt{c}}) \quad \mathtt{P}\in\mathtt{P}^* \quad \mathtt{BLOCK} = \mathtt{R}_{\mathtt{safe}}(\mathtt{P},(\mathtt{G})\lfloor_{\mathtt{c}})}{\langle\langle\sigma,\mathtt{P},\mathtt{recv}(\tilde{\mathtt{c}})\rangle,\mathtt{CC}\rangle \hookrightarrow \langle\langle\sigma,\mathtt{P},\mathtt{recv}(\tilde{\mathtt{c}})\rangle,\mathtt{CC}\rangle} \;[\mathbf{iOP{-}RECV{-}BLOCK}]$$

Figure 6.7: Instrumented Semantic Rules: Per-Thread Reduction

$$\frac{}{\langle\langle\sigma,\_,\mathtt{assert\ Peer}(\mathtt{P})\rangle,\mathtt{CC}\rangle \hookrightarrow \langle\langle\sigma,\mathtt{P},\mathtt{skip}\rangle,\mathtt{CC}\rangle} \;[\mathbf{iOP{-}ASSERT{-}PEER}]$$

$$\frac{(\mathtt{CS},\_) = \mathtt{CC} \quad \mathtt{CC}' := (\mathtt{CS},\mathtt{G})}{\langle\langle\sigma,\_,\mathtt{assert\ G}(\{\mathtt{P}_1..\mathtt{P}_n\},\mathtt{c}^*)\rangle,\mathtt{CC}\rangle \hookrightarrow \langle\langle\sigma,\mathtt{P},\mathtt{skip}\rangle,\mathtt{CC}'\rangle} \;[\mathbf{iOP{-}ASSERT{-}PROT}]$$

Figure 6.8: Instrumented Semantic Rules: Ghost Transition

and $\mathtt{e}_2$ which are interference-free, except for the communication related interactions carefully guided by a global protocol.

Definition 14 (Compatible instrumented states)**.** *Two instrumented states* $\overline{\sigma}^1_{\mathtt{i}}, \overline{\sigma}^2_{\mathtt{i}} \in \mathtt{IState}$ *are compatible, written* $\mathtt{comp}(\overline{\sigma}^1_{\mathtt{i}}, \overline{\sigma}^2_{\mathtt{i}})$*, if and only if there exists* $\mathtt{P} \in \mathcal{R}\mathtt{ole},\ \mathtt{e}_1, \mathtt{e}_2 \in \mathcal{P}$*, and* $\sigma_1, \sigma_2 \in \mathtt{TState}$ *such that* $\overline{\sigma}^1_{\mathtt{i}} = \langle \sigma_1, \mathtt{P}, \mathtt{e}_1 \rangle$ *and* $\overline{\sigma}^2_{\mathtt{i}} = \langle \sigma_2, \mathtt{P}, \mathtt{e}_2 \rangle$ *and* $\sigma_1 \perp \sigma_2$ *and either* $\mathtt{e}_1 = \mathsf{skip}$ *or* $\mathtt{e}_2 = \mathsf{skip}$*.*

Definition 15 (Composition of instrumented states)**.** *The composition of two instrumented states* $\overline{\sigma}^1_{\mathtt{i}}, \overline{\sigma}^2_{\mathtt{i}} \in \mathtt{IState}$*, where* $\mathtt{comp}(\overline{\sigma}^1_{\mathtt{i}}, \overline{\sigma}^2_{\mathtt{i}})$ *and* $\overline{\sigma}^1_{\mathtt{i}} = \langle \sigma_1, \mathtt{P}, \mathtt{e} \rangle$ *and* $\overline{\sigma}^2_{\mathtt{i}} = \langle \sigma_2, \mathtt{P}, \mathsf{skip} \rangle$ *is defined as:*

$$\overline{\sigma}^1_{\mathtt{i}} \uplus \overline{\sigma}^2_{\mathtt{i}} \stackrel{\text{def}}{=} \langle \sigma_1 \uplus \sigma_2, \mathtt{P}, \mathtt{e} \rangle$$

with $\overline{\sigma}^1_{\mathtt{i}} \uplus \overline{\sigma}^2_{\mathtt{i}} \in \mathtt{IState}$ *.*

Lemma 2. *For any two instrumented states* $\overline{\sigma}^1_{\mathtt{i}}, \overline{\sigma}^2_{\mathtt{i}} \in \mathtt{IState}$*, where* $\mathtt{comp}(\overline{\sigma}^1_{\mathtt{i}}, \overline{\sigma}^2_{\mathtt{i}})$ *and* $\overline{\sigma}^1_{\mathtt{i}} \uplus \overline{\sigma}^2_{\mathtt{i}} \in \mathtt{IState}$*, there exists* $\mathtt{CC} \in \mathtt{CConfig}$ *such that:*

(1) $\langle \overline{\sigma}^1_{\mathtt{i}} \uplus \overline{\sigma}^2_{\mathtt{i}}, \mathtt{CC} \rangle \hookrightarrow^* \mathtt{error} \ \Rightarrow\ \overline{\sigma}^1_{\mathtt{i}} \hookrightarrow \mathtt{error}$

(2) $\langle \overline{\sigma}^1_{\mathtt{i}} \uplus \overline{\sigma}^2_{\mathtt{i}}, \mathtt{CC} \rangle \hookrightarrow^* \langle \overline{\sigma}'_{\mathtt{i}}, \mathtt{CC}' \rangle \ \Rightarrow\ \exists \overline{\sigma}^0_{\mathtt{i}} \in \mathtt{IState} \cdot \overline{\sigma}'_{\mathtt{i}} = \overline{\sigma}^0_{\mathtt{i}} \uplus \overline{\sigma}^2_{\mathtt{i}}\ \mathtt{and}\ \langle \overline{\sigma}^1_{\mathtt{i}}, \mathtt{CC} \rangle \hookrightarrow^* \langle \overline{\sigma}^0_{\mathtt{i}}, \mathtt{CC}' \rangle.$

Proof: Proving (1) is straightforward. Proving (2) requires induction on the length of the derivation and a case analysis on $\mathtt{e}$ which is similar to the standard locality principle of separation logic [5], with the extra judgement on $\mathtt{CC}$. □

Definition 16 (Instrumented Satisfaction)**.** *An assertion* Δ *is satisfied in an instrumented thread state* $\overline{\sigma}_{\mathtt{i}} = \langle \sigma, \mathtt{P}, \_ \rangle$ *and channel configuration* $\mathtt{CC}$ *written* $\overline{\sigma}_{\mathtt{i}}, \mathtt{CC} \vDash \Delta$*, if* Δ *is satisfied in the thread's local state:*

$$\langle \sigma, \mathtt{P}, \_ \rangle, \mathtt{CC} \vDash \Delta \ \Leftrightarrow\ \sigma \vDash \Delta.$$

Lemma 3. *For any two instrumented states* $\overline{\sigma}^1_{\mathtt{i}}, \overline{\sigma}^2_{\mathtt{i}} \in \mathtt{IState}$*, where* $\mathtt{comp}(\overline{\sigma}^1_{\mathtt{i}}, \overline{\sigma}^2_{\mathtt{i}})$ *and* $\overline{\sigma}^1_{\mathtt{i}} \uplus \overline{\sigma}^2_{\mathtt{i}} \in \mathtt{IState}$*, there exists* $\mathtt{CC} \in \mathtt{CConfig}$*, and formulae* Δ_1, Δ_2 *such that:*

$$\overline{\sigma}^1_{\mathtt{i}}, \mathtt{CC} \vDash \Delta_1 \ \wedge\ \overline{\sigma}^2_{\mathtt{i}}, \mathtt{CC} \vDash \Delta_2 \ \Rightarrow\ \overline{\sigma}^1_{\mathtt{i}} \uplus \overline{\sigma}^2_{\mathtt{i}}, \mathtt{CC} \vDash \Delta_1 * \Delta_2.$$

Proof: This proof is straightforward from the definition of instrumented states composition and the semantic of $*$, provided that Δ_1, Δ_2 are precise. □

The validity of a triple is inductively defined with respect to the reduction rules and satisfaction relation as follows:

$$
\begin{array}{lll}
\overline{\sigma}_{\mathtt{i}}, \mathtt{CC} \vDash \mathtt{init}(\mathtt{c}) & \text{iff} & (\mathtt{CS}, \mathtt{G}) = \mathtt{CC} \wedge \mathtt{c} \in \mathtt{G} \wedge \neg(\exists \tilde{\mathtt{c}} \cdot \mathtt{c}, \_ = \mathtt{CS}(\tilde{\mathtt{c}})) \\
\overline{\sigma}_{\mathtt{i}}, \mathtt{CC} \vDash \mathtt{Peer}(\mathtt{P}) & \text{iff} & \overline{\sigma}_{\mathtt{i}} = \langle \_, \mathtt{P}, \_ \rangle \\
\overline{\sigma}_{\mathtt{i}}, \mathtt{CC} \vDash \mathtt{opened}(\mathtt{c}, \mathtt{P}^*, \mathtt{res}) & \text{iff} & ((\mathtt{CS}, \_) {=} \mathtt{CC}) \wedge (\mathtt{c}, \mathtt{P}^* = \mathtt{CS}(\tilde{\mathtt{c}})) \\
\overline{\sigma}_{\mathtt{i}}, \mathtt{CC} \vDash \mathtt{empty}(\tilde{\mathtt{c}}) & \text{iff} & \exists\, \mathtt{c}, \mathtt{P}^* \cdot (\overline{\sigma}_{\mathtt{i}}, \mathtt{CC} \vDash \mathtt{opened}(\mathtt{c}, \mathtt{P}^*, \tilde{\mathtt{c}})) \wedge \mathtt{EMPTY}(\mathtt{P}^*, (\mathtt{G})\lfloor_{\mathtt{c}}) \\
\overline{\sigma}_{\mathtt{i}}, \mathtt{CC} \vDash \mathcal{C}(\mathtt{c}, \mathtt{P}, \mathtt{L}) & \text{iff} & \exists\, \mathtt{c}, \mathtt{P}^* \cdot (\overline{\sigma}_{\mathtt{i}}, \mathtt{CC} \vDash \mathtt{opened}(\mathtt{c}, \mathtt{P}^*, \tilde{\mathtt{c}})) \wedge (\overline{\sigma}_{\mathtt{i}}, \mathtt{CC} \vDash \mathtt{Peer}(\mathtt{P})) \\
 & & \wedge((\_, \mathtt{G}) {=} \mathtt{CC}) \wedge (\mathtt{L} \equiv (\mathtt{G})\lfloor_{\mathtt{P},\mathtt{c}})
\end{array}
$$

Figure 6.9: The Semantics of Auxiliary Abstract Predicates

Definition 17 (Validity)**.** *A triple* $\{\Delta_1\}\ \mathtt{e}\ \{\Delta_2\}$ *is valid, written* $\vDash \{\Delta_1\}\ \mathtt{e}\ \{\Delta_2\}$ *if:*

$$
\begin{array}{l}
\forall \overline{\sigma}_{\mathtt{i}} \in \mathtt{IState}, \mathtt{CC} \in \mathtt{CConfig} \cdot \\
\quad (\overline{\sigma}_{\mathtt{i}}, \mathtt{CC} \vDash \Delta_1) \ \wedge\ (\overline{\sigma}_{\mathtt{i}} = \langle \_, \_, \mathtt{e} \rangle) \ \wedge\ (\langle \overline{\sigma}_{\mathtt{i}}, \mathtt{CC} \rangle \hookrightarrow \langle \overline{\sigma}'_{\mathtt{i}}, \mathtt{CC}' \rangle) \ \wedge\ (\overline{\sigma}'_{\mathtt{i}} = \langle \_, \_, \mathtt{e}' \rangle) \\
\qquad \Rightarrow\ \exists \Delta \cdot (\overline{\sigma}'_{\mathtt{i}}, \mathtt{CC}' \vDash \Delta) \ \wedge\ (\{\Delta\}\ \mathtt{e}'\ \{\Delta_2\}).
\end{array}
$$

Theorem 1 (Preservation)**.** *For expression* e *and states* Δ_1 *and* Δ_2*, if* $\vdash \{\Delta_1\}\,\mathtt{e}\,\{\Delta_2\}$ *then* $\vDash \{\Delta_1\}\ \mathtt{e}\ \{\Delta_2\}$.

