Imperial College London
Department of Computing

Multiparty Session Types for Dynamic Verification of Distributed Systems

Rumyana Neykova

Supervised by Nobuko Yoshida

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of Imperial College London, February 2017

Declaration

I herewith certify that all material in this thesis which is not my own work has been properly acknowledged.

Rumyana Neykova

Copyright Declaration

Abstract

In large-scale distributed systems, each application is realised through interactions among distributed components. To guarantee safe communication (no deadlocks and communication mismatches) we need programming languages and tools that structure, manage, and policy-check these interactions. Multiparty session types (MPST), a typing discipline for structured interactions between communicating processes, offers a promising approach. To date, however, session types applications have been limited to static verification, which is not always feasible and is often restrictive in terms of programming API and specifying policies.

This thesis investigates the design and implementation of a runtime verification framework, ensuring conformance between programs and specifications. Specifications are written in Scribble, a protocol description language formally founded on MPST. The central idea of the approach is a dynamic monitor, which takes a form of a communicating finite state machine, automatically generated from Scribble specifications, and a communication runtime stipulating a message format.

We extend and apply Scribble-based runtime verification in manifold ways. First, we implement a Python library, facilitated with session primitives and verification runtime. We integrate the library in a large cyber-infrastructure project for oceanography. Second, we examine multiple communication patterns, which reveal and motivate two novel extensions, asynchronous interrupts for verification of exception handling behaviours, and time constraints for enforcement of real-time protocols. Third, we apply the verification framework to actor programming by augmenting an actor library in Python with protocol annotations. For both implementations, measurements show Scribble-based dynamic checking delivers minimal overhead and allows expressive specifications.

Finally, we explore a static analysis of Scribble specifications as to efficiently compute a safe global state from which a monitored system of interacting processes can be recovered after a failure. We provide an implementation of a verification framework for recovery in Erlang. Benchmarks show our recovery strategy outperforms a built-in static recovery strategy, in Erlang, on a number of use cases.

This thesis is dedicated to the memory of Kohei Honda. Kohei, I consider myself extremely lucky that I had the chance to know you. Once you wrote: ”*Take a look at the attached note, I wish to hear your ideas: this is intended to be a basis of what we can grow together*”. Wherever you are, please have a look at my ideas in the note that follows . . .

Acknowledgments

This has been an incredible journey. I was lucky to meet and work with true visionaries, to form dear friendships, to develop my skills as a researcher. For that, I would like to first and foremost thank my supervisor Nobuko Yoshida, who provided me with constant guidelines, support, and supervision. This thesis is dedicated to the late Kohei Honda, who not only believed in me but made me believe in myself. I thank him for his enthusiasm towards research, his immense knowledge, and for our tireless discussions. My sincere thanks also go to my examiners for their positive comments and for their proposed corrections which improved my work, and to all the reviewers of the papers which I co-authored.

Appreciation is also expressed to Susan Eisenbach for her early advice as a mentor of my master degree and guiding me to find the right path in academia. I am extremely grateful to Sophia Drossopoulou which inspired me to be a researcher. I would also like to thank Pierre-Malo Deniélou, whose early work inspired the inception of this thesis. Romain Demangeon and Raymond Hu, that helped me with technical insights and writing. Ray, I would have not made it without you. Thank you for always being kind, for your feedback, for putting up with the early inexperienced version of me. I also want to thank my coauthor and friend, Laura Bocchi.

I would also like to thank VMware (Rita Tavilla, Steve Muir) and the RabbitMQ team (Matthias Radestock, Alexis Richardson, Ask Solem) in particular, for supporting this Ph.D. and for giving me this opportunity. Thank you, Ask for spending hours on discussions on actors, for patiently listening about session types, for always answering my questions, for reviewing my code, I learned so much from you. Thanks to Steve and Rita for inviting me to the VMware seminar in Boston. Gratitude is also extended to Mathew Arrott and Stephen Henrie from the OOI team for our discussions, and for the unique experience of working with them. Thank you, Stephen, for explaining me the OOI codebase, listening to my ideas, helping with any technical struggle.

Next I would like to thank all minions of MRG: Juliana Franco, Julien Lange, Tiago Cogumbreiro, Bernardo Toninho, Nicholas Ng, Raymond Hu, Alceste Scalas, Dominic Orchard. You are so much more than colleagues. You are my academic family, but most importantly my friends. You showed me that research is not only about theory or practice, it is about people and ideas. Your technical excellence, your dedication to research, your passion, albeit you do not always show it, inspire me every day. Thank you for being the great model of excellence that you are. Thank you, Tiago, that you were always there to listen and talk about dependent types. Thank you, Bernardo and Julien for making me (hopefully) troll-resilient! Thank you, Ju, for discussing my premature ideas. I also thank my fellow office mates (Florian, Fred, Alex, Bjorn and Victor), who supported me in the beginning of my Ph.D., when I was confused, unsure and alone.

A Ph.D. is also a personal journey, it is a road that is not always easy, hence I would like to thank my dear friends, who were with me in my darkest (and brightest) moments. Endless thanks to Ina, for always being there for me no matter what and no matter when, for her insights and support, analysis and friendship; Tanya for teaching me how to enjoy life and for being the incredible person she is; Miro and Miro and Miro for being an inspiration, I am glad you are back in my life, Ani and Sisi for their constant support, Uza for always being ready to engage in intellectual discussions; Jo, for the wonderful weekends spend together and our endless talks; Christian for our PhD struggle talks; Tamer, Salva and Mark for giving me the chance to be a subwarden.

Finally, special recognition goes out to my family - my mother Galia, my sister Iva, my grandmother Vesi and my late grandfather Ivan. I would also like to thank Georgi, who has always been concerned with my future, ready to give me advice and support. I am extremely lucky to have an amazing family. I love you more than anything. Ivushi, you are the most incredible sister someone could dream of. Mum, you have given me endless love and made me the person I am today.

Thank you!

Contents

List of Figures

List of Tables

1. Introduction

1.1. Motivation and Objectives

1.1.1. Distributed Programming

One of the main engineering challenges in distributed systems is ensuring that a composition of independently implementable components will form a correct system as a whole. To tackle this challenge we need a sound verification model, which prevents communication errors and ambiguous specifications. We demonstrate the importance of addressing these objectives below.

Communication errors. Coordination errors such as communication mismatches are hard to debug. Undetected errors in one process may affect the correctness of other processes, or even deadlock the whole system. In addition, components are often developed in isolation using programming languages and tools that do not validate compositionality of processes. For example, sockets and remote procedure calls (RPC) are two popular paradigms for programming distributed systems. The former approach is untyped, while the latter cannot guarantee a specific order when sequences of interactions are executed. The actor programming model, which (re)gains attention in the research community and in the industry, addresses the problem of untyped interactions [TDJ13]. Most actor implementations provide static typing within a single actor (as surveyed in Section 6.6). However, communications between actors – the complex communication patterns that are most likely to deadlock – are not checked.

Ambiguous specifications. Although there is a widespread recognition of the need for programming languages and tools that structure, manage, and policy-check interactions between distributed components, there is no widespread specification language for writing communication protocols. Internet protocols (such as SMTP and HTTP) are specified as text descriptions in the request for comments

(RFC) documents. Such documents not only contain ambiguities, but are cumbersome to implement and verify. For application protocols, sequence diagrams are used as a descriptive tool. Sequence diagrams depict a communication flow as an ordered sequence of message passing among communicating participants that represent particular roles in a communication protocol. Although intuitive, they lack a formal basis and allow protocols with ambiguous behaviour.

1.1.2. Session Types

Multiparty Session Types (MPST) [HYC08], one of the formalisms that have been proposed to structure interactions and to reason over communicating processes and their behaviour, offer a promising approach towards a rigorous specification and verification model for distributed systems. The core idea of MPST is that the structure of a conversation (sequence of interactions) is abstracted as a type through an intuitive syntax, which is then used as a basis for validating programs through an associated type discipline. A description of a multiparty protocol from a global view-point is provided using a global type, and then the global type is projected to end-point types against which processes can be efficiently type-checked. Using static type checking, multiparty session types ensure non-trivial communication properties such as communication safety (there is no communication mismatch), protocol fidelity (communications proceed as expected), and progress among the communicating components.

The application of the theoretical MPST techniques to current practice, however, faces a few obstacles.

- Session type checking is typically designed for languages with first-class communication and concurrency primitives, whereas our collaborations use mainstream engineering languages such as Python and Java. These languages either lack the features required to make static session typing tractable, or, in the case of dynamically typed languages like Python, may simply be unsuited to this approach. Certain programming techniques can further complicate static analysis; for example, the obfuscation of control flow in event-driven programming, a common paradigm in distributed systems.
- Distributed systems are often heterogeneous in nature, meaning that a range of languages and platforms may be involved in the implementation of a given system, as well as third-party components or services for which the source code is unavailable for a static type checking. Dynamic verification by com-

munication monitoring allows us to verify MPST safety properties directly for mainstream languages in a more scalable way.

- Certain protocol specification features, such as assertions on specific message values, can be evaluated precisely at runtime, whereas static treatments are often more conservative.

1.2. Challenges of the Approach

This thesis tackles the challenges that are mentioned above. More precisely, the aim of the thesis is to design, implement and investigate the applicability of a runtime verification framework, ensuring compliance between programs and MPST specifications. In this endeavour we explore several research themes, each of which induces a set of research questions.

The first question concerns finding a suitable specification language, facilitated with a user-friendly syntax, appealing to software architects and developers, and having a runtime representation, permitting for an efficient trace checking. The Scribble protocol description language [scrb], which is based on MPST, is a promising choice. However, despite Scribble being described as the practical incarnation of MPST [scrb], a formal correspondence between the two has not been proven in the literature.

The second question concerns the expressiveness of MPST. For example, some application-level protocols contain real-time constraints (such as timeouts), and exception handling, which are not supported by the current theory. *How can the session type theory and the accompanying tools be extended to account for popular communication patterns?*

The third question concerns the usability of the framework. Most mainstream programming languages do not support message-style concurrency. Applying session types to mainstream languages and also to other concurrency paradigms must be done in a uniform and non-disruptive way. *How can we provide communication libraries that are expressive and yet non-disruptive?*

The final question concerns extending the application domain of MPST. Session types theory is primarily studied and applied as a verification approach. In contrast, data types are not only useful as a way to structure, describe and verify data properties, but they are also applied for optimising the memory layout of a program. Similarly, session types may be amenable for optimisations of failures. *In the case of a failure, can we, by exploiting the structure of the interaction, guide* *and minimise the propagated information and the required communication?*

1.3. Contributions

The main contributions of this thesis are the design, implementation, and evaluation of a session type-based runtime verification framework. First, using this framework, developers write protocols using the specification language Scribble, which is formally founded on MPST. Then programs are implemented using an API for session programming. A monitor, attached to each distributed program, translates Scribble specifications to Communicating Finite State Machines (CFSMs) to check that program traces comply with protocol specifications. The approach of this thesis addresses the challenges discussed above as follows.