Proof: We first show that each proof rule is sound: if the premisses are valid, then the conclusion is valid. It then follows, by structural induction on e, that every provable formula is valid. We only focus on the communication related rules, since the rest are standard. Assume $\langle \overline{\sigma}_{\mathtt{i}}, \mathtt{CC} \rangle$ as the initial configuration for each of the case studies below:

Parallel Decomposition

Suppose that $\overline{\sigma}_{\mathtt{i}}, \mathtt{CC} \vDash \Delta_1 * \Delta_2$. Since we assume only precise formulae, it results that there exists $\overline{\sigma}^1_{\mathtt{i}}, \overline{\sigma}^2_{\mathtt{i}} \in \mathtt{IState}$ such that $\overline{\sigma}_{\mathtt{i}} = \overline{\sigma}^1_{\mathtt{i}} \uplus \overline{\sigma}^2_{\mathtt{i}}$, and $\overline{\sigma}^1_{\mathtt{i}}, \mathtt{CC} \vDash \Delta_1$ and $\overline{\sigma}^2_{\mathtt{i}}, \mathtt{CC} \vDash \Delta_2$. From the dynamic semantics then either $\langle \overline{\sigma}^1_{\mathtt{i}}, \mathtt{CC} \rangle \hookrightarrow^* \mathtt{error}$, or $\langle \overline{\sigma}^2_{\mathtt{i}}, \mathtt{CC} \rangle \hookrightarrow^* \mathtt{error}$, in which case the whole program faults as per [iOP–JOIN–ERR1] and [iOP–JOIN–ERR2], respectively, or there exists $\overline{\sigma}^{1'}_{\mathtt{i}}, \overline{\sigma}^{2'}_{\mathtt{i}} \in \mathtt{IState}$ such that $\langle \overline{\sigma}^1_{\mathtt{i}}, \mathtt{CC} \rangle \hookrightarrow^* \langle \overline{\sigma}^{1'}_{\mathtt{i}}, \mathtt{CC}_1 \rangle$, and $\langle \overline{\sigma}^2_{\mathtt{i}}, \mathtt{CC} \rangle \hookrightarrow^* \langle \overline{\sigma}^{2'}_{\mathtt{i}}, \mathtt{CC}_2 \rangle$. with $\overline{\sigma}^{1'}_{\mathtt{i}} = \langle \sigma'_1, , \mathtt{e}'_1 \rangle$ and $\overline{\sigma}^{2'}_{\mathtt{i}} = \langle \sigma'_2, , \mathtt{e}'_2 \rangle$. If $\neg(\mathtt{CC}_1 {\equiv} \mathtt{CC}_2)$, then it must the case that either $\mathtt{e}'_1 \neq \mathsf{skip}$ or $\mathtt{e}'_2 \neq \mathsf{skip}$, corresponding thus to either [iOP–JOIN–SKIP1] or [iOP–JOIN–SKIP2], respectively. If there is no race, then both threads stabilize according to [iOP–JOIN], meaning that there exists $\overline{\sigma}^{1''}_{\mathtt{i}}, \overline{\sigma}^{2''}_{\mathtt{i}} \in \mathtt{IState}$ such that $\langle \overline{\sigma}^{1'}_{\mathtt{i}}, \mathtt{CC}'_1 \rangle \hookrightarrow^* \langle \overline{\sigma}^{1''}_{\mathtt{i}}, \mathtt{CC}''_1 \rangle$, and $\langle \overline{\sigma}^{2'}_{\mathtt{i}}, \mathtt{CC}'_2 \rangle \hookrightarrow^* \langle \overline{\sigma}^{2''}_{\mathtt{i}}, \mathtt{CC}''_2 \rangle$, with $\mathtt{CC}''_1 {\equiv} \mathtt{CC}''_2$, and $\overline{\sigma}^{1''}_{\mathtt{i}} = \langle \sigma''_1, , \mathsf{skip} \rangle$ and $\overline{\sigma}^{2''}_{\mathtt{i}} = \langle \sigma''_2, , \mathsf{skip} \rangle$. Therefore, there exists precise formulae Δ'_1, Δ'_2 such that $\overline{\sigma}^{1''}_{\mathtt{i}}, \mathtt{CC} \vDash \Delta'_1$ and $\overline{\sigma}^{2''}_{\mathtt{i}}, \mathtt{CC} \vDash \Delta'_2$. It results from Lemma 3 that

$\overline{\sigma}_{\mathtt{i}}^{1''} \uplus \overline{\sigma}_{\mathtt{i}}^{2''}, \mathtt{CC} \vDash \Delta_1' * \Delta_2'$, hence the conclusion.

Open If $\overline{\sigma}_{\mathtt{i}}, \mathtt{CC} \vDash \mathtt{init(c)}$ then opening a channel according to [**iOP–OPEN**] involves updating the channel store with a map from `res` to $\mathtt{c}, \mathtt{P}^*$, that is a state which, according to Fig. 6.9, satisfies the $\mathtt{opened(c, P^*, res)}$ predicate, therefore the conclusion holds.

Close If $\overline{\sigma}_{\mathtt{i}}, \mathtt{CC} \vDash \mathtt{empty}(\tilde{\mathtt{c}})$ then $\mathtt{EMPTY}(\mathtt{P}^*, (\mathtt{G})\downarrow_{\mathtt{c}})$ holds (assuming `HOLE`s are ignored during projection), and this thread cannot, therefore, dynamically reach fault according to the operational semantics of Fig. 6.4. The post-state trivially holds in this case.

Send If $\overline{\sigma}_{\mathtt{i}}, \mathtt{CC} \vDash \mathcal{C}(\mathtt{c}, \mathtt{P}, !\mathtt{v} \cdot \mathtt{V(v)};\mathtt{L}) * \mathtt{V(x)} * \mathtt{Peer(P)} * \mathtt{opened(c, P^*, \tilde{c})} \wedge \mathtt{P} \in \mathtt{P}^*$ (1) holds, it then follows immediately that the first three premises of [**iOP–SEND**] also hold: $(\mathtt{CS}, \mathtt{G}) = \mathtt{CC} \wedge (\mathtt{c}, \mathtt{P}^* = \mathtt{CS}(\tilde{\mathtt{c}})) \wedge (\mathtt{P} \in \mathtt{P}^*)$. We next need to prove that $\mathtt{S_{safe}}$ also holds. Since $\mathcal{C}(\mathtt{c}, \mathtt{P}, !\mathtt{v} \cdot \mathtt{V(v)};\mathtt{L})$ with the race-free specification $!\mathtt{v} \cdot \mathtt{V(v)};\mathtt{L}$ for channel `c` and party `P` is satisfied in the configuration $\overline{\sigma}_{\mathtt{i}}, \mathtt{CC}$, it implies that indeed the next sending operation expected by $(\mathtt{G})\downarrow_{\mathtt{c}}$ get consumed on `P`, hence $\mathtt{SAFE} = \mathtt{S_{safe}}(\mathtt{P}, (\mathtt{G})\downarrow_{\mathtt{c}})$ also holds. This ensures that the current execution cannot fault with `RACE_ERR`, as depicted by [**iOP–SEND–ERACE**]. We next check whether `FID` holds. The fact that $\mathtt{S_{safe}}$ holds, guarantees that the next operation described by `G` w.r.t. `c` and `P` is a send, we need to check that `P` is not expected to perform another transmission on a channel different than `c`. That follows directly from the pattern of the communication specification: $!\mathtt{v} \cdot \mathtt{V(v)};\mathtt{L}$. From Prop. 1 and the projection rules in Fig. 4.6c, it results that should there be any other transmission w.r.t. party `P` but on a different channel, then the specification should have contained a guard of the form $\ominus(\mathtt{P}^{(i)})$. Since the channel specification holds for $!\mathtt{v} \cdot \mathtt{V(v)}$ as the next transmission (2), it implies that there are no other intermediary transmissions on `P`, hence $(\mathtt{SAFE}, \_, \_) = \mathtt{FID}(\mathtt{G}, \mathtt{c}, \mathtt{P})$, which is a shorthand for $\exists \Delta \cdot (\mathtt{SAFE}, \mathtt{v} \cdot \Delta, \_) = \mathtt{FID}(\mathtt{G}, \mathtt{c}, \mathtt{P})$ (3), hence this computation cannot fault with `PROT_ERR` ([**iOP–SEND–EPROT**]). From (2) and (3) we can conclude that $\mathtt{V(v)} = \Delta$. Since the current state satisfies $\mathtt{V(x)}$, it follows immediately that $\exists \sigma_{\mathtt{x}}, \sigma' \cdot (\sigma = \sigma' \uplus \sigma_{\mathtt{x}}) \wedge (\sigma_{\mathtt{x}} \vDash [\mathtt{v}/\mathtt{x}]\Delta)$ (4), or in other words the current state owns the message it sends on $\tilde{\mathtt{c}}$. This proves that the current execution cannot fault with `RES_ERR` ([**iOP–SEND–ERES**]). We can conclude therefore that only [**iOP–SEND–EPROT**] can be fired in this case, meaning that:

$\langle \sigma', \mathtt{P}, \_ \rangle, \mathtt{CC}' \vDash \mathcal{C}(\mathtt{c}, \mathtt{P}, \mathtt{L}) * \mathtt{Peer(P)} * \mathtt{opened(c, P^*, \tilde{c})} \wedge \mathtt{P} \in \mathtt{P}^*$, which trivially holds by

(1) and (4) since the only updates to the configuration are represented by the thread state σ', reflecting now the fact that the current thread lost the ownership of the transmitted message, and $\mathtt{CC}'$ whose global protocol $\mathtt{G}$ is updated with a $\mathtt{HOLE}$ to denote the consumed transmission: $(\mathtt{P},\mathtt{c})\lfloor_{\mathtt{G}} \equiv \mathtt{L}$.

Receive If $\overline{\sigma}_{\mathtt{i}}, \mathtt{CC} \vDash \mathcal{C}(\mathtt{c},\mathtt{P},?\mathtt{v}\cdot\mathtt{V}(\mathtt{v});\mathtt{L}) * \mathtt{Peer}(\mathtt{P}) * \mathtt{opened}(\mathtt{c},\mathtt{P}^*,\tilde{\mathtt{c}}) \wedge \mathtt{P}{\in}\mathtt{P}^*$ (1) holds, similar to [**iOP–SEND**], it follows immediately by Fig. 6.9 that the first three premises of [**iOP–RECV**] also hold: $(\mathtt{CS},\mathtt{G}){=}\mathtt{CC} \wedge (\mathtt{c},\mathtt{P}^* = \mathtt{CS}(\tilde{\mathtt{c}})) \wedge (\mathtt{P} \in \mathtt{P}^*)$. If the corresponding send was not fired yet, the thread will stay in the same state, and when it will be eventually fired, (1) is still satisfied. Since $\mathcal{C}(\mathtt{c},\mathtt{P},?\mathtt{v}\cdot\mathtt{V}(\mathtt{v});\mathtt{L})$ (2) specifies a race-free communication w.r.t. channel $\mathtt{c}$ and party $\mathtt{P}$, it follows immediately that $(\mathtt{G})\lfloor_{\mathtt{c}}$ expects for the next receiving operation over $\mathtt{c}$ is on $\mathtt{P}$, hence $\mathtt{SAFE} = \mathtt{R}_{\mathtt{safe}}(\mathtt{P},(\mathtt{G})\lfloor_{\mathtt{c}})$ (3), assuming the state is not blocked. This ensures that the current execution cannot fault with $\mathtt{RACE\_ERR}$, as depicted by [**iOP–RECV–ERACE**]. We next check whether $\mathtt{FID}$ holds. (3) implies that the next transmission w.r.t. $\mathtt{c}$ and $\mathtt{P}$ is indeed a receive, we need to check that there are no intermediate transmissions on different channels. From Prop. 1, the projection rules Fig. 4.6c and (1), that is the communication specification $\mathcal{C}(\mathtt{c},\mathtt{P},?\mathtt{v}\cdot\mathtt{V}(\mathtt{v});\mathtt{L})$ expects a receive, it results that there are no other prior unconsumed transmissions expected over $\mathtt{c}$, since that would have involved for the specification to be guarded by $\ominus(\mathtt{P}^{(i)})$ prior to the received specified by (2). It then follows that $\exists\Delta\cdot(\mathtt{SAFE},\mathtt{v}{\cdot}\Delta,\_) = \mathtt{FID}(\mathtt{G},\mathtt{c},\mathtt{P})$ (4), hence this thread cannot fault with $\mathtt{PROT\_ERR}$ ([**iOP–RECV–EPROT**]). From (2) and (4) we can conclude that $\mathtt{V}(\mathtt{v}) = \Delta$. Executing the transmission according to [**iOP–RECV–EPROT**] results in the update of the current thread state such that it reflects the message ownership gain: $\exists\sigma_{\mathtt{res}}\cdot\sigma'{:=}\sigma{\uplus}\sigma_{\mathtt{res}}$ where $\sigma_{\mathtt{res}} \vDash [\mathtt{res}/\mathtt{v}]\Delta$. Since $\mathtt{V}(\mathtt{v}) = \Delta$, then the following also holds: $\sigma_{\mathtt{res}} \vDash \mathtt{V}(\mathtt{res})$ (5). With (1), (5) and the update of $\mathtt{G}$ to reflect the consumed receive operation, where $(\mathtt{P},\mathtt{c})\lfloor_{\mathtt{G}} \equiv \mathtt{L}$, we can conclude that the updated thread state satisfies the consequence of **RECV**: $\langle\sigma',\mathtt{P},\_\rangle,\mathtt{CC}' \vDash \mathcal{C}(\mathtt{c},\mathtt{P},\mathtt{L}){*}\mathtt{V}(\mathtt{res}) * \mathtt{Peer}(\mathtt{P}) * \mathtt{opened}(\mathtt{c},\mathtt{P}^*,\tilde{\mathtt{c}}) \wedge \mathtt{P}{\in}\mathtt{P}^*$.

□

Theorem 2 (Progress)**.** *If* $\vdash \{\Delta_1\}\,\mathtt{e}\,\{\Delta_2\}$ *and* $\exists\overline{\sigma}_{\mathtt{i}} \in \mathtt{IState}, \mathtt{CC} \in \mathtt{CConfig}\cdot\overline{\sigma}_{\mathtt{i}},\mathtt{CC} \vDash \Delta_1$ *then either* $\mathtt{e}$ *is a value or* $\exists\overline{\sigma}'_{\mathtt{i}} \in \mathtt{IState}, \mathtt{CC}' \in \mathtt{CConfig}\cdot\langle\overline{\sigma}_{\mathtt{i}},\mathtt{CC}\rangle \hookrightarrow \langle\overline{\sigma}'_{\mathtt{i}},\mathtt{CC}'\rangle$.

Proof: By induction on the length of the execution and by case analysis on the steps taken we could show that if $\langle \overline{\sigma}_{\mathtt{i}}, \mathtt{CC} \rangle$ is a non-final, fault-free configuration, then $\langle \overline{\sigma}_{\mathtt{i}}, \mathtt{CC} \rangle$ doesn't get stuck. The communication related cases are straightforward assuming the well-formedness of communication protocols. It may appear as if RECV could cause a process to get stuck, however, if the protocol which describes the communication is well-formed, as per Def. 6, it is guaranteed for a corresponding sender to get fired within a finite number of machine steps. □

Chapter 7

MODULAR PROTOCOLS

Modularity is essential in designing and implementing new software since it often involves reusing and composing existing components. Software components are modular if their composition in larger software preserves the overall expected computation and, equally important, the safety properties. Checking software compatibility is a known hard problem, even more so when asynchronous communication across components is involved, a heavily used mechanism in building distributed system. In the subsequent subsections, we list a series of extensions to the current logic which support the design of modular protocols. Modular protocols are designed and refined once, and then re-used multiple times to support the design of more complex protocols.

7.1 Labelling

Thus far our reasoning was based on the fact that each transmission label is unique. To ensure that there is no label clash across all transmission, even in the presence of composed protocols and multiple instantiations of the same protocol, a label is defined as a composition between a parameterized label root and a unique local id: $i\#i_{\%\mathtt{d}}$, where such a composed label is a potential label root for nested protocol instantiations.

Example 1. Let $\mathtt{H}$ be a global protocol which is assembled using protocol $\mathtt{H_0}$:

$\mathtt{H_0}(\mathtt{A},\mathtt{B},\mathtt{c},i) \triangleq \mathtt{A} \xrightarrow{i\#1} \mathtt{B} : \mathtt{c}.$ $\qquad$ $\mathtt{H}(\mathtt{A},\mathtt{B},\mathtt{C},\mathtt{c},i) \triangleq \mathtt{A} \xrightarrow{i\#1} \mathtt{B} : \mathtt{c}; \mathtt{H_0}(\mathtt{B},\mathtt{C},\mathtt{c},i\#2).$

Unrolling $\mathtt{H_0}$ within the definition of $\mathtt{H}$ results in the following protocol with unique labels:

$\mathtt{H}(\mathtt{A},\mathtt{B},\mathtt{C},\mathtt{c},i) \triangleq \mathtt{A}\xrightarrow{i\#1}\mathtt{B}:\mathtt{c};\mathtt{B}\xrightarrow{i\#2\#1}\mathtt{C}:\mathtt{c}.$

7.2 Parameterized Frontier for Previous State

As highlighted in Sec. 4.2.3 the ordering assumptions and race-free proof obligations are collected via a manipulation of protocol summaries, where a summary consists of a backtier, a frontier and accumulated ordering constraints. The orderings are then added to the global protocol as part of the refinement phase. To extend this approach to composable protocols we equip each protocol with an extra parameter meant to link the current protocol with a generic previous state ensuring therefore that wherever plugged-in, the considered protocol does not create a communication race. Generally speaking, a protocol is described by a predicate in our session logic whose parameters represent the communicating peers, the logical channels used for communication, a root label and the previous state frontier: $\mathtt{H}(\mathtt{P}^*,\mathtt{c}^*,i,\mathtt{F})$. The same $\mathtt{F}$ variable is also used later in Sec 7.4 to denote the frontier for the next stage, via $\mathtt{F}'$,

Example 1 - revisited. Using the same simple example as in the previous paragraph we highlight the key ideas of parameterized frontier, $\mathtt{F}$, as below:

$\mathtt{H}_0(\mathtt{A},\mathtt{B},\mathtt{c},i,\mathtt{F}) \triangleq \mathtt{A}\xrightarrow{i\#1}\mathtt{B}:\mathtt{c}. \qquad \mathtt{H}(\mathtt{A},\mathtt{B},\mathtt{C},\mathtt{c},i,\mathtt{F}) \triangleq \mathtt{A}\xrightarrow{i\#1}\mathtt{B}:\mathtt{c};\mathtt{H}_0(\mathtt{B},\mathtt{C},\mathtt{c},i\#2,\mathtt{F}_0).$

where $\mathtt{F}_0{=}\mathtt{F}\lfloor;\rfloor\mathcal{S}_1^{\mathtt{ft}}$ and $\mathcal{S}_1{=}\mathtt{CC}(\mathtt{A}\xrightarrow{i\#1}\mathtt{B}:\mathtt{c})$, therefore $\mathtt{F}_0{=}\mathtt{F}\lfloor;\rfloor\langle[\mathtt{A}{:}\mathtt{A}^{(i\#1)},\mathtt{B}{:}\mathtt{B}^{(i\#1)}],[\mathtt{c}{:}\mathtt{A}\xrightarrow{i\#1}\mathtt{B}]\rangle$.

After the refinement phase and the proper instantiation of $\mathtt{H}_0$ in the body of $\mathtt{H}$, we get:

$\mathtt{H}_0(\mathtt{A},\mathtt{B},\mathtt{c},i,\mathtt{F}) \triangleq$

$$\mathtt{A}\xrightarrow{i\#1}\mathtt{B}:\mathtt{c};\oplus(\mathtt{A}\xrightarrow{i\#1}\mathtt{B});\oplus(\mathtt{F}.\mathtt{K}(\mathtt{A})\prec_{\mathtt{HB}}\mathtt{A}^{(i\#1)});\oplus(\mathtt{F}.\mathtt{K}(\mathtt{B})\prec_{\mathtt{HB}}\mathtt{B}^{(i\#1)});\ominus(\mathtt{F}.\Gamma(\mathtt{c})\prec_{\mathtt{HB}}i\#1).$$

$\mathtt{H}(\mathtt{A},\mathtt{B},\mathtt{C},\mathtt{c},i,\mathtt{F}) \triangleq$

$$\mathtt{A}\xrightarrow{i\#1}\mathtt{B}:\mathtt{c};\oplus(\mathtt{A}\xrightarrow{i\#1}\mathtt{B});\oplus(\mathtt{F}.\mathtt{K}(\mathtt{A})\prec_{\mathtt{HB}}\mathtt{A}^{(i\#1)});\oplus(\mathtt{F}.\mathtt{K}(\mathtt{B})\prec_{\mathtt{HB}}\mathtt{B}^{(i\#1)});\quad\ominus(\mathtt{F}.\Gamma(\mathtt{c})\prec_{\mathtt{HB}}i\#1);$$

$$\rho(\mathtt{A}\xrightarrow{i\#1}\mathtt{B}:\mathtt{c};\oplus(\mathtt{A}\xrightarrow{i\#1}\mathtt{B});\oplus(\mathtt{F}.\mathtt{K}(\mathtt{A})\prec_{\mathtt{HB}}\mathtt{A}^{(i\#1)});\oplus(\mathtt{F}.\mathtt{K}(\mathtt{B})\prec_{\mathtt{HB}}\mathtt{B}^{(i\#1)});\ominus(\mathtt{F}.\Gamma(\mathtt{c})\prec_{\mathtt{HB}}i\#1)).$$

where the parameter substitution is described by $\rho{=}[\mathtt{B}/\mathtt{A},\mathtt{C}/\mathtt{B},i\#2/i,\mathtt{F}_0/\mathtt{F}]$. Applying the substitution and identifying the elements of $\mathtt{F}_0$, $\mathtt{H}$ is then normalized to:

$\mathtt{H}(\mathtt{A},\mathtt{B},\mathtt{C},\mathtt{c},i,\mathtt{F}) \triangleq$

$$\mathtt{A}\xrightarrow{i\#1}\mathtt{B}:\mathtt{c};\oplus(\mathtt{A}\xrightarrow{i\#1}\mathtt{B});\oplus(\mathtt{F}.\mathtt{K}(\mathtt{A})\prec_{\mathtt{HB}}\mathtt{A}^{(i\#1)});\oplus(\mathtt{F}.\mathtt{K}(\mathtt{B})\prec_{\mathtt{HB}}\mathtt{B}^{(i\#1)});\ominus(\mathtt{F}.\Gamma(\mathtt{c})\prec_{\mathtt{HB}}i\#1);$$

$$\mathtt{B}\xrightarrow{i\#2\#1}\mathtt{C}:\mathtt{c};\oplus(\mathtt{B}\xrightarrow{i\#2\#1}\mathtt{C});\oplus(\mathtt{B}^{(i\#1)}\prec_{\mathtt{HB}}\mathtt{B}^{(i\#2\#1)});\oplus(\mathtt{F}.\mathtt{K}(\mathtt{C})\prec_{\mathtt{HB}}\mathtt{C}^{(i\#2\#1)});\ominus(i\#1\prec_{\mathtt{HB}}i\#2\#1).$$

Note that, for the clarity of this example (and subsequent ones), we assume that $\mathtt{F}.\Gamma(\mathtt{c})$ returns exactly one transmission and $\mathtt{F}.\mathtt{K}(\mathtt{P})$ returns exactly one event, which is actually the case for this example. However, according to the definition of a map element, Fig. 4.4, the

maps are populated with a composition of transmissions or events. To handle element maps, we should simply form a $\prec_{\mathtt{HB}}$ relation for each event/transmission returned by the map and add an assumption or a guard for each such created relation.

7.3 Sufficient Condition for Implicit Synchronization

Adding an instantiable frontier to link the current protocol with its usage context enables the refinement process to add the necessary ordering assumptions and guards within the current protocol. As explained in previous sections, these guards hold either when sufficient implicit synchronization is provided, or when explicit synchronization mechanisms are used. To give system designer the option to compose only protocols that are implicitly sychronized with the environment where they are plugged in, we could derive a guard which enforces the implicit synchronization between the protocol and its pre-usage context. To this purpose, we consider a diagramatic view of the ordering relations where each edge is either an HB or CB relation, and aim to find those missing edges which would make the race-free guards (involving the pre-state) hold.

Definition 18 (Diagramatic Ordering Relations)**.** *A diagramatic view for a set* $\mathtt{A}^{\oplus}$ *of event orderings is a directed acyclic graph* $\mathcal{G}(\mathtt{A}^{\oplus})=(\mathtt{V},\mathtt{Edg})$*, where the set* V *of vertices contains all the events in* $\mathtt{A}^{\oplus}$*,* $\mathtt{V}=\bigcup_{\Psi\in\mathtt{A}^{\oplus}}\mathtt{EV}(\Psi)$*, and the set* Edg *of edges represents all the* HB *and* CB *ordering relations in* $\mathtt{A}^{\oplus}$*,* $\mathtt{Edg}=\{(\mathtt{E}_1,\mathtt{E}_2)\mid \mathtt{E}_1\prec_{\mathtt{HB}}\mathtt{E}_2\in\mathtt{A}^{\oplus} \text{ or } \mathtt{E}_1\prec_{\mathtt{CB}}\mathtt{E}_2\in\mathtt{A}^{\oplus}\}$.