Formalising Scribble. We give a background overview of the protocol description language Scribble [scrb]. Then we present a core syntax of Scribble protocols and define operational semantics. We prove a correspondence between Scribble and MPST. The correspondence allows us to examine and prove the soundness of our runtime framework. More precisely, we give a translation between local Scribble protocols and CFSMs using the correspondence between local types and CFSM, given in [DY12], and prove that a global protocol is equivalent to a configuration of CFSMs.

A runtime monitor tool for dynamic verification of distributed systems. We present the first design and implementation of session types for dynamic verification. The library, developed in Python, exposes an API of core primitives for session programming. Benchmarks show the feasibility of the approach and reasonable performance overhead for protocol checking. In addition to microbenchmarks, we have also integrated the library in a large cyber-infrastructure project for oceanography.

Session types extensions. We extend the framework with exception handling capabilities and time constraints. The former allow us to reason about fault-tolerant systems, while the latter enlarge the domain of the protocols that can be expressed.

More concretely, we present the implementation and formalisation of a new construct for verifying *asynchronous multiparty session interrupts*. Asynchronous session interrupts express communication patterns in which the behaviour of the roles following the default flow through a protocol segment may be overruled by one or more other roles concurrently raising asynchronous interrupt messages.

We also embed primitives for timed protocol specifications into the Scribble toolchain, and give an implementation in the Scribble toolchain of an algorithms for consistency checking of real-time properties. In addition, we exploit the encoding from timed-MPST into Communicating Timed Automata, given in [BYY14a], to produce run-time monitors for programs implemented using the Python API. We implement several monitoring modes, with different degrees of intervention of the monitor in the ongoing interactions, from just observing interactions to attempts to fix time mismatches. To assess the usability of timed Scribble we have gathered a wider (albeit not exhaustive) portfolio of properties of timed distributed protocols from the literature, and we have provided a number of timed patterns which demonstrate how these properties can be expressed in Scribble.

Session types for actor programming. We prove the generality of our MPST framework by showing that Scribble protocols offer wider usage, in particular for actor programming. This is the first design and implementation of session types and their dynamic verification toolchain in an actor library. Our programming model is grounded in three new design ideas: (1) use Scribble protocols and their relation to finite state machines for specification and runtime verification of actor interactions; (2) augment actor messages and their mailboxes dynamically with protocol (role) information; and (3) propose an algorithm based on virtual routers (protocol mailboxes) for the dynamic discovery of actor mailboxes within a protocol. We implement a session actor library in Python as to demonstrate the applicability of the approach through examples.

Recoverable session types. We propose a recovery framework based on MPSTs and show how session type-based analysis is used to provide correct fault-tolerant recovery. Specifically, we statically analyse the communication flow of a program, given as a multiparty protocol, to extract the causal dependencies between processes and to localise failures. We formalise a recovery strategy, using communicating automata, and implement several use cases as to show that the proposed strategy is more efficient for common message-passing protocols than the Erlang all-for-one supervision.

1.3.1. Publications and Detailed Contributions of the Author

The following papers were published as a result of the research done for the requirements of this thesis and are its primary contributing sources. The papers are presented in chronological order. For each paper my contribution is given. The relevance of each work to this thesis is given through the thesis chapter correspondence.

[Ney13] ***Rumyana Neykova***. Session Types Go Dynamic or How to Verify Your Python Conversations. In PLACES, 2013

Author's Contribution The paper introduces an early design and implementation of a runtime verification framework based on session types that is the basis of this thesis. All of the presented material is my own work with the supervision and advice of my supervisor.

Corresponding part This preliminary work comprises a significant part of Chapter 3 and has evolved into the material presented in the following paper.

[HNYD13] Raymond Hu, ***Rumyana Neykova***, Nobuko Yoshida, Romain Demangeon and Kohei Honda: Practical Interruptible Conversations - Distributed Dynamic Verification with Session Types and Python. In Runtime Verification, 2013.

Author's Contribution The paper presents a Scribble-based runtime verification framework with support for exception handling behaviour. I contributed writing, implementation and benchmarking material to the working draft of this paper, as well as proof reading of other sections.

Corresponding part The design of interruptible extensions of this work forms the basis of Chapter 4.

[NYH13] ***Rumyana Neykova***, Nobuko Yoshida and Raymond Hu: SPY: Local Verification of Global Protocols. In Runtime Verification, 2013

Author's Contribution The paper introduces a Scribble tool with facilities for projection. The tool presented is a shared work by me and Raymond Hu. The writing is done by me.

Corresponding part The example and the tool are discussed in Chapter 3.

[YHNN13] Nobuko Yoshida, Raymond Hu, ***Rumyana Neykova*** and Nicholas Ng: The Scribble Protocol Language. In TGC, 2013

Author's Contribution The paper presents an overview of the protocol description language Scribble, which is the specification language used by the framework presented in this thesis. I contributed writing Section 3.5, which explains the Python API for session programming.

Corresponding part Although the work undertaken towards this paper has influenced parts of the thesis, the material is not included here directly.

[NY14a] ***Rumyana Neykova*** and Nobuko Yoshida: Multiparty Session Actors. In PLACES, 2014

Author's Contribution The work presents the design and implementation of an API for actor programming. All of the presented material is my own work with the supervision and advice of my coauthor.

Corresponding part This preliminary work comprises a significant part of Chapter 6 and has evolved into the material presented in the following paper.

[NY14b] ***Rumyana Neykova*** and Nobuko Yoshida: Multiparty Session Actors. In COORDINATION, 2014

Author's Contribution The work presents the design and implementation of a programming API with support of runtime verification of actor programs. All of the presented material is my own work with the supervision and advice of my coauthor.

Corresponding part This paper forms the basis of Chapter 6.

[NBY14] ***Rumyana Neykova***, Laura Bocchi and Nobuko Yoshida: Timed Runtime Monitoring for Multiparty Conversations. In BEAT, 2014

Author's Contribution This work presents Scribble-based runtime verification with support for time constraints. The design, implementation and examples is done entirely by me. Writing is shared between the authors of the paper.

Corresponding part This paper forms the basis of Chapter 5.

[HHN$^+$14a] Kohei Honda, Raymond Hu, ***Rumyana Neykova***, Tzu-Chun Chen, Romain Demangeon, Pierre-Malo Denilou and Nobuko Yoshida: Structuring Communication with Session Types. In Concurrent Objects and Beyond, 2014

Author's Contribution I have contributed writing the paper. In particular, I contributed writing Section 3.5 on how to write programs with session types. I also implemented the example presented in this section.

Corresponding part Although the work undertaken towards this paper has influenced parts of the thesis, the material is not included here directly.

[DHH$^+$15] Romain Demangeon, Kohei Honda, Raymond Hu, ***Rumyana Neykova*** and Nobuko Yoshida: Practical interruptible conversations: distributed dynamic verification with multiparty session types and Python. Formal Methods in System Design 46(3), (2015)

Author's Contribution This is a journal version of [HNYD13] and it extends the initial submition with additional explanations, event-driven implementation and theory for interrupts in session types. The additional implementation and writing are done by me. I have been involved in the deign of formalising the implemented extension along with Romain Demangeon.

Corresponding part The event-driven example from the paper is included in Chapter 4 and the theoretical part forms the basis of Section 4.3.3.

[NY17a] **Rumyana Neykova**, Nobuko Yoshda Let it Recover: Multiparty-protocol induced recovery. In Compiler Construction, 2017

Author's Contribution The work presents a static analysis based on multiparty session types that can efficiently compute a safe global state from which a system of interacting processes should be recovered after a failure. All of the presented material is my own work with the supervision and advice of my coauthor.

Corresponding part This paper forms the basis of Chapter 7.

[NY17b] **Rumyana Neykova**, Nobuko Yoshida: Multiparty Session Actors. In Logical Methods in Computer Science, 2017

Author's Contribution This is a journal version of [NY14b] and it extends the initial submition with additional explanations, and evaluation of the framework. The work is entirely done by me with with the supervision and advice of my coauthor.

Corresponding part Examples from the article are included in Chapter 6.5.

[NY17c] **Rumyana Neykova**, Nobuko Yoshida: Timed Runtime Monitoring for Multiparty Conversations. Formal Aspects of Computing, 2017

Author's Contribution This is a journal version of [NBY14] and it extends the initial submission with the design and implementation of an algorithm for checking consistency of timed global protocols and evaluation on the usability of our specification language to express commonly occurring temporal patterns in distributed protocols. We also present the formal syntax for timed global protocols along with an explanation of the constructs and the projection mechanism, that were omitted from [NBY14]

Corresponding part This work forms the basis of Section 5.

1.3.2. Thesis Outline

Chapter 2. Scribbling Multiparty Session Types In this chapter we present the formal semantics of the Scribble language. We then show that the encoding of Scribble local and global protocols to global and local session types is behaviour-preserving, and gives the translation between local Scribble protocols and Communicating Finite State Machines (CFSMs).

Chapter 3. From Scribble Specifications to Runtime Monitors In this chapter we present an Scribble-based runtime verification and embed session types in a dynamically-typed language Python. The advantages, expressiveness, and performance of dynamic protocol checking are demonstrated through use cases and benchmarks. We also present the integration of our framework in a large cyberinfrastructure project for oceanography.

Chapter 4. Monitoring Interruptible Systems In this chapter we present an extension of Scribble to support the specification of asynchronously interruptible conversations. We then implement a concise API for conversation programming with interrupts in Python that enables dynamic verification of safety properties. Then we prove that our framework ensures the global safety of a system in the presence of asynchronous interrupts.