The derivation of the precondition for the implicit synchronization is depicted in **Algorithm 2**: the generic frontier is merged with the backtier of the protocol's body (line 2) to generate the RF guards which ensure safe composition with the environment. For each such guard $\mathtt{E}_1\prec_{\mathtt{HB}}\mathtt{E}_2\in\mathtt{A}^{\ominus}$ to be satisfied, the algorithm searches backwards, starting from $\mathtt{E}_2$, a way to connect it to $\mathtt{E}_1$ using only ancestor nodes associated to the generic frontier. The $\mathtt{ancestors}(\mathtt{E},\mathcal{G})$ method returns all the ancestors nodes of node E in graph $\mathcal{G}$. The Cartesian product $\times_{\mathtt{HB}}$ is used to create *weak* HB relations between $\mathtt{E}_1$ and the ancestors of $\mathtt{E}_2$. To omit redundancies, the algorithm only considers $\mathtt{E}_2$'s earliest ancestors. The Cartesian product between sets of events is defined as follows:

$$\mathtt{S}_1\times_{\mathtt{HB}}\mathtt{S}_2 \stackrel{\text{def}}{=} \{\mathtt{E}_1\preceq_{\mathtt{HB}}\mathtt{E}_2\mid \mathtt{E}_1\in\mathtt{S}_1 \text{ and } \mathtt{E}_2\in\mathtt{S}_2\} \text{ where } \mathtt{E}_1\preceq_{\mathtt{HB}}\mathtt{E}_2 \stackrel{\text{def}}{=} \mathtt{E}_1\prec_{\mathtt{HB}}\mathtt{E}_2\vee\mathtt{E}_1=\mathtt{E}_2$$

The weaker relation $\preceq_{\mathtt{HB}}$ is needed to express that two events might be derived as identical. It is easy to notice that the [HB-HB] propagation rule defined in Fig. 4.3b can soundly be extended

to account for the $\preceq_{\mathtt{HB}}$ relations as well:

$$\mathtt{E}_1 \prec_{\mathtt{HB}} \mathtt{E}_2 \wedge \mathtt{E}_2 \preceq_{\mathtt{HB}} \mathtt{E}_3 \quad \Rightarrow \mathtt{E}_1 \prec_{\mathtt{HB}} \mathtt{E}_3 \quad \text{[HB-HB(a)]}$$

$$\mathtt{E}_1 \preceq_{\mathtt{HB}} \mathtt{E}_2 \wedge \mathtt{E}_2 \prec_{\mathtt{HB}} \mathtt{E}_3 \quad \Rightarrow \mathtt{E}_1 \prec_{\mathtt{HB}} \mathtt{E}_3 \quad \text{[HB-HB(b)]}$$

$$\mathtt{E}_1 \preceq_{\mathtt{HB}} \mathtt{E}_2 \wedge \mathtt{E}_2 \preceq_{\mathtt{HB}} \mathtt{E}_3 \quad \Rightarrow \mathtt{E}_1 \preceq_{\mathtt{HB}} \mathtt{E}_3 \quad \text{[HB-HB(c)]}$$

however, the [CB-HB] cannot be fired in the presence of $\preceq_{\mathtt{HB}}$ since that would involve changing a $\prec_{\mathtt{CB}}$ relation into $\prec_{\mathtt{HB}}$ which would lead to unsoundness. For brevity, we use the shorthand: $\mathtt{EV}(\mathtt{F.K})$ to denote $\bigcup_{\mathtt{P}\in\mathtt{dom}(\mathtt{F.K})} \mathtt{EV}(\mathtt{F.K(P)})$ and $\mathtt{EV}(\mathtt{F}.\Gamma)$ to denote $\bigcup_{\mathtt{c}\in\mathtt{dom}(\mathtt{F.K})} \mathtt{EV}(\mathtt{F}.\Gamma(\mathtt{c}))$.

Algorithm 2: Derives the necessary conditions for this protocol to be implicitly synchronized with the pre-context

input : $\mathtt{H}(\mathtt{P}^*, \mathtt{c}^*, i, \mathtt{F}) \stackrel{\text{def}}{=} \mathtt{G}$ - a global multi-party protocol
output: $\mathtt{A}^{\ominus}_{\mathtt{pre}}$ - a set of orderings guards

1 $\mathcal{S} \leftarrow \mathtt{CC}(\mathtt{G})$; $\mathtt{A}^{\ominus}_{\mathtt{pre}} \leftarrow \emptyset$;
2 $\langle \mathtt{A}^{\oplus}, \mathtt{A}^{\ominus} \rangle \leftarrow \mathtt{F}\lfloor ; \rfloor \mathcal{S}^{\mathtt{bt}}$; /* relations between environment and H */
3 **foreach** $\mathtt{E}_1 \prec_{\mathtt{HB}} \mathtt{E}_2 \in \mathtt{A}^{\ominus}$ **do**
4 $\quad \mathtt{A}^{\mathtt{ev}} \leftarrow \mathtt{ancestors}(\mathtt{E}_2, \mathcal{G}(\mathtt{A}^{\oplus})) \cap (\mathtt{EV}(\mathtt{F.K}) \cup \mathtt{EV}(\mathtt{F}.\Gamma))$; /* only frontier ancestor */
5 $\quad \overline{\mathtt{A}^{\mathtt{ev}}} \leftarrow \{\mathtt{E} \mid \mathtt{ancestors}(\mathtt{E}, \mathcal{G}(\mathtt{A}^{\oplus})) \cap \mathtt{A}^{\mathtt{ev}} = \emptyset\}$; /* only earliest ancestors */
6 $\quad \mathtt{A}^{\mathtt{hb}} \leftarrow \{\mathtt{E}_1\} \times_{\mathtt{HB}} \overline{\mathtt{A}^{\mathtt{ev}}}$; /* candidate HB relations */
7 $\quad \overline{\mathtt{A}^{\mathtt{hb}}} \leftarrow \{\vartheta_{\mathtt{hb}} \mid \vartheta_{\mathtt{hb}} \in \mathtt{A}^{\mathtt{hb}} \text{ s.t. } \mathtt{A}^{\oplus} \cup \{\vartheta_{\mathtt{hb}}\} \vdash_{\mathtt{RF}} \mathtt{E}_1 \prec_{\mathtt{HB}} \mathtt{E}_2\}$; /* keep only useful HB */
8 $\quad \mathtt{A}^{\ominus}_{\mathtt{pre}} \leftarrow \mathtt{A}^{\ominus}_{\mathtt{pre}} \cup \{ \bigvee_{\vartheta_{\mathtt{hb}} \in \overline{\mathtt{A}^{\mathtt{hb}}}} \vartheta_{\mathtt{hb}} \}$; /* updates the pre guards */
9 **end**
10 **return** $\mathtt{A}^{\ominus}_{\mathtt{pre}}$

Example 1 - revisited. Using the same simple example as in the previous paragraph, we derive the conditions for which $\mathtt{H}_0$ could be implicitly synchronized with the body of H:

$$\langle \mathtt{A}^{\oplus}, \mathtt{A}^{\ominus} \rangle := \langle \{\mathtt{A}^{(i\#1)}, \mathtt{B}^{(i\#1)}, \mathtt{A}^{(i\#1)} \prec_{\mathtt{CB}} \mathtt{B}^{(i\#1)}, \mathtt{F.K(A)} \prec_{\mathtt{HB}} \mathtt{A}^{(i\#1)}, \mathtt{F.K(B)} \prec_{\mathtt{HB}} \mathtt{B}^{(i\#1)}\},$$
$$\{\mathtt{send}(\mathtt{F}.\Gamma(\mathtt{c})) \prec_{\mathtt{HB}} \mathtt{A}^{(i\#1)}, \mathtt{recv}(\mathtt{F}.\Gamma(\mathtt{c})) \prec_{\mathtt{HB}} \mathtt{B}^{(i\#1)}\}\rangle.$$

Building the graph corresponding to $\mathtt{A}^{\oplus}$, finding the ancestors of $\mathtt{A}^{(i\#1)}$ and $\mathtt{B}^{(i\#1)}$, and building the HB candidates yields $\mathtt{A}^{\ominus}_{\mathtt{pre}} = \{\mathtt{send}(\mathtt{F}.\Gamma(\mathtt{c})) \preceq_{\mathtt{HB}} \mathtt{F.K(A)}, \mathtt{recv}(\mathtt{F}.\Gamma(\mathtt{c})) \preceq_{\mathtt{HB}} \mathtt{F.K(B)}\}$ in two iterations of the loop. Given this precondition for $\mathtt{H}_0$ it is easy to observe with the appropriate instantiation that $\mathtt{H}_0$ cannot be simply plugged into H without proper explicit synchronization:

$$\{\mathtt{A}^{(i\#1)}, \mathtt{B}^{(i\#1)}, \mathtt{A}^{(i\#1)} \prec_{\mathtt{CB}} \mathtt{B}^{(i\#1)}, \mathtt{F.K(A)} \prec_{\mathtt{HB}} \mathtt{A}^{(i\#1)}, \mathtt{F.K(B)} \prec_{\mathtt{HB}} \mathtt{B}^{(i\#1)}\} \vdash_{\mathtt{RF}}$$
$$\rho(\mathtt{send}(\mathtt{F}.\Gamma(\mathtt{c})) \preceq_{\mathtt{HB}} \mathtt{F.K(A)} \wedge \mathtt{recv}(\mathtt{F}.\Gamma(\mathtt{c})) \preceq_{\mathtt{HB}} \mathtt{F.K(B)}).$$

with $\rho = [\mathtt{B}/\mathtt{A}, \mathtt{C}/\mathtt{B}, i\#2/i, \mathtt{F}_0/\mathtt{F}]$, the above involves proving that:

$\mathtt{A}^{(i\#1)} \preceq_{\mathtt{HB}} \mathtt{B}^{(i\#1)} \wedge \mathtt{B}^{(i\#1)} \preceq_{\mathtt{HB}} \mathtt{F.K(C)}$.

This precondition cannot be proved since there is no sufficient implicit synchronization.

Using $\mathtt{H_0}$ in a different context: $\mathtt{H_1(A,B,c},i,\mathtt{F}) \triangleq \mathtt{A} \xrightarrow{i\#1} \mathtt{B} : \mathtt{c}; \mathtt{H_0(A,B,c},i\#2,\mathtt{F_0})$. yields the following proof obligation: $\mathtt{A}^{(i\#1)} \preceq_{\mathtt{HB}} \mathtt{A}^{(i\#1)} \wedge \mathtt{B}^{(i\#1)} \preceq_{\mathtt{HB}} \mathtt{B}^{(i\#1)}$ which is trivially true. Therefore, $\mathtt{H_0}$ can be plugged in the body of $\mathtt{H_1}$ without any additional synchronization.

7.4 Predicate Summary for Post-Context

The condition for implicit synchronization ensures race-freedom w.r.t. the pre-usage context. To also ensure communication safety w.r.t. its post-usage context the predicate's frontier is made available to be plugged-in for merging with the usage site backtier.

Example 2. Using a variation of the previous examples, we highlight the role of a predicate's frontier at its usage context:

$\mathtt{H_2(A,B,C,c},i,\mathtt{F}) \triangleq \mathtt{A} \xrightarrow{i\#1} \mathtt{B} : \mathtt{c}\ ;\ \mathtt{B} \xrightarrow{i\#2} \mathtt{C} : \mathtt{c}.\mathtt{H_3(A,B,C,c},i,\mathtt{F}) \triangleq \mathtt{H_2(A,B,C,c},i\#1,\mathtt{F})\ ;\ \mathtt{B} \xrightarrow{i\#2} \mathtt{C} : \mathtt{c}.$

The frontier of $\mathtt{H_2}$ is derived to be $\langle[\mathtt{A}{:}\mathtt{A}^{(i\#1)}, \mathtt{B}{:}\mathtt{B}^{(i\#2)}, \mathtt{C}{:}\mathtt{C}^{(i\#2)}], [\mathtt{c}{:}i\#2]\rangle$. Using this information within the body of $\mathtt{H_3}$, leads to the following refined protocol:

$\mathtt{H_3(A,B,C,c},i,\mathtt{F}) \triangleq$

$\quad \mathtt{H_2(A,B,C,c},i\#1,\mathtt{F})\ ;$

$\quad \mathtt{B} \xrightarrow{i\#2} \mathtt{C} : \mathtt{c}; \oplus(\mathtt{B} \xrightarrow{i\#2} \mathtt{C}); \oplus(\mathtt{B}^{(i\#1\#2)} \prec_{\mathtt{HB}} \mathtt{B}^{(i\#2)}); \oplus(\mathtt{C}^{(i\#1\#2)} \prec_{\mathtt{HB}} \mathtt{C}^{(i\#2)}); \ominus(i\#1\#2 \prec_{\mathtt{HB}} i\#1).$

Notice that without the frontier of $\mathtt{H_2}$, its usage in the body of $\mathtt{H_3}$ would potentially lead to a race on $\mathtt{c}$ since $\mathtt{H_3}$ would otherwise not have enough information to add the ordering relations and race-free guards within its body (events labelled $i\#1\#2$).

7.5 Recursion

For recursive predicates we only consider those predicates which are self-contained, that is to say explicit synchronization is only supported within the body of the predicate but not across recursive calls. If this condition is not met, the recursion would have to pass synchronized objects across recursion, creating a much more complex communication model. We leave this more complex scenario for future work.

To support recursion we: (1) derive the sufficient condition for implicit synchronization with the pre-state according to the algorithm described in Sec. 7.3; (2) we derive the summary of the protocol up to the recursive invocation of the considered predicate, as described in Sec. 7.4;

(3) given the summary in (2) as ordering context, we next check whether the implicit synchronization condition holds for the recursive invocation. If it does, then the protocol is implicitly synchronized and could be safely plugged-in within any environment which satisfies its implicit synchronization condition.

Example 3. We now consider the following recursively defined communication protocol:

$\mathtt{H_5}(\mathtt{A},\mathtt{B},\mathtt{C},\mathtt{c},i,\mathtt{F}) \triangleq \mathtt{A} \xrightarrow{i\#1} \mathtt{B} : \mathtt{c}; \mathtt{C} \xrightarrow{i\#2} \mathtt{A} : \mathtt{c}; \mathtt{A} \xrightarrow{i\#3} \mathtt{B} : \mathtt{c}; \mathtt{H_5}(\mathtt{A},\mathtt{B},\mathtt{C},\mathtt{c},i,\mathtt{F})$.

To check whether this protocol is self-contained, the algorithm first follows the steps described in Sec. 7.3 to derive the sufficient condition for implicit synchronization. The condition is derived to be: $\mathtt{send}(\mathtt{F}.\Gamma(\mathtt{c})) \preceq_{\mathtt{HB}} \mathtt{F}.\mathtt{K}(\mathtt{A}) \wedge \mathtt{recv}(\mathtt{F}.\Gamma(\mathtt{c})) \preceq_{\mathtt{HB}} \mathtt{F}.\mathtt{K}(\mathtt{B})$, which, when instantiated to the body of $\mathtt{H_5}$ it boils down to check that: $\mathtt{A}^{(i\#3)} \preceq_{\mathtt{HB}} \mathtt{A}^{(i\#3)} \wedge \mathtt{B}^{(i\#3)} \preceq_{\mathtt{HB}} \mathtt{B}^{(i\#3)}$, which is trivially true. So despite containing races within its body, this protocol is actually safely synchronized across the recursive invocations.

Chapter 8

RELATED WORKS SURVEY

Two of the most common approaches meant to tackle the formalism challenges of concurrent systems are either based on *behavioral types* or on the *logics with support for synchronization primitives*. This chapter surveys some of the seminal works covering both of the aforementioned categories, and emphasizes on some notable extensions proposed for reasoning about message passing programs. Since our proposal best resembles the logic approaches rather than the types approaches, we shall individually draw an analogy with respect to each referenced logic, and treat the behavioural types as an ensemble when framing the comparison with the current proposal.

Another line of active research in the area of software verification uses model checking to reason about concurrent programs. However, the formalization of communication protocols seems to be less popular in this line of works, with the majority of model checkers being built for specific protocols. We finalize the related work chapter by describing some of these model checkers.

8.1 Behavioral Types

As a result of intensive research, behavioral types lay the foundation for the formalization of communication-centered applications. The key idea behind these type systems is to describe the communicating entities (whether communicators, or communicating mediums) in terms of sequences of events. In other words, the behavioural types specifies the expected interaction

pattern of communicating entities. The majority of these type systems are developed to capture properties of computations in process calculi. Depending strictly on the operating context, the check of the underlying code (or code abstraction in the considered calculi) against the type system could discover potential communication flaws, such as deadlock scenarios, resource race, or unexpected communication.

Among the first behavioural works for concurrency are those of Nielson and Nielson [73, 74], where the authors design a type and effect system in the context of a concurrent functional language. The aim is to statically describe both the intensional aspects of a computation as well as the usual type information. This is achieved by extending the type judgement (what is computed) to also capture the effect (or behaviour) of a command, and offering transition semantics also for the latter. Such a system can then help to regulate the resource usage and the communication topology.

Approaches to Behavioural Types. Some early seminal works which have influenced the current state of the art of types systems for π-calculus include those of Kobayashi [53, 54] for deadlock[56] and livelock detection [52] as well as the information-flow analysis [55] in the presence of communication paradigms. Linearity is also studied in these systems since the type of a channel carries information not only about the arity of the transmission, but also about its usage - the communication pattern. In other words it comprises information related to the sequence of transmission, the channel used for the communication, etc. Later developments extend [56] to handle deadlock detection with recursion and arbitrary networks of nodes [57]. This approach is developed for asynchronous CCS processes rather than π-calculus, and it is able to handle infinite-state systems due to the support for arbitrary number of nodes. Instead of reasoning on finite approximations of such system, the authors associate the compounded processes with terms of a basic recursive model, called lam. The lam terms are meant for binding the inter-channel dependencies which are then used to check for deadlocks by checking circularity over dependencies. The lam terms are collected by the type system, but the circularity detection is achieved using a sound and complete decision procedure, independent of the type system.

More or less in the same time with Kobayashi's type system [57], Padovani introduced another type system for deadlock-freedom [79] for linear π-calculus, which uses a form of channel polymorphism. This offers a better reasoning about unbounded dependency chains and

recursive types, at the cost of precision and support for non-linear channels.

Igarashi and Kobayashi [48] propose the Generic Type System, a general framework which can be instantiated to specific type systems for the π-calculus. The key idea of generic types is to express types and type environments as abstract processes, and then guarantee their deadlock-freedom and race-freedom by checking these type environments for the said properties. The generic type systems are augmented with general purpose subtyping relation and consistency condition which can be instantiated to cope with concrete properties. However, the general type system alone satisfies certain properties (eg. subject reduction), regardless of the subtyping relation and the consistency condition instantiation. The immediate benefit is that these generic properties need not be checked separately for each instantiation.

A more abstract system is that of Honda's [44] where processes are composable based on channel compatibility. The approach is built to support additive system. It is thus general enough to go beyond π-calculus, into supporting a wide range of other process calculi for the input language. It does suffer, however, at the precision level since it is unable to reason about inter-channel dependency, therefore it cannot detect deadlocks.

Following ideas from process algebras, Caires and Seco [11] propose behavioral separation for disciplining the interference of higher-order programs in the presence of concurrency and aliasing. Behavioural separation types build upon the knowledge of behavioural type theories, behavioral-spatial types [7], and separation logic [76, 80, 85, 50]. Specifically, behavioral types are used to model protocols by integrating behavioural operations such as parallel and sequential composition and intersection within a substructural type theory. Ideas from separation logic are used to model state abstraction in order to ensure safety in the presence of aliasing. This approach enforces fine-grained interference control without sacrificing compositionality, flexibility, and information hiding.

Dyadic Session Types. Some of the most intensely studied type systems for controlling communication protocols and its concurrency related properties are the session types, a refinement of the linear behavioral type systems described before. At the base of session types lies the observation that a communication-centred application is normally structured as a sequence of interactions (which could even be recursively described), where an interaction is either a single transmission or a choice of transmissions. This whole sequence of interactions form a conversation, or more formally called a sequence. The whole session is then abstracted as a type (session

types) used for checking the underlying program (abstracted in π-calculus) against the expected conversation structure (abstracted as a session type).

Initially proposed for systems with interaction between exactly two peers [45], session types have been extended by Honda et al. [47, 46] to systems with an arbitrary number of participants.

Even though dyadic session involves two participants, a binary session type describes a protocol from the perspective of only one peer, with support for branching and selection and even with the possibility to delegate the communication to a third party [45]. The views of the two participants are dual to each other [33]. Extensions of binary session types include adding primitives to handle exceptions across participants with guarantee for safety and liveness [15], adding support for multithreaded functional language in [70, 34, 77, 66, 95], and support for event-driven systems [58] by capturing nonblocking detection of transmission events and dynamically inspecting session types.

Moreover, there are also works which investigate the links between binary session and other disciplines: Caires and Pfenning introduce in [9, 10] a types system for π-calculus showing this types system to correspond to the standard sequent calculus proof system for dual intuitionistic linear logic. More precisely, they draw a tight correspondence between session types and intuitionistic linear propositions, and between π-calculus reductions and cut elimination steps. As an important consequence, they manage to provides a purely logical account of both shared and linear features of session types. In other words, several notions and features that normally require complex proofs, e.g. deadlock-freedom, follow for free from the theory of linear logic. Subsequently, Wadler [95] lays out a similar correspondence between a session typed process calculus (in the context of functional language) and classical linear logic, where session types correspond to propositions. Lindley and Morris [65] draw the reverse encoding, showing operational correspondence between propositions in classic linear logic and the session-types of the process calculus proposed by Wadler. Embedding from PCF augmented with a parameterised effect system into a π-calculus with session types, and the reverse, demonstrates that session types are able to express effects, and the effect system is powerful enough in order to express session types [77].

Multiparty Session Types. Binary session types have proved to be a lightweight, yet powerful formalism, rapidly adopted by a number of different systems. However, since real-world communication-centered systems normally use more than just two participants, it was important

to extend this formalism to an arbitrary number of participants, where simple composition of binary session is not precise enough to describe the communication. To this purpose, Honda et al. [47] generalize binary session types to multiparty session types, extending the binary session for π-calculus to multiparty asynchronous sessions for π-calculus.

In a multiparty session types calculus, a user is given the means to describe the global interactions between all the communication participants, as opposed to binary types where each session type described a single endpoint. The formalism doesn't simply discard the importance of individual channel endpoint description, but it is being pushed at a lower level, and mechanized by a projection mechanisms instead of being specified by the user. Therefore, once the user provides a global description of the communication, a projection algorithm computes the local view of each communicating party. The local party's abstract implementation (normally by means of π-calculus) is later checked against its corresponding local type.

One of the attractive feature of Honda's type system is that channels are quite general: asynchronous, shared across participants, FIFO style attached queue. Even though it offers general communication mediums, this channel implementation complicates the formalization due to possible race conditions, etc. A solution which is less general but offers a clearer formalization, avoiding race conditions altogether, is that of Coppo et al.[24] which binds the number of session channels created upon initialization to that of session roles. In other words, the number of channels is no longer arbitrary, but equal to the number of distinct communicating participants pairs in the system. Another solution to disambiguate the usage of a single channel across multiple participants is to label the communication actions [12]. This way, a single operation on a channel shared by multiple participants can be identified by the (channel, label) pair.

Carbone and Montesi [17] described an immediate application of multiparty session types in that of designing deadlock-free choreographies and π-calculus abstractions, where even if syntactically written in a sequence, certain transmissions which are not in a causal dependence can be swapped. The type theory comes without support for parallel composition.

Multiparty session types benefit from many kinds of extensions. Carbone [14] and Capecchi et al. [13] extend the work on exceptions for binary session types to integrate exceptions with multiparty session types. Bocchi et al. [2] made an attempt to describe the content of the exchanged message, and not just its type, by adding support for assertions to session types. A

natural application of session types would be that of describing the interactions within high-performance computing. To this purpose López et al. [67] use session types for typing MPI programs, in a context where the size of the problem is parameterised and the exchanged messages could have restrictions imposed, which go beyond type specifications. In the same line of works, Ng and Yoshida [71] introduce Paddle, a language designed for protocol specification with variable number of participants interacting in multiple dimensions. To ensure communication safety and deadlock freedom of MPI programs, Paddle also integrates a projection mechanisms for generating local protocols in the form of session types which are checked against the underlying code.

In showing a Curry-Howard correspondence between classical linear logic and multiparty session types [18, 19], Carbone et al. relate propositions to the local behaviour of a participant in a multiparty session type, proofs to processes, and proof normalisation to executing communications. More importantly, they generalize the notion of duality (relating two types) specific to binary session types to that of coherence(relating an arbitrary number of types) in the multiparty environment. To this purpose the authors propose Multiparty Classical Processes (MCP), a proof theory for reasoning on multiparty communications, providing thus a concise reconstruction of the foundations of Honda's multiparty session types. In a subsequent work [16], Carbone et al. refines further this correspondence, by introducing an intermediate calculus, that of Globally-governed Classical Processes (GCP). By doing so, the improved work preserves a closer resemblance to the the initially proposed multiparty session types [47], lifting the annotations from MCP to GCP, while also allowing support for parametric polymorphism.