Chapter 5. Monitoring Real-Time Systems In this chapter we extend the verification framework presented in the previous chapters to support verification and enforcement of real-time constraints. We extend (1) Scribble with clocks, resets, and clock predicates in order to constrain the times at which interactions occur; (2) the Python API with time primitives; (3) the runtime monitors with checks for time constraints. To evaluate the practicality of the real-time extension, we express and verify four categories of widely used temporal patterns from use cases in the literature.

Chapter 6. Monitoring Actor Programs In this chapter we study the applicability of MPST protocols to verification of actor programs. We associate sessions to actors by introducing minimum additions to the model such as the notion of actor roles and protocol mailboxes. We demonstrate our framework by designing and implementing a session actor library in Python and its runtime verification mechanism. The presented benchmark results demonstrate that the runtime checks induce negligible overhead. We evaluate the applicability of our verification framework to actor interaction specification by implementing twelve examples from an actor benchmark suit.

Chapter 7. Recoverability for Monitored MPST Processes In this chapter we propose a static analysis based on multiparty session types that can efficiently compute a safe global state from which a system of interacting processes should be recovered. We formalise our recovery algorithm and show it provides lower communication cost (only affected processes are notified) and overall execution time (only required states are repeated). On top of our analysis, we design and implement a runtime framework in Erlang where failed processes and their dependencies are restarted from a consistent state. We evaluate our recovery framework on a set of message-passing benchmarks and use cases.

2. Scribbling Multiparty Session Types

Scribble [scrb, HMB$^+$11] is a protocol description language, formally based on the multiparty session type theory [HYC08, BCD$^+$08]. A protocol in Scribble represents an agreement on how participating systems interact with each other. More precisely, it specifies a format and a predefined order for messages to be exchanged.

The name of the language embodies the motivation for its creation, as explained by the following quote from the inventor of the Scribble language Kohei Honda:

> *The name (Scribble) comes from our desire to create an effective tool for architects, designers and developers alike to quickly and accurately write down protocols.*

The development of Scribble is a result of a persistent dialogue between researchers and industry partners. Currently Scribble tools are applied to verification of multiple languages (Java [HY16], Python [HNYD13], MPI [NCY15]). However, it all started from a simple idea, from a "hello world" in a verbose language, from the desire to structure a chaos, from a whisper in Kohei Honda's mind, as best described in the following fragment from a speech of his in 2007:

> *All great ideas of architectural construction come from that unconscious moment, when you do not realise what it is, when there is no concrete shape, only a whisper which is not a whisper, an image which is not an image, somehow it starts to urge you in your mind, in so small a voice but how persistent it is, at that point you start scribbling.*

Although Scribble is often refereed as "the practical incarnation of multiparty session types (MPST)" [scrb, HMB$^+$11], a formal correspondence between the two is not proven in the literature. The language semantics are not formalised either.

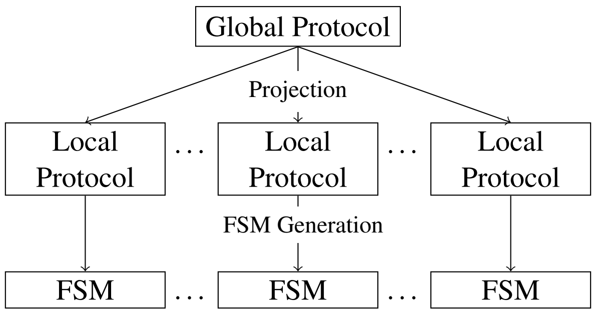


Figure 2.1.: Scribble development methodology

Contributions and outline In this chapter we give a brief overview of Scribble. Then we present the semantics of Scribble protocols and prove a correspondence between Scribble and MPST. The outline of this chapter is given below:

Section 2.1 gives an overview of Scribble and explains the Scribble framework.

Section 2.2 presents the formal semantics of the Scribble language.

Section 2.3 proves the encoding of Scribble local and global protocols to global and local session types to be behaviour-preserving.

Section 2.4 discusses results on MPST applied to Scribble and gives the translation between local Scribble protocols and Communicating Finite State machines (CFSMs) [DY12].

2.1. Scribble Overview

2.1.1. Example: Online Wallet

Scribble protocols describe an abstract structure of message exchanges between *roles*: roles abstract from the actual identity of the endpoints that may participate in a run-time conversation instantiating the protocol. Scribble enables the description of abstract interaction structures including asynchronous message passing, choice, and recursion.

Here we demonstrate the basic Scribble constructs via an example. The example is a simple version of an online payment service. Fig. 2.2 lists the global type as a Scribble protocol, `OnlineWallet`. The first line declares, under the name `OnlineWallet`, the Scribble global protocol and the two participating roles. The protocol has a recursion at the top-level. In each iteration, the `Server` (`S`) sends the `Client` (`C`) the current balance and the overdraft limit for `Client`'s account. The `Balance` message has an `int` payload; similarly for the `OverdraftLimit`. `C`

```
global protocol OnlineWallet(role Server,role Client){
  rec LOOP {
    Balance(int) from Server to Client;
    OverdraftLimit(int) from Server to Client;
    choice at Client {
      MakePayment(string, int) from Client to Server;
      continue LOOP;
    } or {
      CloseAccount from Client to Server;
    } or {
      Quit from Client to Server;
} } }
```

Figure 2.2.: A protocol specification in Scribble

then has the choice to make a payment, close the account or quit this session. The `MakePayment` payload contains a `string` and an `int` for the payee identity and the amount to pay.

2.1.2. Scribble Toolchain

Fig. 2.1 gives an abstract overview of the Scribble verification process. From a global Scribble protocol, the toolchain produces (1) a set of local protocols or (2) a set of finite state machines (FSMs). The main actions performed by the Scribble toolchain are specified below:

1. (**well-formedness check**) A global Scribble protocol is verified for correctness to ensure that the protocol is *well-formed*, which intuitively means that a protocol describes a meaningful interaction, beyond the basic syntax defined by the language grammar. This is necessary because some of the protocols are unsafe or inconsistent even if they follow the grammar. For example, two choice branches from the same sender to the same receiver with the same message signature lead to ambiguity at the receiver side. More precisely, a protocol is well-formed if local protocols can be generated for all of its roles, i.e the projection function is defined for all roles. The formal definition of projection is given in Definition 2.3.12. Here we give intuition as to what the main syntactic restrictions are:
   - in each branch of a choice the first interaction (possibly after a number of unfoldings) is from the same sender (e.g., `A`) and to the same set of receivers.
   - in each branch of a choice the labels are pair-wise distinguished (i.e.,

```
local protocol OnlineWallet at Server(
  self Server, role Client) {
  rec LOOP {
    Account(int, int) to Client;
    choice {
      MakePayment(string, int) from Client;
      continue LOOP;
    } or {
      CloseAccount() from Client;
    } or {
      Quit() from Client;
} } }
```

Figure 2.3.: `OnlineWallet` local protocol for `Server`

protocols are deterministic).

2. (**projection**) A global Scribble protocol is projected to a set of local protocols. More precisely, a local Scribble protocol is generated per each role declared in the definition of the global protocol. Local protocols correspond to local (MPST) types, they describe interactions from the viewpoint of a single entity. They can be used directly by a type checker to verify that an endpoint code implementation complies to the interactions prescribed by a specification. Fig. 2.3 lists the Scribble local protocol `OnlineWallet` projected for role `Server`.
3. (**FSM generation**) An alternative representation of a local protocol can be given in the form a communicating finite state machine (FSM). This representation is useful for runtime verification. Specifically, at runtime the traces emitted by a program are checked against the language accepted by the FSM.

An implementation of Java-based and Python-base Scribble tools for projection and validation [scrb], as well as static verification for various languages can be found in [ses, NYH12, SS, NYH13].

2.2. Syntax and Semantics of Scribble

2.2.1. Scribble Global Protocols

The syntax of Scribble global protocols is given by the grammar below.

Definition 2.2.1 (Scribble Global Protocols)

$$
\begin{array}{lcll}
\mathtt{P} & ::= & \mathtt{global\ protocol\ pro\ (role\ A_1,...,role\ A_n)\{G\}} & \textit{specification}\\
\mathtt{G} & ::= & \mathtt{a(S)\ from\ A\ to\ B;G} & \textit{interaction}\\
 & \mid & \mathtt{choice\ at\ p\ \{G\}\ or\ \ldots\ or\ \{G\}} & \textit{choice}\\
 & \mid & \mathtt{rec\ t\ \{G\}} & \textit{recursion}\\
 & \mid & \mathtt{continue\ t} & \textit{call}
\end{array}
$$

Protocol names are ranged over by pro. A (global) specification P declares a protocol with name pro, involving a list ($\mathtt{A_1,..A_n}$) of roles, and prescribing the behaviour in G. The other constructs are explained below:

- An interaction a(S) from p to q;G specifies that a message a(S) should be sent from role A to role B and that the protocol should then continue as prescribed by the continuation G. Messages are of the form a(S) with a being a label and S being the constant type of exchanged messages (such as real, bool and int).
- A choice choice at A {G} or ... or {G} specifies a branching where role A chooses to engage in the interactions prescribed by one of the options G. The decision itself is an internal process to role A, i.e. how A decides which scenario to follow is not specified by the protocol.
- A recursion rec t {G} defines a scope with a name t and a body G. Any call continue t occurring inside G executes another recursion instance (if continue t is not in an appropriate scope than it remains idle).

Formal semantics of global protocols The formal semantics of global protocols characterises the desired/correct behaviour of the roles in a multiparty protocol. We give the semantics for Scribble protocols as a Labelled Transition System (LTS). The LTS is defined over the following set of transition labels:

$$\ell ::= \mathtt{AB!a(S)} \mid \mathtt{AB?a(S)}$$

Label AB!a(S) is for a send action where role A sends to role B a message a(S). Label AB?a(S) is for a receive action where B receives (i.e., collects from the queue associated to the appropriate channel) message a(S) that was previously sent by A. We define the subject of an action, modelling the role that has the responsibility of performing that action, as follows:

$$subj(\mathtt{AB!a(S)}) = \mathtt{A} \qquad subj(\mathtt{AB?a(S)}) = \mathtt{B}$$

As, due to asynchrony, send and receive are two distinct actions, the LTS shall also model the intermediate state where a message has been sent but it has not been yet received. To model these intermediate states we introduce the following additional global Scribble interaction:

$$\texttt{transit}:\texttt{a(S) from A to B;G}$$

to describe the state in which a message a(S) has been sent by A but not yet received by B. We call *runtime* global protocol a protocol obtained by extending the syntax of Scribble with these intermediate states.