Caires and Pérez present in [8] a formal relation between the theory of binary session types - rooted in linear logic - and that of multiparty session types - one based on automata theory. An immediate consequence of this relation is that it enables the analysis of the more complex multiparty protocols using the simpler type theory for binary protocols. The authors use protocol fidelity and deadlock freedom to showcases the proposed analysis techniques, reusing results from binary session types to check properties for multiparty protocols.

Comparison with the Current Thesis. *The type-based approaches mentioned this far provide a lightweight solution to the formalization of communication-centered programs. The behavioral types in the style of Kobayashi et. al [53, 54, 56, 52, 55, 56, 57] are effective in detecting*

circularity for programs which communicate using channels. However, their circularity detection mechanisms, whether in the recursive or the non-recursive model, only considers the inter-channel deadlock. As described by us in Sec. 3.6, intra-channel deadlock may also occur independently of the interaction with the other channels, and without creating circularity. Behavioral types do not account for this kind of deadlock detection. Moreover, even the more abstract behavioral types, namely the generic types [48], which are expressive enough to help detect safety issues even in the presence of delegation, do not handle protocols, therefore these approaches do not check programs (or their abstraction in the considered calculi) for protocol conformance, i.e. for the user intended behavior.

Session types add to the behavioral types the extra support for protocol specifications. This kind of type theory is able therefore to check whether a program meets its communication specification. As mentioned in the above survey, session types have been successfully used in the formalization of a wide range of communication related problems (such as protocol conformance, deadlock detection, race-freedom guarantee), both statically as well as dynamically, both for two-party as well as for multiparty communication, and in the context of different languages and process calculi. This thesis aims to go beyond type checking, into using the more expressive logical formulae to express protocol conformance in a static setting. Using logical formulae allows us to not only describe the type property of the exchanged message, but also its numerical properties, resource ownership transfer, or properties about the shape and the content of the message. Moreover, we aim to more precisely detect race-freedom: while session types detect communication race by means of causality analysis applied directly over the user-provided communication protocol, this thesis detects the communication race at the program code level, i.e. at the protocol implementation level rather than at the protocol specification level. By doing so, we not only reduce the number of channels needed to ensure race-freedom (which is unnecessarily high for ensuring this safety property using session types), but we also allow the communication developer to implement the protocol in a realistic scenario using both implicit synchronization as well as explicit synchronization mechanisms. Furthermore, we support the parallel composition of protocols in a more relaxed setting than session types, in that the same communication participant may be used across parallel protocols, as long as these parallel protocols respect the well-formedness constraints introduced in this thesis. In contrast, session

types support parallel composition only between disjoint elements [46]. In a nutshell, our approach to race-freedom transforms every user-provided protocol into a race-free one, which is later used to ensure that the protocol's implementation is race-free. Lastly, the logic proposed in this thesis is expressive enough to support (1) uniform choices, avoiding thus the need for special operators and special constructs to distinguish between internal and external choice as it is the case for the session type theory, and (2) transparent delegation due to the higher-order approach adopted in this thesis which requires no additional specification or language support other than the send and receive atomic constructs and communication primitives.

8.2 Logics with Channel Primitives

Hoare and O'Hearn [39] study the idea of coupling together the model theory of concurrent separation logic with that of Communicating Sequential Processes [38]. The work models communicating processes by using *trace semantics*, drawing an analogy between communicating channels and heap cells, and distinguishing between separation in space from separation in time. Furthermore, it describes the semantics of a message-passing language with dynamic manipulation of channels, where reasoning about the trace semantics of parallel composition is reduced to a composition operation on traces which distributes a channel's endpoints between concurrent processes. Our proposal shares the same idea of distinguishing between separation in space and separation in time, by using the $*$ and ; operators, respectively. However, their model differs significantly from ours since they rely on process algebras, while we propose an expressive logic based on separation logic able to also tackle memory management. Moreover, the trace model does not address the deadlock problem, which we handle by carefully checking the channels usage. And lastly, our communication model is more general, yet more precise, by accounting for explicit synchronizations as well.

Villard et al. [93] developed *Heap-Hop*, a sound proof system for copyless message passing managed by contracts. The proof system is integrated within a static analyzer which checks whether messages are safely transmitted, or in other words it checks the absence of ownership violations. Similar with our proposal, this work is also based on separation logic, a natural choice when dealing with memory management, and more generally when reasoning about resources (and resources ownership). As opposed to our proposal, the work of Villard et al. only discusses the case of two-party transmissions, whereas we extend our two-party approach

to also handle multi-party communications. Moreover, their communication model is limited to bidirectional channels, whereas our model is general enough to also capture buffered channels which may be shared among an arbitrary number of participants.

Similar to Villard et al., the logic of Bell et al. [1] is an extension of separation logic built for reasoning about multithreaded programs which communicate asynchronously through channels and which share memory. Their work however, different from Villard's and similar to our proposal, supports multiple players communicating through shared channel endpoints. To co-relate the proofs across processes, they keep track of transmitted messages in sets of ordered *histories*. Having a history for each process, when processes join, their system checks whether the sent values over a channel `x` corresponds to the receivers running in parallel and which were also using `x`. The check is done in a fashion which applies a matching rule on the idea that the last sent on `x` should match the last receive on `x`. Our approach does not need to track histories of values, since we rely on an event ordering constraint system which ensures not only the correct transmission order, but also that of explicit synchronization.

Leino et al. [64] advances Chalice [63] with support for communication paradigms. The paper proposes a verification methodology to prevent deadlocks of concurrent programs communicating via asynchronous shared channels, while permitting copy-less message passing and sharing memory via locks. Deadlock prevention is achieved by enforcing safe usage of channels - a receive is considered safe only when another thread holds an obligation to send - and safe global ordering - in the sense that each acquire and receive must obey a set of *permissions and credit transfers* rule. This work is similar to ours in the sense that it acknowledges the benefits but also the complications of combining different synchronization mechanisms. However, their design is cased around message passing and locks only, while the ordering system we propose is general enough to accommodate general explicit synchronization mechanisms along with the message passing. Moreover, they also ensure memory safety and absence of deadlocks, but do not incorporate the notion of communication protocols. Therefore, they do not verify whether a program follows a certain communication pattern or not.

A resource analysis for π-calculus by Turon and Wand [91] continues the work of Hoare and O'Hearn [39] by adding support for two kinds of channels, namely public and private, to express whether a channel is shareable or not. In this context they define two denotational models to reason about both communication safety and liveness. Similar to our approach, they also capture

the memory model, therefore they can also handle copy-less message passing. However, since their approach is based on the work of Hoare and O'Hearn, they also inherit both the strengths and the weaknesses of their precursor. Therefore, our benefits over their system also add up to to a better verification precision and a more expressive model.

8.3 Model Checking

Model checking [23, 31, 82, 21, 22] is a widely used technique to automatically verify the correctness properties of finite-state systems. Similar to our approach, model checking relies on specifications which are formulated in some precise mathematical language. It then checks whether a given model, also written in some precise formal language, satisfies a given logical formula, i.e. checks whether the model has the properties abstracted by the specification. The versatility and scalability of model checking have made it a favorite technique in many areas of verification, specially for concurrent reasoning. However, due to the complex and diverse factors which influence the correct and safe communication, protocols have rarely been the main subject of the verification problems tackled by model checking. In the next paragraphs we survey some notable works which built model checkers on top of some abstract calculi which support value passing such as the calculus of communicating systems (CCS) and π-calculus.

In [83], Ramakrishna et al. propose the usage of XSB tabled logic programming system to implement efficient model checkers. To demonstrate its efficiency the authors create a model checker for a CCS-like value-passing language. The resulted model checker is shown to be a lightweight solution (comprises about 200 lines of code) and efficient (in terms of time and memory usage) in analyzing CCS programs. Their solution is sufficient in checking both deadlock freedom and consensus properties in the context of concurrent programs synchronized via value passing. Different from our approach, they do not offer support for expressing a global view of the communication. Moreover, this model checker cannot cater for the common scenarios which use delegation. Finally, their work does not address type safety and race freedom.

Holzmann focuses in [43] on the verification of a concrete protocol, that of the i-protocol from GNU uucp version 1.04. He shows how to build and tune SPIN [42] such that it can efficiently verify that the i-protocol conforms its specification. Different from this approach, the solution proposed in the current thesis does not restrict the reasoning to a single kind of protocol, but presents an expressive solution for specifying and verifying less specific communication

protocols.

MMC [96] is a model checker for mobile systems specified in the style of the π-calculus. The operational semantics of the π-calculus is encoded as a Prolog relation, generating symbolic transition systems from agent definitions. The choice of Prolog as an engine for MMC together with the right abstraction, make MMC suitable for both monadic and polyadic versions of π-calculus. Moreover, MMC can also evaluate over the semantics of spi-calculus, therefore it can reason about cryptographic protocols. In contrast, the current thesis has not investigated cryptographic protocols, but, offering support for protocols where messages are abstracted as logical formulae, our approach supports a more precise reasoning about the exchanged resources, and of the communication safety, i.e. race freedom.

Chapter 9

CONCLUSIONS AND FUTURE WORK

We have proposed a solution for safe concurrency with synchronization via message-passing. More specifically, we have designed a logic with a rich set of assertions able to describe and verify communication scenarios which easily fit into today's programming style for distributed applications. To understand some of the concepts and difficulties in this subject matter we have initially designed a logic for dyadic protocols and discussed in-depth the advantages of a logic approach to session protocols. We have next advanced our proposal to the more complex case of multiparty protocols and integrated it in a logic which considers both implicit and explicit synchronization mechanisms. In this concluding chapter we summarize some of the main advantages of the current proposal.

Expressivity. Having a rich specification language, our proposed logic goes beyond the traditional type check, being able to express complex properties of the exchanged messages, such as numerical properties, memory shape, etc. Moreover, since our approach is based on separation logic we can naturally express copyless message passing, offering thus the possibility to have an efficient implementation of the communication tagged with memory safety guarantees. Another design feature which favors the expressivity of communication protocols is the use of disjunction. Disjunction treats choices in a uniform manner, as opposed to the more complex internal and external choice constructs which impose more restrictions on the underlying language. For example, we can use normal conditional constructs in the program implementing the

protocol, rather than having to add new specialized choice constructs as in the case of the types system approaches.

Precision. We have designed the entailment checking attached to the proposed logic such that it is flow-aware. In other words, our solver is able to distinguish between the case where the information flows outward and it needs to perform contra-variant subsumption check, and its dual, where the information is received and it checks for co-variant subsumption. An immediate benefit, besides correct message transmission, is that the communication is able to safely use specialized protocols even though it was designed and specified with a more general scenario in mind.

Automation. The careful formalization and implementation of our proof rules within an existing Hoare-style verification framework, lead to an elegant solution for the verification problem where the user is only expected to specify the pre- and post-condition of each method definition. The verifier automatically applies the well defined verification rules and either finds the proof of correctness or identifies the location of a problem. The solution is not complete, since we decided to sacrifice completeness for the sake of expressivity, but it does take a conservative path in declaring the program as unsafe should there be a scenario where it cannot find a proof of correctness.

Another important aspect in terms of the automation within the current proposal is that the user need only specify the global protocol (how the communicating peers interact with each other from a global perspective), while our projection algorithm automatically derives the local views corresponding to each participant. Moreover, we have also proposed a solution to automatically insert event ordering assumptions and proof obligations within the global protocol (where an event is defined as a unique send or receive). These ordering constraints are also reflected within the local view of each participant, since the automated projection has built-in rules for projecting the assumptions and proof obligations.

Separation in Space and Time. Separation logic, the foundation of the current proposal, offers an elegant access path to local reasoning by carefully tracking the resources allocation,

deallocation and usage. However, in a concurrent setting, where the order of events is sometimes crucial for the safety of the communication, separation in space is insufficient. To this purpose we introduced a novel constraint system to reason about the order of the events which influence the communication. The constraint systems mainly plies two ordering relations: a happens-before relation similar to Lamport's happened-before, and a communicates-before relation which correlates the events affiliated with the same transmission. Separation in time is therefore achieved by virtue of a rigorous manipulation of the two relations. This ordering system helped us design a solution to ensure race-free communication (where race is defined in terms of communicating processes competing for the same channel) and to relax the protocols to parallel composition where such race is not possible (concurrent transmissions) or where the communication is insensitive to race (nondeterministic protocols).

Realistic Concurrency Scenario. An overwhelming majority of the proposed solutions to safe message-passing programs make the unrealistic assumption that synchronization is achieved solely implicitly, guided by the order of sends and receives. However, this assumptions does not hold for most of today's systems which often require a combination of synchronization mechanisms in order to obtain efficient concurrency. Our solution aims to move towards this direction, where implicit and explicit synchronizations coexists. That is why by simply analyzing the global protocol as most of the existing techniques do, we cannot conclude whether the communication contains or not any race or deadlock scenarios. Differently, our proposed scheme first annotates the global protocol with event ordering constraints, and defer the race and deadlock detection to the verification process, which then checks whether the ordering proof obligations hold at the appropriate place.

Higher-Order Communication. Aiming for a minimalistic programming language, with little or no extra requirement other than what the standard C libraries offer, we have opted for a higher-order approach when designing the session logic. Specifically, the proposed logic is capable of naturally handle delegation through the use of higher-order channels, without the need of special delegation constructs. Moreover, the proposed solution also offers support for higher-order variables and predicates in order to support generic specification for method definitions. This way the program requires only one pair of generic specifications for the send and receive

methods, which will then cater for each calling context. What happens under the hood is that the prover facilitates the instantiation of the higher-order variable during the verification process guided by the communication specification at the caller's site.

9.1 Future Work

I conclude this thesis by pointing to some directions for future research avenues. First, to lift the burden of specifying each instance of explicit synchronization, it would be of great benefit to design a technique which automatically synthesizes the synchronization instruments and their specification. Second, to step further into the synthesis area and automation, it would be of significant value to analyze the program code without any annotation, and use inference methods, such as bi-abduction, to infer the communication pattern for each participant, and then compose all the results to obtain a global view of the communications. This approach is helpful when dealing with an existing code base without any prior communication specification. In a way, it is like the reverse of the current thesis approach. We could also investigate nondeterministic protocols further, by verifying the underlying communication implementation for commutative properties, i.e. if the messages received via nondeterministic interactions with the other peers are used in commutative computation. Last, to get even closer to a real concurrent model it would be interesting to investigate the possibility of formalizing dynamic communication patterns, with arbitrary spawning of new threads.

BIBLIOGRAPHY

[1] BELL, C. J., APPEL, A. W., and WALKER, D., "Concurrent Separation Logic for Pipelined Parallelization," in *International Static Analysis Symposium*, pp. 151–166, Springer, 2010.

[2] BOCCHI, L., HONDA, K., TUOSTO, E., and YOSHIDA, N., "A Theory of Design-by-Contract for Distributed Multiparty Interactions," in *CONCUR 2010*, vol. 6269 of *LNCS*, Springer, 2010.

[3] BORNAT, R., CALCAGNO, C., O'HEARN, P. W., and PARKINSON, M. J., "Permission Accounting in Separation Logic," in *POPL*, pp. 259–270, 2005.

[4] BOYLAND, J., "Checking Interference with Fractional Permissions," in *International Static Analysis Symposium*, pp. 55–72, Springer Berlin Heidelberg, 2003.

[5] BROOKES, S., "A semantics for concurrent separation logic," *Theor. Comput. Sci.*, vol. 375, pp. 227–270, Apr. 2007.

[6] BROOKES, S., O'HEARN, P. W., and REDDY, U., "The essence of reynolds," in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, (New York, NY, USA), pp. 251–255, ACM, 2014.

[7] CAIRES, L., "Spatial-Behavioral Types for Concurrency and Resource Control in Distributed Systems," *Theoretical Computer Science*, vol. 402, no. 2-3, pp. 120–141, 2008.

[8] CAIRES, L. and PÉREZ, J. A., "Multiparty Session Types within a Canonical Binary Theory, and Beyond," in *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*, pp. 74–95, Springer, 2016.

[9] CAIRES, L. and PFENNING, F., "Session types as intuitionistic linear propositions.," in *Proceedings of the 21st International Conference on Concurrency Theory*, pp. 222–236, Springer, 2010.

[10] CAIRES, L., PFENNING, F., and TONINHO, B., "Linear logic propositions as session types," *Mathematical Structures in Computer Science*, vol. 26, no. 3, pp. 367–423, 2016.

[11] CAIRES, L. and SECO, J. C., "The Type Discipline of Behavioral Separation," in *ACM SIGPLAN Notices*, vol. 48, pp. 275–286, ACM, 2013.

[12] CAIRES, L. and VIEIRA, H. T., "Conversation Types," in *European Symposium on Programming*, pp. 285–300, Springer, 2009.

[13] CAPECCHI, S., GIACHINO, E., and YOSHIDA, N., "Global Escape in Multiparty Sessions," *Mathematical Structures in Computer Science*, vol. 26, no. 02, pp. 156–205, 2016.

[14] CARBONE, M., "Session-based Choreography with Exceptions," *Electronic Notes in Theoretical Computer Science*, vol. 241, pp. 35–55, 2009.

[15] CARBONE, M., HONDA, K., and YOSHIDA, N., "Structured Interactional Exceptions in Session Types," in *International Conference on Concurrency Theory*, pp. 402–417, Springer, 2008.

[16] CARBONE, M., LINDLEY, S., MONTESI, F., SCHÜRMANN, C., and WADLER, P., "Coherence Generalises Duality: a logical explanation of multiparty session types," in *27 International Conference on Concurrency Theory (CONCUR'16)*, 2016.

[17] CARBONE, M. and MONTESI, F., "Deadlock-freedom-by-design: Multiparty Asynchronous Global Programming," *SIGPLAN Not.*, vol. 48, pp. 263–274, Jan. 2013.

[18] CARBONE, M., MONTESI, F., SCHÜRMANN, C., and YOSHIDA, N., "Multiparty session types as coherence proofs," *Acta Informatica*, vol. 54, no. 3, pp. 243–269, 2017.

[19] CARBONE, M., MONTESI, F., SCHÜRMANN, C., and YOSHIDA, N., "Multiparty Session Types as Coherence Proofs," in *26th International Conference on Concurrency Theory*, vol. 42 of *LIPIcs*, pp. 412–426, Schloss Dagstuhl, 2015.

[20] CHIN, W.-N., DAVID, C., NGUYEN, H. H., and QIN, S., "Automated verification of shape, size and bag properties via user-defined predicates in separation logic," *Sci. Comput. Program.*, vol. 77, pp. 1006–1036, Aug. 2012.

[21] CLARKE, E. M. and EMERSON, E. A., "Design and synthesis of synchronization skeletons using branching time temporal logic," in *Workshop on Logic of Programs*, pp. 52–71, Springer, 1981.

[22] CLARKE, E. M., EMERSON, E. A., and SISTLA, A. P., "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 8, no. 2, pp. 244–263, 1986.

[23] CLARKE, E. M., GRUMBERG, O., and PELED, D., *Model checking*. MIT press, 1999.

[24] COPPO, M., DEZANI-CIANCAGLINI, M., YOSHIDA, N., and PADOVANI, L., "Global Progress for Dynamically Interleaved Multiparty Sessions," *Mathematical Structures in Computer Science*, vol. 26, no. 02, pp. 238–302, 2016.

[25] DENIELOU, P.-M. and YOSHIDA, N., "Buffered Communication Analysis in Distributed Multiparty Sessions," in *CONCUR 2010*, vol. 6269 of *LNCS*, Springer, 2010.

[26] DENIÉLOU, P.-M. and YOSHIDA, N., "Dynamic multirole session types," in *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, (New York, NY, USA), pp. 435–446, ACM, 2011.

[27] DEZANI-CIANCAGLINI, M., DE'LIGUORO, U., and YOSHIDA, N., "On progress for structured communications," in *International Symposium on Trustworthy Global Computing*, pp. 257–275, Springer, 2007.

[28] DEZANI-CIANCAGLINI, M., MOSTROUS, D., YOSHIDA, N., and DROSSOPOULOU, S., "Session types for object-oriented languages," in *Proceedings of the 20th European Conference on Object-Oriented Programming*, ECOOP'06, (Berlin, Heidelberg), pp. 328–352, Springer-Verlag, 2006.

[29] DINSDALE-YOUNG, T., BIRKEDAL, L., GARDNER, P., PARKINSON, M., and YANG, H., "Views: Compositional Reasoning for Concurrent Programs," in *ACM SIGPLAN Notices*, vol. 48, pp. 287–300, ACM, 2013.

[30] DINSDALE-YOUNG, T., DODDS, M., GARDNER, P., PARKINSON, M. J., and VAFEIADIS, V., "Concurrent abstract predicates," in *Proceedings of the 24th European Conference on Object-oriented Programming*, ECOOP'10, (Berlin, Heidelberg), pp. 504–528, Springer-Verlag, 2010.

[31] EMERSON, E. A. and CLARKE, E. M., "Characterizing correctness properties of parallel programs using fixpoints," in *International Colloquium on Automata, Languages, and Programming*, pp. 169–181, Springer, 1980.

[32] FENG, X., FERREIRA, R., and SHAO, Z., "On the relationship between concurrent separation logic and assume-guarantee reasoning," in *Proceedings of the 16th European Symposium on Programming*, ESOP'07, (Berlin, Heidelberg), pp. 173–188, Springer-Verlag, 2007.

[33] GAY, S. and HOLE, M., "Subtyping for Session Types in the pi-Calculus," *Acta Informatica*, vol. 42, no. 2-3, pp. 191–225, 2005.

[34] GAY, S. J. and VASCONCELOS, V. T., "Linear Type Theory for Asynchronous Session Types," *Journal of Functional Programming*, vol. 20, no. 01, pp. 19–50, 2010.

[35] GIUNTI, M. and VASCONCELOS, V. T., "A linear account of session types in the pi calculus," in *International Conference on Concurrency Theory*, pp. 432–446, Springer, 2010.

[36] GOTSMAN, A., BERDINE, J., COOK, B., RINETZKY, N., and SAGIV, M., "Local Reasoning for Storable Locks and Threads," in *Asian Symposium on Programming Languages And Systems*, pp. 19–37, Springer, 2007.

[37] HOARE, C. A. R., "An Axiomatic Basis for Computer Programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[38] HOARE, C. A. R., "Communicating Sequential Processes," in *The origin of concurrent programming*, pp. 413–443, Springer, 1978.

[39] HOARE, T. and O'HEARN, P., "Separation Logic Semantics for Communicating Processes," *Electronic Notes in Theoretical Computer Science*, vol. 212, pp. 3–25, 2008.

[40] HOBOR, A., APPEL, A. W., and NARDELLI, F. Z., "Oracle Semantics for Concurrent Separation Logic," in *European Symposium on Programming*, pp. 353–367, Springer, 2008.

[41] HOBOR, A. and GHERGHINA, C., "Barriers in Concurrent Separation Logic," in *European Symposium on Programming*, pp. 276–296, Springer, 2011.

[42] HOLZMANN, G. J., "The model checker spin," *IEEE Transactions on software engineering*, vol. 23, no. 5, pp. 279–295, 1997.

[43] HOLZMANN, G. J., "The engineering of a model checker: the gnu i-protocol case study revisited.," in *International SPIN Workshop on Model Checking of Software*, pp. 232–244, Springer, 1999.

[44] HONDA, K., "Composing Processes," in *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 344–357, ACM, 1996.

[45] HONDA, K., VASCONCELOS, V. T., and KUBO, M., "Language primitives and type discipline for structured communication-based programming," in *Proceedings of the 7th European Symposium on Programming: Programming Languages and Systems*, ESOP '98, (London, UK, UK), pp. 122–138, Springer-Verlag, 1998.

[46] HONDA, K., YOSHIDA, N., and CARBONE, M., "Multiparty asynchronous session types," in *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, (New York, NY, USA), pp. 273–284, ACM, 2008.

[47] HONDA, K., YOSHIDA, N., and CARBONE, M., "Multiparty Asynchronous Session Types," *Journal of the ACM*, vol. 63, pp. 1–67, 2016.

[48] IGARASHI, A. and KOBAYASHI, N., "A Generic Type System for the Pi-Calculus," *Theoretical Computer Science*, vol. 311, no. 1, pp. 121 – 163, 2004.

[49] ISHTIAQ, S. and O'HEARN, P. W., "BI as an Assertion Language for Mutable Data Structures," in *ACM POPL*, (London), Jan. 2001.

[50] ISTHIAQ, S. and O'HEARN, P. W., "BI as an assertion language for mutable data structures," in *ACM POPL*, (London), Jan. 2001.

[51] KIM, D. and RINARD, M. C., "Verification of semantic commutativity conditions and inverse operations on linked data structures," in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, (New York, NY, USA), pp. 528–541, ACM, 2011.

[52] KOBAYASHI, N., "Type Systems for Concurrent Processes: From Deadlock-freedom to Livelock-freedom, Time-boundedness," in *IFIP International Conference on Theoretical Computer Science*, pp. 365–389, Springer, 2000.