The transition rules are given in Fig. 2.4. Rule $\lfloor\textsc{send}\rfloor$ models a sending action; it produces a label AB!a(S). The sending action yields a state in which the global protocol is in an intermediate state.

Rule $\lfloor\textsc{recv}\rfloor$ models the dual receive action, from an intermediate state to a continuation G. Rule $\lfloor\textsc{choice}\rfloor$ continues the execution of the protocol as a continuation of one of the branches. Rule $\lfloor\textsc{rec}\rfloor$ is standard and unfolds recursive protocols.

The remaining rules deserve a detailed explanation.

Due to asynchrony and distribution, in a particular state of a Scribble global protocol it may be possible to trigger more than one action. For instance, the protocol in (2.1) allows two possible actions: AB!a(S) or CD!a(S).

$$\begin{array}{l}\texttt{a(S) from A to B;}\\ \texttt{a(S) from C to D;}\end{array} \tag{2.1}$$

This is due to the fact that the two send actions are not *causally related* as they have different subjects (which are independent roles). We want the semantics of Scribble to allow, in the state with protocol (2.1), not only the first action that occurs syntactically (e.g., AB!a(S)) but also any action that occurs later, syntactically, but it is not causally related with previous actions in the protocol (e.g., CD!a(S)). Rule $\lfloor\textsc{async1}\rfloor$ allows exactly this. CD!a(S), which occurs syntactically later than AB!a(S) to possibly occur before. In fact, the LTS allows (2.1) to take one of these two actions: either AB!a(S) by rule $\lfloor\textsc{send}\rfloor$ or CD!a(S) is allowed by $\lfloor\textsc{async1}\rfloor$. Rule $\lfloor\textsc{async2}\rfloor$ is similar to $\lfloor\textsc{async1}\rfloor$ but caters for intermediate states, and is illustrated by the protocol in (2.2), obtained from (2.1) via transition AB!a(S) by rule $\lfloor\textsc{send}\rfloor$.

$$\begin{array}{l}\texttt{transit :a(S) from A to B;}\\ \texttt{a(S) from C to D;}\end{array} \tag{2.2}$$

$$\mathtt{a(S)}\ \mathtt{from}\ \mathtt{A}\ \mathtt{to}\ \mathtt{B};\mathtt{G} \xrightarrow{\mathtt{AB!a(S)}} \mathtt{transit}:\mathtt{a(S)}\ \mathtt{from}\ \mathtt{A}\ \mathtt{to}\ \mathtt{B};\mathtt{G} \qquad \lfloor\text{SEND}\rfloor$$

$$\mathtt{transit}:\mathtt{a(S)}\ \mathtt{from}\ \mathtt{A}\ \mathtt{to}\ \mathtt{B};\mathtt{G} \xrightarrow{\mathtt{AB?a(S)}} \mathtt{G} \qquad \lfloor\text{RECV}\rfloor$$

$$\frac{i \in \{1,..,n\} \quad \mathtt{G_i} \xrightarrow{\ell} \mathtt{G'_i}}{\mathtt{choice\ at\ A}\ \{\mathtt{G_1}\}\ \mathtt{or}\ \ldots\ \mathtt{or}\ \{\mathtt{G_n}\} \xrightarrow{\ell} \mathtt{G'_i}} \qquad \lfloor\text{CHOICE}\rfloor$$

$$\frac{\mathtt{G} \xrightarrow{\ell} \mathtt{G'} \qquad \mathtt{A},\mathtt{B} \notin subj(\ell)}{\mathtt{a(S)}\ \mathtt{from}\ \mathtt{A}\ \mathtt{to}\ \mathtt{B};\mathtt{G} \xrightarrow{\ell} \mathtt{a(S)}\ \mathtt{from}\ \mathtt{A}\ \mathtt{to}\ \mathtt{B};\mathtt{G'}} \qquad \lfloor\text{ASYNC1}\rfloor$$

$$\frac{\mathtt{G} \xrightarrow{\ell} \mathtt{G'} \qquad \mathtt{B} \notin subj(\ell)}{\mathtt{transit}:\mathtt{a(S)}\ \mathtt{from}\ \mathtt{A}\ \mathtt{to}\ \mathtt{B};\mathtt{G} \xrightarrow{\ell} \mathtt{transit}:\mathtt{a(S)}\ \mathtt{from}\ \mathtt{A}\ \mathtt{to}\ \mathtt{B};\mathtt{G'}} \qquad \lfloor\text{ASYNC2}\rfloor$$

$$\frac{\mathtt{G}[\mathtt{rec\ t}\ \{\mathtt{G}\}/\mathtt{continue t}] \xrightarrow{\ell} \mathtt{G'}}{\mathtt{rec\ t}\ \{\mathtt{G}\} \xrightarrow{\ell} \mathtt{G'}} \qquad \lfloor\text{REC}\rfloor$$

Figure 2.4.: Labelled transitions for global protocols.

The protocol in (2.2) can execute either $\mathtt{AB?a(S)}$ by rule $\lfloor\text{RECV}\rfloor$, or $\mathtt{CD!a(S)}$ by rule $\lfloor\text{ASYNC2}\rfloor$.

2.2.2. Scribble Local Protocols

Scribble local protocols describe a session from the perspective of a single participant. The syntax of Scribble local protocols is given below.

Definition 2.2.2 (Scribble Local Protocols)

$$\begin{array}{lcl}
\mathtt{L} & ::= & \mathtt{local\ protocol\ pro\ at\ A_i(role\ A_1,...,role)\{T\}} \\
\mathtt{T} & ::= & \mathtt{a(S)\ to\ B;T} \\
 & \mid & \mathtt{a(S)\ from\ B;T} \\
 & \mid & \mathtt{choice\ at\ A}\ \{\mathtt{T_1}\}\ \mathtt{or} \ldots \mathtt{or}\ \{\mathtt{T_n}\} \\
 & \mid & \mathtt{rec\ pro}\ \{\mathtt{T}\} \\
 & \mid & \mathtt{continue\ pro} \\
 & \mid & \mathtt{end}
\end{array}$$

The construct $\mathtt{a(S)\ to\ B;T}$ models a send action from $\mathtt{A}$ to $\mathtt{B}$; the dual local protocol is $\mathtt{a(S)\ from\ B;T}$ that models a receive action of $\mathtt{A}$ from $\mathtt{B}$. The other

$$\mathtt{a(S)}\ \mathtt{to}\ \mathtt{B;T} \xrightarrow{\mathtt{AB!a(S)}} \mathtt{T} \qquad \lfloor\text{SEND}\rfloor$$

$$\mathtt{a(S)}\ \mathtt{from}\ \mathtt{B;T} \xrightarrow{\mathtt{BA?a(S)}} \mathtt{T} \qquad \lfloor\text{RECV}\rfloor$$

$$\frac{i \in \{1,..,n\} \quad \mathtt{T_i} \xrightarrow{\ell} \mathtt{T_i'}}{\mathtt{choice\ at\ A}\ \{\mathtt{T_1}\}\ \mathtt{or}\ldots\{\mathtt{T_n}\} \xrightarrow{\ell} \mathtt{T_i'}} \qquad \lfloor\text{CHOICE}\rfloor$$

$$\frac{\mathtt{T}[\mathtt{rec}\,\mathtt{t}\ \{\mathtt{T}\}/\mathtt{continue}\,\mathtt{t}] \xrightarrow{l} \mathtt{T'}}{\mathtt{rec}\,\mathtt{t}\ \{\mathtt{T}\} \xrightarrow{l} \mathtt{T'}} \qquad \lfloor\text{REC}\rfloor$$

Figure 2.5.: Labelled transitions for local protocols (from A's point of view).

protocol constructs are similar to the corresponding global protocol constructs. Recursive variables are guarded in the standard way, i.e. they only occur under a prefix. For convenience we will, sometimes, use the notation

$$\mathtt{choice\ at}\ \{\mathtt{a_i(S_i)}\ \mathtt{from}\ \mathtt{A;T_i}\}_{i\in\{1,..,n\}}$$

to denote Scribble local protocols of the form

$$\mathtt{choice\ at}\ \{\mathtt{a_1(S_1)}\ \mathtt{from}\ \mathtt{A;T_1}\}\ \mathtt{or}\ldots\mathtt{or}\ \{\mathtt{a_n(S_n)}\ \mathtt{from}\ \mathtt{A;T_n}\}$$

with $n > 1$. or of the form (i.e., $n = 1$)

$$\mathtt{a(S)}\ \mathtt{from}\ \mathtt{A;T}$$

Decomposing global protocols into a set of local protocols is called *projection*. Projection is a key mechanism to enable distributed enforcement of global properties. Projection preserves the interaction structures and message exchanges required for the target role to fulfil his/her part in the conversation. We give the formal definition on projection later in this chapter (Definition 2.3.12) when we introduce a normal (canonical) form for global protocols.

Formal semantics of local protocols The LTS for local protocols is defined by the rules in Fig. 2.5, which use the same labels as the global semantics in Fig. 2.4. The rules $\lfloor$SEND$\rfloor$, $\lfloor$RECV$\rfloor$, $\lfloor$CHOICE$\rfloor$, $\lfloor$REC$\rfloor$ are similar to the respective rules for global protocols. We do not need rules for modelling asynchrony as each participant is assumed to be single threaded.

Formal semantics of configurations The LTS in Fig. 2.5 describes the behaviour of each single role in isolation. In the rest of this section we give the semantics of systems resulting from the composition of Scribble local protocols and communication channels. Given a set of roles $\{1,\dots,n\}$ we define configurations $(\mathtt{T_1},\dots,\mathtt{T_n},\vec{w})$ where $\vec{w} ::= \{w_{ij}\}_{i\neq j\in\{1,\dots,n\}}$ are unidirectional, possibly empty (denoted by ε), unbounded FIFO queues with elements of the form $\mathtt{a}(\mathtt{S})$.