[53] KOBAYASHI, N., "A Type System for Lock-free Processes," *Information and Computation*, vol. 177, no. 2, pp. 122–159, 2002.

[54] KOBAYASHI, N., "Type Systems for Concurrent Programs," in *Formal Methods at the Crossroads. From Panacea to Foundational Support*, pp. 439–453, Springer, 2003.

[55] KOBAYASHI, N., "Type-based Information Flow Analysis for the π-calculus," *Acta Informatica*, vol. 42, no. 4-5, pp. 291–347, 2005.

[56] KOBAYASHI, N., "A New Type System for Deadlock-free Processes," in *CONCUR 2006–Concurrency Theory*, pp. 233–247, Springer Berlin Heidelberg, 2006.

[57] KOBAYASHI, N. and LANEVE, C., "Deadlock Analysis of Unbounded Process Networks," *Information and Computation*, vol. 252, pp. 48–70, 2017.

[58] KOUZAPAS, D., YOSHIDA, N., HU, R., and HONDA, K., "On Asynchronous Eventful Session Semantics," *Mathematical Structures in Computer Science*, vol. 26, no. 02, pp. 303–364, 2016.

[59] KRISHNASWAMI, N. R., BIRKEDAL, L., and ALDRICH, J., "Verifying event-driven programs using ramified frame properties," in *Proceedings of the 5th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '10, (New York, NY, USA), pp. 63–76, ACM, 2010.

[60] KRISHNASWAMI, N. R. and BENTON, N., "Ultrametric semantics of reactive programs," in *Proceedings of the 2011 IEEE 26th Annual Symposium on Logic in Computer Science*, LICS '11, (Washington, DC, USA), pp. 257–266, IEEE Computer Society, 2011.

[61] LAMPORT, L., "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[62] LANGE, J., NG, N., TONINHO, B., and YOSHIDA, N., "Fencing off go: Liveness and safety for channel-based programming," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, (New York, NY, USA), pp. 748–761, ACM, 2017.

[63] LEINO, K. R. M. and MÜLLER, P., "A Basis for Verifying Multi-Threaded Programs," in *European Symposium on Programming*, pp. 378–393, Springer, 2009.

[64] LEINO, K. R. M., MÜLLER, P., and SMANS, J., "Deadlock-Free Channels and Locks," in *European Symposium on Programming*, pp. 407–426, Springer, 2010.

[65] LINDLEY, S. and MORRIS, J. G., "A Semantics for Propositions as Sessions," in *European Symposium on Programming Languages and Systems*, pp. 560–584, Springer, 2015.

[66] LINDLEY, S. and MORRIS, J. G., "Embedding Session Types in Haskell," in *Proceedings of the 9th International Symposium on Haskell*, pp. 133–145, ACM, 2016.

[67] LÓPEZ, H. A., MARQUES, E. R., MARTINS, F., NG, N., SANTOS, C., VASCONCELOS, V. T., and YOSHIDA, N., "Protocol-Based Verification of Message-Passing Parallel Programs," in *ACM SIGPLAN Notices*, vol. 50, pp. 280–298, ACM, 2015.

[68] LOZES, E. and VILLARD, J., "Shared Contract-Obedient Channels," *Science of Computer Programming*, vol. 100, pp. 28 – 60, 2015. Selected Papers from the 5th Interaction and Concurrency Experience (ICE 2012).

[69] NADEN, K., BOCCHINO, R., ALDRICH, J., and BIERHOFF, K., "A Type System for Borrowing Permissions," in *ACM SIGPLAN Notices*, vol. 47, pp. 557–570, ACM, 2012.

[70] NEUBAUER, M. and THIEMANN, P., "An Implementation of Session Types," in *International Symposium on Practical Aspects of Declarative Languages*, pp. 56–70, Springer, 2004.

[71] NG, N. and YOSHIDA, N., "Pabble: Parameterised Scribble for Parallel Programming," in *Parallel, Distributed and Network-Based Processing (PDP), 2014 22nd Euromicro International Conference on*, pp. 707–714, IEEE, 2014.

[72] NGUYEN, H. H. and CHIN, W.-N., "Enhancing program verification with lemmas," in *CAV*, pp. 355–369, 2008.

[73] NIELSON, F. and NIELSON, H. R., "From cml to process algebras," in *International Conference on Concurrency Theory*, pp. 493–508, Springer, 1993.

[74] NIELSON, F. and NIELSON, H. R., "From CML to its Process Algebra," *Theoretical Computer Science*, vol. 155, no. 1, pp. 179–219, 1996.

[75] O'HEARN, P. W., REYNOLDS, J., and YANG, H., "Local Reasoning about Programs that Alter Data Structures," in *Proceedings of the 15th Annual Conference of the European Association for Computer Science Logic*, (Paris, France), Sept. 2001.

[76] O'HEARN, P., YANG, H., and REYNOLDS, J., "Separation and Information Hiding," in *ACM POPL*, (Venice, Italy), Jan. 2004.

[77] ORCHARD, D. and YOSHIDA, N., "Effects as Sessions, Sessions as Effects," in *ACM SIGPLAN Notices*, vol. 51, pp. 568–581, ACM, 2016.

[78] O'HEARN, P. W., "Resources, Concurrency, and Local Reasoning," *Theoretical computer science*, vol. 375, no. 1-3, pp. 271–307, 2007.

[79] PADOVANI, L., "Deadlock and Lock Freedom in the Linear π-calculus," in *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, p. 72, ACM, 2014.

[80] PARKINSON, M. J. and BIERMAN, G. M., "Separation logic and abstraction," in *ACM POPL*, pp. 247–258, 2005.

[81] PARKINSON, M. and BIERMAN, G., "Separation logic and abstraction," in *ACM SIGPLAN Notices*, vol. 40, pp. 247–258, ACM, 2005.

[82] QUEILLE, J.-P. and SIFAKIS, J., "Specification and verification of concurrent systems in cesar," in *International Symposium on programming*, pp. 337–351, Springer, 1982.

[83] RAMAKRISHNA, Y., RAMAKRISHNAN, C., RAMAKRISHNAN, I., SMOLKA, S. A., SWIFT, T., and WARREN, D. S., "Efficient model checking using tabled resolution," in *International Conference on Computer Aided Verification*, pp. 143–154, Springer, 1997.

[84] REDDY, U. S. and REYNOLDS, J. C., "Syntactic control of interference for separation logic," in *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, (New York, NY, USA), pp. 323–336, ACM, 2012.

[85] REYNOLDS, J., "Separation Logic: A Logic for Shared Mutable Data Structures," in *IEEE Logic in Computer Science*, (Copenhagen, Denmark), July 2002.

[86] REYNOLDS, J. C., "Separation logic: A logic for shared mutable data structures," in *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, pp. 55–74, IEEE, 2002.

[87] REYNOLDS, J. C., *An Overview of Separation Logic*, pp. 460–469. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008.

[88] SCALAS, A., DARDHA, O., HU, R., and YOSHIDA, N., "A linear decomposition of multiparty sessions for safe distributed programming," in *ECOOP*, vol. 74, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[89] SILBERSCHATZ, A., GALVIN, P. B., and GAGNE, G., *Operating System Concepts*. Wiley Publishing, 8th ed., 2008.

[90] SVENDSEN, K., BIRKEDAL, L., and PARKINSON, M., "Joins: a Case Study in Modular Specification of a Concurrent Reentrant Higher-Order Library," in *European Conference on Object-Oriented Programming*, pp. 327–351, Springer, 2013.

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

[92] VAFEIADIS, V. and PARKINSON, M. J., "A Marriage of Rely/Guarantee and Separation Logic," in *CONCUR*, pp. 256–271, 2007.

[93] VILLARD, J., LOZES, É., and CALCAGNO, C., "Proving Copyless Message Passing," in *Asian Symposium on Programming Languages and Systems*, pp. 194–209, Springer, 2009.

[94] VILLARD, J., LOZES, E., and CALCAGNO, C., "Tracking heaps that hop with heap-hop," in *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'10, (Berlin, Heidelberg), pp. 275–279, Springer-Verlag, 2010.

[95] WADLER, P., "Propositions as Sessions," *ACM SIGPLAN Notices*, vol. 47, no. 9, pp. 273–286, 2012.

[96] YANG, P., RAMAKRISHNAN, C., and SMOLKA, S. A., "A logical encoding of the π-calculus: Model checking mobile processes using tabled resolution," in *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pp. 116–131, Springer, 2003.

GLOSSARY OF SYMBOLS AND NOTATIONS

$\oplus(\Psi)$	assumption, page 59
$\ominus(\Psi)$	guard, page 59
$\mathtt{e}$	expression, page 21
$\sim$	duality, page 29
$\mathtt{Z}$	per channel specification, page 78
$\overline{\mathtt{L}}$	the specification of channels in dyadic protocols, page 33
$\tilde{\mathtt{c}}$	program channel / channel endpoint, page 92
$\mathtt{c}$	logical channel, page 92
Π	set of ordering constraints, page 73
$\mathtt{L}$	per endpoint specification, page 78
$(\mathtt{G})\downarrow_{\mathtt{All}}$	projection of shared ordering relations, page 79
$(\Upsilon)\downarrow_{\mathtt{c}}$	per channel projection, page 79
$(\mathtt{G})\downarrow_{\mathtt{P}}$	per party projection, page 79
Υ	per party specification, page 78
S	residual formula/state, page 18
ρ	substitution, page 18
$\mathtt{c}_1 \mapsto \mathtt{c}_2$	$\mathtt{c}_2$ waits for/depends on $\mathtt{c}_1$, page 52
$\mathcal{P}$	program, page 11
κ	heap formula, page 16
π	pure formula, page 16
Δ	spatial formula, page 16

Φ	specification formula, page 16
$\mathtt{emp}$	empty heap, page 14
$\mathtt{v}{\mapsto}\mathtt{d}\langle\mathtt{v}^*\rangle$	points-to, page 14
$\kappa * \kappa$	separating conjunction, page 14
$\mathtt{V}$	higher-order variable, page 16
$\mathtt{A}^{\oplus}$	a set of ordering assumptions, page 75
$\mathtt{A}^{\ominus}$	a set of ordering proof obligations, page 75
$\mathcal{S}$	protocol summary, page 75
$\mathcal{C}(\mathtt{c}, \overline{\mathtt{L}})$	channel specification for dyadic protocols, page 33
$\mathcal{C}(\mathtt{c}, \mathtt{P}, \mathtt{L})$	channel specification for multiparty protocols, page 92
$\mathtt{TR}(\mathtt{G})$	returns all the transmissions in $\mathtt{G}$, page 67
$\mathtt{P}^{(i)}$	an event, page 68
$\mathtt{EV}(\mathtt{G})$	returns all the events in $\mathtt{G}$, page 68
$\mathtt{seq}(\mathtt{G})$	function which returns all the pairs of sequenced transmissions in $\mathtt{G}$, page 69
$i_1 \prec i_2$	sequenced transmissions, page 69
$\mathtt{Adj}(i_1, i_2)$	adjacent transmissions, page 69
$\mathtt{Adj}^+(i_1, i_2)$	linked transmissions, page 69
$!\mathtt{r} \cdot \Delta$	the sending of a message $\mathtt{r}$ described by Δ , page 33
$?\mathtt{r} \cdot \Delta$	the receiving of a message $\mathtt{r}$ described by Δ , page 33
$\overline{\mathtt{L}}_1; \overline{\mathtt{L}}_2$	sequential composition, page 33
$\overline{\mathtt{L}}_1 \vee \overline{\mathtt{L}}_2$	choice, page 33
$\mathtt{S} \xrightarrow{i} \mathtt{R} : \mathtt{c}\langle\mathtt{v} \cdot \Delta\rangle$	in transmission i, the sender $\mathtt{S}$ transmits to the receiver $\mathtt{R}$ a message $\mathtt{v}$ described by Δ over channel $\mathtt{c}$, page 59

APPENDIX A

The auxiliary functions of the examples in Chapter 3 are depicted in Fig. A.1.

```
int get_prod_id()
  requires  emp
  ensures   res > 0;

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

Prod get_prod(int id)
  requires id > 0
  ensures  res↦Prod⟨id⟩;

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

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

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

Figure A.1: Summaries of Auxiliary Functions

The Buyer implementation of Buyer-Seller-Shipper protocol, Sec. 3.4, is as follows:

```
1   void buyer(Channel c, int id, Double budget, Addr a)
2      requires C(c, buy_sp)
3      ensures  C(c, emp)
4  { send(c,id);
5    Double price = receive(c);
6    if (price <= budget){
7      send(c,1);
8      send(c,a);
9      Date sd = receive(c);
10   } else {
11     send(c,0)
12   }
13 }
```