Definition 2.2.3 (Semantics of configurations) *The LTS of* $(\mathtt{T_1},\dots,\mathtt{T_n},\vec{w})$ *is defined as follows:*
$(\mathtt{T_1},\dots,\mathtt{T_n},\vec{w}) \xrightarrow{\ell} (\mathtt{T'_1},\dots,\mathtt{T'_n},\vec{w}')$ *iff: :*
(1) $\mathtt{T_B} \xrightarrow{\mathtt{AB!a(S)}} \mathtt{T'_B} \wedge w'_{\mathtt{AB}} = w_{\mathtt{AB}} \cdot \mathtt{a(S)} \wedge (ij \neq \mathtt{AB} \Rightarrow w_{ij} = w'_{ij} \wedge \mathtt{T_i} = \mathtt{T'_i})$
(2) $\mathtt{T_B} \xrightarrow{\mathtt{AB?a(S)}} \mathtt{T'_B} \wedge \mathtt{a(S)} \cdot w'_{\mathtt{AB}} = w_{\mathtt{AB}} \wedge (ij \neq \mathtt{AB} \Rightarrow w_{ij} = w'_{ij} \wedge \mathtt{T_j} = \mathtt{T'_j})$
with $\mathtt{A},\mathtt{B},i,j \in \{1,\dots,n\}$.

In (1) the configuration makes a send action given that one of the participants can perform that send action. Case (1) has the effect of adding a message, that is sent, to the corresponding queue. In (2) the configuration makes a receive action given that one of its participant can perform such an action and that the message being received is currently stored in the corresponding queue. Thus, (2) has the effect of removing the message received from the queue.

2.3. Correspondence between Scribble and MPST

In this section we show that a trace of a global protocol corresponds exactly to a trace of its projected local protocols. Correspondence is important as it ensures that the composition of processes, each implementing some local protocol, will behave as prescribed by the original global specification. In the context of MPST, this property is known as *soundness of the projection* (Theorem 3.1, [DY13]) and has already been proven for global types as defined in [DY13]. As explained in Section 2.3.1 a translation of this result to Scribble, however, is not obvious.

Fig. 2.6 gives a high level overview of the results presented in this section. First, we discuss the (syntactic) differences between global types and global protocols. We present a normal form for global protocols such that a Scribble global protocol in a normal form can be encoded into (MPST) global types and the translation is semantic preserving. We then prove a similar correspondence between Scribble local protocols and (MPST) local types. The soundness of the projection of

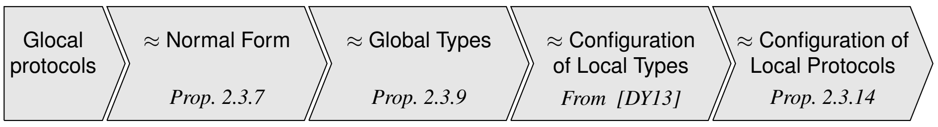


Figure 2.6.: Workflow of proving soundness of the projection

global protocol then follows from soundness of the projection of MPST global types (Theorem 3.1 from [DY13]).

2.3.1. Scribble Normal Form

The syntax of global session types (thereafter global types), in the framework of MPSTs given in [DY13], is presented below:

Definition 2.3.1 (Global Types)

$$G \quad ::= \quad \mathtt{A} \rightarrow \mathtt{B} : \{\mathtt{a}_i \langle \mathtt{S}_i \rangle . G_i\}_{i \in I} \;\mid\; \mu \mathtt{t}.G \;\mid\; \mathtt{t} \;\mid\; \mathtt{end}$$

The syntax of global types is very similar to the syntax of Scribble global protocols in Section 2.2 except: (1) Scribble does not cater for delegation and higher order protocols whereas global types do; and (2) the choice and interaction protocols are two separated constructs in Scribble while they are modelled as a unique construct in global types and (3) opposed to MPST, Scribble allows unguarded choice. The case of (2) is a consequence of the specific focus of Scribble as a protocol design language directed at practitioners that are familiar with e.g., Java notation, who proved to find this notation friendlier [scrb, HMB$^+$11, YHNN13, HHN$^+$14b]. Regarding (3) the choice construct in Scribble directly supports recursion and choice while in MPST the choice is always directly followed by an interaction. In the following section we explain that these differences are indeed syntactic and do not affect the soundness of the language.

First, we observe that a Scribble syntax with a guarded and a singleton choice directly corresponds to MPST. We refer to a Scribble protocol, where all choices are guarded, as a Scribble Normal Form (SNF). Later we show that there is a behaviour preserving translation between a well-formed Scribble protocol and its normal form.

The Scribble Normal Form (SNF) for global protocols is given below:

Definition 2.3.2 (Scribble Normal Form (SNF))

$$\begin{array}{lll} \mathtt{G} & ::= & \mathtt{choice\ at\ A}\ \{\mathtt{N_i}\}_{i\in\{1,..,n\}} \mid \mathtt{N} \mid \mathtt{rec\ t}\ \{\mathtt{N}\} \\ \mathtt{N} & ::= & \mathtt{a(S)\ from\ A\ to\ B;G} \mid \mathtt{continue\ t} \mid \mathit{end} \end{array}$$

The SNF does not contain nested choice and does not allow recursion after the choice.

The encoding of Scribble global protocols to SNF requires two auxiliary functions: $\mathtt{flatten}(\mathtt{G})$ and $\mathtt{unfold}(\mathtt{G})$. The latter transforms nested choice into one choice, while the former performs one unfolding of a recursion. First, we focus on $\mathtt{flatten}(\mathtt{G})$, given below.

Definition 2.3.3 (Flatten) *For all local Scribble protocols* $\mathtt{G}$*, we define* $\mathtt{flatten}(\mathtt{G})$ *in the following way:*

$$\mathtt{flatten}(\mathtt{G}) = \begin{cases} \mathtt{flatten}(\mathtt{G_0}) \cup \ldots \cup \mathtt{flatten}(\mathtt{G_n}) & \text{if } \mathtt{G} = \mathtt{choice\ at\ A}\ \{\mathtt{G_i}\}_{i\in\{1,..,n\}} \\ \mathtt{G} & \text{otherwise} \end{cases}$$

We demonstrate $\mathtt{flatten}(\mathtt{G})$ in the example in Fig. 2.7. The example on the right shows a Scribble protocol with nested choice, while its `flatten` form is shown on the right.

```
choice at A {
  choice at A
    {msg1 from A to B;}
  or
    {msg2 from A to B;}}
or
  {msg3 from A to B; }
```

```
choice at A
  {msg1 from A to B;}
or
  {msg2 from A to B;}
or
  {msg3 from A to B;}
```

Figure 2.7.: Scribble protocol (left), and its `flatten` form (right).

Next we give the definition of $\mathtt{unfold}(\mathtt{G})$

Definition 2.3.4 (Unfold) *For all global Scribble protocols* $\mathtt{G}$*, we define* $\mathtt{unfold}(\mathtt{G})$ *by recursion on the structure of* $\mathtt{G}$ *in the following way:*

$$\mathtt{unfold}(\mathtt{G}) \begin{cases} \mathtt{unfold}(\mathtt{G'}[\mathtt{rec\ t}\ \{\mathtt{G'}\}/\mathtt{continue\ t}]) & \text{if } \mathtt{G} = \mathtt{rec\ t}\ \{\mathtt{G'}\} \\ \mathtt{G} & \text{otherwise} \end{cases}$$

Thus for any recursive type, unfold is the result of repeatedly unfolding the top level recursion until a non-recursive type constructor is reached. Because we assume that recursive types are contractive, unfold terminates.

Finally, we give the encoding below:

Definition 2.3.5 (Encoding of Global Protocols to SNF) *The encoding $\langle\rangle$ from (Scribble) global protocols to SNF types is given below:*

$$\begin{array}{lcl}
\langle \mathtt{end} \rangle & = & \mathtt{end} \\
\langle \mathtt{rec}\ \mathtt{t}\ \{\mathtt{G}\} \rangle & = & \mathtt{unfold}(\mathtt{rec}\ \mathtt{t}\ \{\langle \mathtt{G} \rangle\}) \\
\langle \mathtt{a(S)}\ \mathtt{from}\ \mathtt{A}\ \mathtt{to}\ \mathtt{B}; \mathtt{G} \rangle & = & \mathtt{a(S)}\ \mathtt{from}\ \mathtt{A}\ \mathtt{to}\ \mathtt{B}; \langle \mathtt{G} \rangle \\
\langle \mathtt{continue}\ \mathtt{t} \rangle & = & \mathtt{continue}\ \mathtt{t} \\
\langle \mathtt{choice}\ \mathtt{at}\ \mathtt{A}\ \{\mathtt{G_i}\}_{i \in \{1,..,n\}} \rangle & = & \mathtt{choice}\ \mathtt{at}\ \mathtt{A}\ \{\mathtt{flatten}(\langle \mathtt{G_i} \rangle)\}_{i \in \{1,..,n\}}
\end{array}$$

Intuitively, a protocol is translated to a normal form after first unfolding all recursions once and then flattening nested choice. For example, Fig. 2.8 shows a Scribble protocol and its translation to its normal form.

```
choice at A {
  rec Loop{
  choice at A
    {msg1 from A to B;
     continue Loop;}
  or {msg2 from A to B;}}
or {msg3 from A to B; }
```

```
choice at A {
  msg1 from A to B;
    rec Loop{
    choice at A
      {msg1 from A to B;
       continue Loop;}
    or {msg2 from A to B;}}
or {msg2 from A to B;}
or {msg3 from A to B; }
```

Figure 2.8.: A Scribble global protocol (left) and its corresponding normal form translation (left)

Traces We write $TR(\mathtt{G})$ for the set of traces obtained by reducing $\mathtt{G}$. We assume $\mathtt{G}$ is closed, i.e without free type variables. More precisely, $TR(\mathtt{G}) = \{\vec{\ell} \mid \exists \mathtt{G}', \mathtt{G} \xrightarrow{\vec{\ell}} \mathtt{G}'\}$. Similarly for local protocols, global and local types.

We denote trace equivalence by $\approx$. We write $\mathtt{G} \approx \mathtt{G}'$ if $TR(\mathtt{G}) = TR(\mathtt{G}')$. We also write $\mathtt{G} \lesssim \mathtt{G}'$ if $TR(\mathtt{G}) \subseteq TR(\mathtt{G}')$. We extend $\approx$ and $\lesssim$ to local protocols, as well as global and local types, and configuration of local protocols.

We use the following lemmas to prove that the translation of global protocols to SNF is semantics preserving:

Lemma 2.3.6 *Let* $\mathtt{G}$ *be a Scribble local protocol, then:*

- $\mathtt{G} \approx \mathtt{flatten}(\mathtt{G})$

- $\mathtt{G} \approx \mathtt{unfold}(\mathtt{G})$
- $\langle \mathtt{G'} \rangle[\mathtt{rec}\ \mathtt{t}\ \{\langle \mathtt{G'} \rangle\}/\mathtt{continue}\ \mathtt{t}] \approx \langle \mathtt{G'}[\mathtt{rec}\ \mathtt{t}\ \{\mathtt{G'}\}/\mathtt{continue}\ \mathtt{t}] \rangle$

Proposition 2.3.7 (SNF Translation) *Let $\mathtt{G}$ be a Scribble local protocol, then $\mathtt{G} \approx \langle \mathtt{G} \rangle$*

Proof. First we consider $\mathtt{G} \lesssim \langle \mathtt{G} \rangle$. The proof is mechanical and is done by induction on the transition rules applied for closed terms of $\mathtt{G}$.

1. (base case) If $\mathtt{G} = \mathtt{end}$ then both $\mathtt{G}$ and $\langle \mathtt{G} \rangle$ produce an empty set of traces and no rules can be applied.
2. (inductive case) if $\mathtt{G} \xrightarrow{\ell} \mathtt{G'}$ then $\langle \mathtt{G} \rangle \xrightarrow{\ell} \mathtt{G''}$ such that $\mathtt{G'} \approx \mathtt{G''}$.
   a) if $\mathtt{G} = \mathtt{a}(\mathtt{S})\ \mathtt{from}\ \mathtt{A}\ \mathtt{to}\ \mathtt{B}; \mathtt{G'}$
      $\mathtt{G}$ can do $\mathtt{AB}!\mathtt{msg}$ or ℓ by $\lfloor \textsc{send} \rfloor$ or $\lfloor \textsc{async1} \rfloor$ respectively.
      Then $\mathtt{G} \lesssim \langle \mathtt{G} \rangle$ follows by the induction hypothesis (IH) and by the definition of encoding
   b) $\mathtt{G} = \mathtt{rec}\ \mathtt{t}\ \{\mathtt{G'}\}$
      $\mathtt{G} \xrightarrow{\ell} \mathtt{G''}$
      By $\lfloor \textsc{rec} \rfloor$ $\mathtt{G'}[\mathtt{rec}\ \mathtt{t}\ \{\mathtt{G'}\}/\mathtt{continue}\ \mathtt{t}] \xrightarrow{\ell} \mathtt{G''}$
      By IH $\langle \mathtt{G'}[\mathtt{rec}\ \mathtt{t}\ \{\mathtt{G'}\}/\mathtt{continue}\ \mathtt{t}] \rangle \xrightarrow{\ell} \mathtt{G'''}$ s.t $\mathtt{G''} \approx \mathtt{G'''}$
      By Lemma 2.3.6
      $\langle \mathtt{G'} \rangle[\mathtt{rec}\ \mathtt{t}\ \{\langle \mathtt{G'} \rangle\}/\mathtt{continue}\ \mathtt{t}] \approx \langle \mathtt{G'}[\mathtt{rec}\ \mathtt{t}\ \{\mathtt{G'}\}/\mathtt{continue}\ \mathtt{t}] \rangle$
      Thus, $\langle \mathtt{G'} \rangle[\mathtt{rec}\ \mathtt{t}\ \{\langle \mathtt{G'} \rangle\}/\mathtt{continue}\ \mathtt{t}] \xrightarrow{\ell} \mathtt{G''''}$ s.t $\mathtt{G'''} \approx \mathtt{G''''}$
      By $\lfloor \textsc{rec} \rfloor$ $\mathtt{rec}\ \mathtt{t}\ \{\langle \mathtt{G'} \rangle\} \xrightarrow{\ell} \mathtt{G''''}$
      By Lemma 2.3.6 $\mathtt{rec}\ \mathtt{t}\ \{\langle \mathtt{G'} \rangle\} \approx \mathtt{unfold}(\mathtt{rec}\ \mathtt{t}\ \{\langle \mathtt{G'} \rangle\}) = \langle \mathtt{G} \rangle$
   c) $\mathtt{G} = \mathtt{choice}\ \mathtt{at}\ \mathtt{A}\ \{\mathtt{G_i}\}_{i \in \{1,..,n\}}$
      From $\lfloor \textsc{choice} \rfloor \mathtt{G} \xrightarrow{\ell} \mathtt{G'}$ with $\mathtt{G_i} \xrightarrow{\ell} \mathtt{G'}$
      From IH $\langle \mathtt{G_i} \rangle \xrightarrow{\ell} \mathtt{G''}$ s.t $\mathtt{G''} \approx \mathtt{G'}$ From $\mathtt{flatten}(\mathtt{G}) \approx \mathtt{G}$ it follows that $\mathtt{flatten}(\mathtt{G_i}) \xrightarrow{\ell} \mathtt{G'''}$ s.t $\mathtt{G'''} \approx \mathtt{G''}$
      From $\lfloor \textsc{choice} \rfloor$ it follows $\langle \mathtt{G} \rangle \xrightarrow{\ell} \mathtt{G'''}$

Now we consider $\langle \mathtt{G} \rangle \lesssim \mathtt{G}$. The proof is by induction on the definition of encoding of closed terms of $\mathtt{G}$.

1. (base case) If $\langle \mathtt{G} \rangle = \mathtt{end}$ then both $\mathtt{G}$ and $\langle \mathtt{G} \rangle$ produce an empty set of traces and no rules can be applied.
2. (inductive case) if $\langle \mathtt{G} \rangle \xrightarrow{\ell} \mathtt{G'}$ then $\mathtt{G} \xrightarrow{\ell} \mathtt{G''}$ such that $\mathtt{G'} \approx \mathtt{G''}$.
   a) $\langle \mathtt{a}(\mathtt{S})\ \mathtt{from}\ \mathtt{A}\ \mathtt{to}\ \mathtt{B}; \mathtt{G} \rangle = \mathtt{a}(\mathtt{S})\ \mathtt{from}\ \mathtt{A}\ \mathtt{to}\ \mathtt{B}; \langle \mathtt{G} \rangle$
      $\langle \mathtt{G} \rangle$ can do $\mathtt{AB}!\mathtt{msg}$ or ℓ by $\lfloor \textsc{send} \rfloor$ or $\lfloor \textsc{async1} \rfloor$ respectively.

Then $\mathtt{G} \lesssim \langle\mathtt{G}\rangle$ follows by the IH and by the definition of encoding

b) $\langle \mathtt{rec}\ \mathtt{t}\ \{\mathtt{G}\}\rangle = \mathtt{unfold}(\mathtt{rec}\ \mathtt{t}\ \{\langle\mathtt{G}\rangle\}) = \langle\mathtt{G}\rangle[\mathtt{rec}\ \mathtt{t}\ \{\langle\mathtt{G}\rangle\}/\mathtt{continue}\ \mathtt{t}]$
From IH: $\mathtt{G}[\mathtt{rec}\ \mathtt{t}\ \{\mathtt{G}\}/\mathtt{continue}\ \mathtt{t}] \approx \langle\mathtt{G}\rangle[\mathtt{rec}\ \mathtt{t}\ \{\langle\mathtt{G}\rangle\}/\mathtt{continue}\ \mathtt{t}]$
Thus, if $\langle\mathtt{G}\rangle[\mathtt{rec}\ \mathtt{t}\ \{\langle\mathtt{G}\rangle\}/\mathtt{continue}\ \mathtt{t}] \xrightarrow{\ell} \mathtt{G}'$
then $\mathtt{G}[\mathtt{rec}\ \mathtt{t}\ \{\mathtt{G}\}/\mathtt{continue}\ \mathtt{t}] \xrightarrow{\ell} \mathtt{G}''$ s.t $\mathtt{G}' \approx \mathtt{G}''$
From $\lfloor\textsc{rec}\rfloor$ rule: $\mathtt{rec}\ \mathtt{t}\ \{\mathtt{G}\} \xrightarrow{\ell} \mathtt{G}''$

c) $\langle\mathtt{G}\rangle = \langle \mathtt{choice\ at}\ \mathtt{A}\ \{\mathtt{G_i}\}_{i\in\{1,..,n\}}\rangle = \mathtt{choice\ at}\ \mathtt{A}\ \{\mathtt{flatten}(\langle\mathtt{G_i}\rangle)\}_{i\in\{1,..,n\}}$
From $\lfloor\textsc{choice}\rfloor$ $\langle\mathtt{G}\rangle \xrightarrow{\ell} \mathtt{G}'$ with $\mathtt{flatten}(\langle\mathtt{G_i}\rangle) \xrightarrow{\ell} \mathtt{G}'$
By Lemma 2.3.6 $\mathtt{flatten}(\langle\mathtt{G}\rangle) \approx \langle\mathtt{G}\rangle$ it follows that
$\langle\mathtt{G_i}\rangle \xrightarrow{\ell} \mathtt{G}''$ s.t $\mathtt{G}'' \approx \mathtt{G}'$
From IH it follows that $\mathtt{G} \xrightarrow{\ell} \mathtt{G}'''$ s.t $\mathtt{G}''' \approx \mathtt{G}'' \approx \mathtt{G}'$.

2.3.2. From Global Protocols to Global Types

Next we give the encoding from a Scribble Normal Form to MPST, which is straightforward.

Definition 2.3.8 (Encoding of Global Protocols to Global Types) *The encoding* $[\![\,]\!]$ *from SNF to global types is given below:*

$$\begin{aligned}
&[\![\ \mathtt{a}(\mathtt{S})\ \mathtt{from}\ \mathtt{A}\ \mathtt{to}\ \mathtt{B};\mathtt{G}]\!] = \mathtt{A} \to \mathtt{B} : \{\mathtt{a}\langle\mathtt{S}\rangle.[\![\mathtt{G}]\!]\}\\
&[\![\mathtt{choice\ at}\ \mathtt{A}\ \{\mathtt{G^b_j}\}_{j\in\{1,..,n\}}]\!] =\ \mathtt{A} \to \mathtt{B} : \{\mathtt{a}_j\langle\mathtt{S}_j\rangle.[\![\mathtt{G_j}]\!]\}_{j\in\{1,..,n\}}\\
&\qquad\qquad\qquad\qquad\qquad \textit{where}\ \mathtt{G^b_j} =\ \mathtt{a}_j(\mathtt{S}_j)\ \mathtt{from}\ \mathtt{A}\ \mathtt{to}\ \mathtt{B};\mathtt{G_j}\\
&[\![\mathtt{rec}\ \mathtt{t}\ \{\mathtt{G}\}]\!] = \mu\mathtt{t}.[\![\mathtt{G}]\!]\\
&[\![\mathtt{continue}\ \mathtt{t}]\!] = \mathtt{t}\\
&[\![\mathtt{end}]\!] = \mathtt{end}
\end{aligned}$$

For convenience, we recall the semantics of global types in Fig. 2.9. The semantics of global protocols and global types are similar except that the one for MPSTs from [DY13] have no rule $\lfloor\textsc{Choice}\rfloor$ as choice is handled directly in the rule for send/selection and branch/receive.

To match Scribble global protocols and MPST step by step we extend the definition of encoding to account for intermediate steps:
$[\![\mathtt{transit}:\mathtt{a}(\mathtt{S})\ \mathtt{from}\ \mathtt{A}\ \mathtt{to}\ \mathtt{B};\mathtt{G}'']\!] = \mathtt{A} \rightsquigarrow \mathtt{B} : \mathtt{a}\langle\mathtt{S}\rangle.[\![\mathtt{G}'']\!]$

Proposition 2.3.9 (Correspondence of Global Protocols and Global Types) *Let* $\mathtt{G}$ *be a Scribble global protocol, then* $\mathtt{G} \approx [\![\mathtt{G}]\!]$.

$$\frac{j \in I}{(\mathtt{A} \to \mathtt{B} : \{\mathtt{a}_i\langle \mathtt{S}_i\rangle.G_i\}_{i\in I}) \xrightarrow{\mathtt{AB!a}_j\langle \mathtt{S}_j\rangle} (\mathtt{A} \rightsquigarrow \mathtt{B} : \mathtt{a}_j\langle \mathtt{S}_j\rangle.G_j)} \quad \lfloor\text{SELECT}\rfloor$$

$$\frac{}{(\mathtt{A} \rightsquigarrow \mathtt{B} : \mathtt{a}\langle \mathtt{S}\rangle.G) \xrightarrow{\mathtt{AB?a}\langle \mathtt{S}\rangle} (G)} \quad \frac{(G[\mu\mathtt{t}.G/\mathtt{t}]) \xrightarrow{\ell} (G')}{(\mu\mathtt{t}.G) \xrightarrow{\ell} (G')} \quad \lfloor\text{BRANCH}\rfloor/\lfloor\text{REC}\rfloor$$

$$\frac{\forall k \in I \quad G_k \xrightarrow{\ell} G'_k \quad \mathtt{A},\mathtt{B} \notin subj(\ell) \quad \ell \neq t}{(\mathtt{A} \to \mathtt{B} : \{\mathtt{a}_i\langle \mathtt{S}_i\rangle.G_i\}_{i\in I}) \xrightarrow{\ell} (\mathtt{A} \to \mathtt{B} : \{\mathtt{a}_i\langle \mathtt{S}_i\rangle.G'_i\}_{i\in I})} \quad \lfloor\text{ASYNC1}\rfloor$$

$$\frac{G \xrightarrow{\ell} G' \quad \mathtt{B} \notin subj(\ell)}{(\mathtt{A} \rightsquigarrow \mathtt{B} : \mathtt{a}\langle \mathtt{S}\rangle.G) \xrightarrow{\ell} (\mathtt{A} \rightsquigarrow \mathtt{B} : \mathtt{a}\langle \mathtt{S}\rangle.G')} \quad \lfloor\text{ASYNC2}\rfloor$$

Figure 2.9.: Labelled transitions for global types in the framework of MPSTs (adapted from [DY13]).

Proof. First, we consider $\mathtt{G} \lesssim [\![\mathtt{G}]\!]$. The proof is done by induction (on the depth of the tree) on the transition rule applied.

1. (Base case) If $\mathtt{G} = \mathtt{end}$ then both $\mathtt{G}$ and $[\![\mathtt{G}]\!]$ produce an empty set of traces.
2. (Inductive case) if $\mathtt{G} \xrightarrow{\ell} \mathtt{G}'$ and we have to prove that $[\![\mathtt{G}]\!] \xrightarrow{\ell} [\![\mathtt{G}']\!]$.

- if $\mathtt{G} = \mathtt{a}(\mathtt{S})$ $\mathtt{from}$ $\mathtt{A}$ $\mathtt{to}$ $\mathtt{B};\mathtt{G}''$ then we either have a send action by $\lfloor\text{SEND}\rfloor$ or ℓ transition by $\lfloor\text{ASYNC1}\rfloor$
  - $\lfloor\text{SEND}\rfloor$ $\mathtt{G} \xrightarrow{\mathtt{AB!a}\langle \mathtt{S}\rangle} \mathtt{transit}:\mathtt{a}(\mathtt{S})$ $\mathtt{from}$ $\mathtt{A}$ $\mathtt{to}$ $\mathtt{B};\mathtt{G}''$
    By (1) $[\![\mathtt{G}]\!] = \mathtt{A} \to \mathtt{B} : \{\mathtt{a}\langle \mathtt{S}\rangle.[\![\mathtt{G}'']\!]\}$ and
    (2) $[\![\mathtt{transit}:\mathtt{a}(\mathtt{S})\ \mathtt{from}\ \mathtt{A}\ \mathtt{to}\ \mathtt{B};\mathtt{G}'']\!] = \mathtt{A} \rightsquigarrow \mathtt{B} : \mathtt{a}\langle \mathtt{S}\rangle.[\![\mathtt{G}'']\!]$ and
    (3) $\lfloor\text{SELECT}\rfloor_{MPST}$: $\mathtt{A} \to \mathtt{B} : \{\mathtt{a}\langle \mathtt{S}\rangle.[\![\mathtt{G}'']\!]\} \xrightarrow{\mathtt{AB!a}\langle \mathtt{S}\rangle} \mathtt{A} \rightsquigarrow \mathtt{B} : \mathtt{a}\langle \mathtt{S}\rangle.[\![\mathtt{G}'']\!]$
    we have $[\![\mathtt{G}]\!] \xrightarrow{\mathtt{AB!a}\langle \mathtt{S}\rangle} [\![\mathtt{G}']\!]$
  - $\lfloor\text{ASYNC1}\rfloor$ $\mathtt{a}(\mathtt{S})$ $\mathtt{from}$ $\mathtt{A}$ $\mathtt{to}$ $\mathtt{B};\mathtt{G}'' \xrightarrow{\ell} \mathtt{a}(\mathtt{S})$ $\mathtt{from}$ $\mathtt{A}$ $\mathtt{to}$ $\mathtt{B};\mathtt{G}''$
    By (1) $[\![\mathtt{G}]\!] = \mathtt{A} \to \mathtt{B} : \{\mathtt{a}\langle \mathtt{S}\rangle.[\![\mathtt{G}'']\!]\}$ and $[\![\mathtt{G}']\!] = \mathtt{A} \to \mathtt{B} : \mathtt{a}\langle \mathtt{S}\rangle.[\![\mathtt{G}''']\!]$ By (2) $[\![\mathtt{G}'']\!] \xrightarrow{\ell} [\![\mathtt{G}''']\!]$, which follows from the premise $\mathtt{G}'' \xrightarrow{\ell} \mathtt{G}'''$ of the $\lfloor\text{ASYNC1}\rfloor$ and by IH and
    (3) $\mathtt{B} \notin subj(\ell)$, which follows from the premise of $\lfloor\text{ASYNC1}\rfloor$:
    we can apply the $\lfloor\text{ASYNC1}\rfloor_{MPST}$ rule: $[\![\mathtt{G}]\!] \xrightarrow{\ell} [\![\mathtt{G}']\!]$
- if $\mathtt{G} = \mathtt{transit}:\mathtt{a}(\mathtt{S})$ $\mathtt{from}$ $\mathtt{A}$ $\mathtt{to}$ $\mathtt{B};\mathtt{G}''$
  We proceed as in the above case. We either have a receive action by the rule $\lfloor\text{RECV}\rfloor$ or ℓ transition by the rule $\lfloor\text{ASYNC2}\rfloor$.
  - $\lfloor\text{RECV}\rfloor$ $\mathtt{G} \xrightarrow{\mathtt{AB?a}\langle \mathtt{S}\rangle} \mathtt{G}'$ where $\mathtt{G}' = \mathtt{G}''$

By (1) $[\![\mathtt{G}]\!] = \mathtt{A} \rightsquigarrow \mathtt{B} : \mathtt{a}\langle\mathtt{S}\rangle.[\![\mathtt{G}'']\!]$ and $[\![\mathtt{G}']\!] = [\![\mathtt{G}'']\!]$ and
(3) $\lfloor\textsc{branch}\rfloor_{MPST}$: $\mathtt{A} \rightsquigarrow \mathtt{B} : \mathtt{a}\langle\mathtt{S}\rangle.[\![\mathtt{G}'']\!] \xrightarrow{\mathtt{AB?a}\langle\mathtt{S}\rangle} [\![\mathtt{G}'']\!]$
therefore $[\![\mathtt{G}]\!] \xrightarrow{\mathtt{AB?a}\langle\mathtt{S}\rangle} [\![\mathtt{G}']\!]$

– $\lfloor\textsc{async2}\rfloor$ $\mathtt{G} \xrightarrow{\ell} \mathtt{G}'$ where $\mathtt{G}' = \mathtt{transit}\,{:}\mathtt{a}(\mathtt{S})\ \mathtt{from}\ \mathtt{A}\ \mathtt{to}\ \mathtt{B};\mathtt{G}''$
By (1) $[\![\mathtt{G}]\!] = \mathtt{A} \rightsquigarrow \mathtt{B} : \mathtt{a}\langle\mathtt{S}\rangle.[\![\mathtt{G}'']\!]$ and $[\![\mathtt{G}']\!] = \mathtt{A} \to \mathtt{B} : \{\mathtt{a}\langle\mathtt{S}\rangle.[\![\mathtt{G}''']\!]\}$ By (2) $[\![\mathtt{G}'']\!] \xrightarrow{\ell} [\![\mathtt{G}''']\!]$, which follows from the premises $\mathtt{G}'' \xrightarrow{\ell} \mathtt{G}'''$ of the $\lfloor\textsc{async1}\rfloor$ and by the induction hypothesis and
(3) $\mathtt{A},\mathtt{B} \notin \mathit{subj}(\ell)$, which follows from the premise of $\lfloor\textsc{async2}\rfloor$:
we can apply the $\lfloor\textsc{async2}\rfloor_{MPST}$ rule: $[\![\mathtt{G}]\!] \xrightarrow{\ell} [\![\mathtt{G}']\!]$

- if $\mathtt{G} = \mathtt{choice\ at}\ \mathtt{A}\ \{\mathtt{G}^{\mathtt{b}}_{\mathtt{j}}\}_{j\in\{1,..,n\}})$
By $\lfloor\textsc{choice}\rfloor$ we have $\mathtt{G} \xrightarrow{\ell} \mathtt{G}'$ where by the rule premise we have for $\mathtt{G}'$: $\mathtt{a}_i(\mathtt{S}_i)\ \mathtt{from}\ \mathtt{A}\ \mathtt{to}\ \mathtt{B};\mathtt{G}'' \xrightarrow{\ell} \mathtt{G}'$ for $(i \in I)$ which brings us back to the first case.
- if $\mathtt{G} = \mathtt{rec}\ \mathtt{t}\ \{\mathtt{G}''\}$ the thesis directly follows by induction since
(1) by $\lfloor\textsc{rec}\rfloor \mathtt{G} \xrightarrow{\ell} \mathtt{G}'$ where $\mathtt{G}[\mathtt{rec}\ \mathtt{t}\ \{\mathtt{G}\}/\mathtt{continue}\ \mathtt{t}] \xrightarrow{\ell} \mathtt{G}'$
(2) $[\![\mathtt{G}]\!] = \mu\mathtt{t}[\![\mathtt{G}'']\!]$
By $\lfloor\textsc{rec}\rfloor$ $[\![\mathtt{G}'']\!][\mu\mathtt{t}.[\![\mathtt{G}'']\!]/\mathtt{t}]) \xrightarrow{\ell} [\![\mathtt{G}']\!]$
(3) From IH, $\mathtt{G}' \lesssim [\![\mathtt{G}']\!]$ and therefore $\mathtt{G} \lesssim [\![\mathtt{G}]\!]$

Now we consider $[\![\mathtt{G}]\!] \lesssim \mathtt{G}$.
The proof is done by induction on transition rules applied to the encoding of $\mathtt{G}$.

1. $[\![\mathtt{G}]\!] = \mathsf{end}$ then both $[\![\mathtt{G}]\!]$ and $\mathtt{G}$ then no rules can be applied.
2. if $[\![\mathtt{G}]\!] = \mathtt{A} \to \mathtt{B} : \{\mathtt{a}\langle\mathtt{S}\rangle.[\![\mathtt{G}]\!]\}$, then we either have a send action by $\lfloor\textsc{select}\rfloor_{MPST}$ or ℓ transition by $\lfloor\textsc{async1}\rfloor$.
   - $\lfloor\textsc{select}\rfloor_{MPST}$ $[\![\mathtt{G}]\!] \xrightarrow{\mathtt{AB!a}\langle\mathtt{S}\rangle} [\![\mathtt{G}']\!]$
   By $\mathtt{G} = \mathtt{a}(\mathtt{S})\ \mathtt{from}\ \mathtt{A}\ \mathtt{to}\ \mathtt{B};\mathtt{G}''$ and $\mathtt{G}' = \mathtt{transit}\,{:}\mathtt{a}(\mathtt{S})\ \mathtt{from}\ \mathtt{A}\ \mathtt{to}\ \mathtt{B};\mathtt{G}''$ and $\lfloor\textsc{send}\rfloor$ it follows that $\mathtt{G} \xrightarrow{\mathtt{AB!a}\langle\mathtt{S}\rangle} \mathtt{G}'$
   - $\lfloor\textsc{async1}\rfloor_{MPST}$ $[\![\mathtt{G}]\!] \xrightarrow{\ell} [\![\mathtt{G}']\!]$ where
   $[\![\mathtt{G}]\!] = \mathtt{A} \to \mathtt{B} : \{\mathtt{a}\langle\mathtt{S}\rangle.[\![\mathtt{G}'']\!]\}$ and $[\![\mathtt{G}']\!] = \mathtt{A} \rightsquigarrow \mathtt{B} : \mathtt{a}\langle\mathtt{S}\rangle.[\![\mathtt{G}'']\!]$
   By (1) the rule premise $[\![\mathtt{G}'']\!] \xrightarrow{\ell} [\![\mathtt{G}''']\!]$ and by (2) IH it follows that $\mathtt{G}'' \xrightarrow{\ell} \mathtt{G}'''$. Given also that $\mathtt{A},\mathtt{B} \notin \mathit{subj}(\ell)$, we can apply $\lfloor\textsc{async1}\rfloor$. Thus, $\mathtt{G} \xrightarrow{\ell} \mathtt{G}'$
3. if $[\![\mathtt{G}]\!] = [\![\mathtt{choice\ at}\ \mathtt{A}\ \mathtt{a}_j(\mathtt{S}_j)\ \mathtt{from}\ \mathtt{A}\ \mathtt{to}\ \mathtt{B};\mathtt{G}_\mathtt{j}]\!] = \mathtt{A} \to \mathtt{B} : \{\mathtt{a}_j\langle\mathtt{S}_j\rangle.[\![\mathtt{G}_\mathtt{j}]\!]\}_{j\in\{1,..,n\}}$
Then by $\lfloor\textsc{choice}\rfloor$ we have that $[\![\mathtt{G}]\!] \xrightarrow{\ell} [\![\mathtt{G}']\!]$ when $[\![\mathtt{a}_i(\mathtt{S}_i)\ \mathtt{from}\ \mathtt{A}\ \mathtt{to}\ \mathtt{B};\mathtt{G}_\mathtt{i}]\!] \xrightarrow{\ell} [\![\mathtt{G}']\!]$ for $i \in I$.
Thus, we have to prove that

if $[\![\mathtt{a}_i(\mathtt{S}_i)\ \mathtt{from}\ \mathtt{A}\ \mathtt{to}\ \mathtt{B};\mathtt{G_i}]\!] \xrightarrow{\ell} [\![\mathtt{G}']\!]$ then $\mathtt{a}_i(\mathtt{S}_i)\ \mathtt{from}\ \mathtt{A}\ \mathtt{to}\ \mathtt{B};\mathtt{G_i} \xrightarrow{\ell} [\![\mathtt{G}']\!]$, which follows from a).

4. if $[\![\mathtt{G}]\!] = \mathtt{A} \rightsquigarrow \mathtt{B} : \mathtt{a}\langle\mathtt{S}\rangle.[\![\mathtt{G}'']\!]$ $[\![\mathtt{G}]\!]$ can do a receive action by $\lfloor\text{BRANCH}\rfloor_{MPST}$ or ℓ transition by $\lfloor\text{ASYNC2}\rfloor$.
   - $\lfloor\text{BRANCH}\rfloor_{MPST}$ $[\![\mathtt{G}]\!] \xrightarrow{\mathtt{AB!a\langle S\rangle}} [\![\mathtt{G}']\!]$
     By $\mathtt{G} = \mathtt{transit}:\mathtt{a}(\mathtt{S})\ \mathtt{from}\ \mathtt{A}\ \mathtt{to}\ \mathtt{B};\mathtt{G}''$ and $\mathtt{G}' = \mathtt{transit}:\mathtt{a}(\mathtt{S})\ \mathtt{from}\ \mathtt{A}\ \mathtt{to}\ \mathtt{B};\mathtt{G}''$ and $\lfloor\text{RECV}\rfloor$ it follows that $\mathtt{G} \xrightarrow{\mathtt{AB?a\langle S\rangle}} \mathtt{G}'$
   - $\lfloor\text{ASYNC2}\rfloor_{MPST}$ $[\![\mathtt{G}]\!] \xrightarrow{\ell} [\![\mathtt{G}']\!]$ where
     $[\![\mathtt{G}]\!] = \mathtt{A} \rightsquigarrow \mathtt{B} : \mathtt{a}\langle\mathtt{S}\rangle.[\![\mathtt{G}'']\!]$ and $[\![\mathtt{G}']\!] = \mathtt{A} \rightarrow \mathtt{B} : \mathtt{a}\langle\mathtt{S}\rangle.[\![\mathtt{G}'']\!]$
     By (1) the rule premise $[\![\mathtt{G}'']\!] \xrightarrow{\ell} [\![\mathtt{G}''']\!]$ and by (2) IH it follows that $\mathtt{G}'' \xrightarrow{\ell} \mathtt{G}'''$. Given also that $\mathtt{A},\mathtt{B} \notin subj(\ell)$, we can apply $\lfloor\text{ASYNC2}\rfloor$. Thus, $\mathtt{G} \xrightarrow{\ell} \mathtt{G}'$

5. if $[\![\mathtt{G}]\!] = \mu\mathtt{t}.[\![\mathtt{G}'']\!]$ the thesis directly follows by induction.

2.3.3. From Local Protocols to Local Types

The syntactic differences between Scribble local protocols and (MPST) local type reflect the difference between Scribble global protocols and MPST global types. The syntax of (MPST) local types from [DY13] is given below:

Definition 2.3.10 (Local Types)

$$T \quad ::= \quad \mathtt{B}!\{\mathtt{a}_i : \langle\mathtt{S}_i\rangle.T_i'\}_{i\in I} \mid \mathtt{B}?\{\mathtt{a}_i : \langle\mathtt{S}_i\rangle.T_i'\}_{i\in I} \mid \mu\mathtt{t}.T \mid \mathtt{t} \mid \mathtt{end}$$

Similarly to the encoding of global protocol to global types, we define the encoding of local protocol to local types on the normal form of a Scribble local protocol. The grammar for normal form for local protocols is given below:

Definition 2.3.11 (Local Scribble Normal Form (LSNF))

$$\begin{array}{rcl} \mathtt{T} & ::= & \mathtt{choice\ at}\ \mathtt{A}\{\mathtt{N_i}\}_{i\in I} \mid \mathtt{N} \mid \mathtt{rec}\ \mathtt{t}\ \{\mathtt{N}\} \\ \mathtt{N} & ::= & \mathtt{a}(\mathtt{S})\ \mathtt{from}\ \mathtt{B};\mathtt{T} \mid \mathtt{a}(\mathtt{S})\ \mathtt{to}\ \mathtt{B};\mathtt{T} \mid \mathtt{continue}\ \mathtt{t} \mid \mathtt{end} \end{array}$$

Next we give the definition of projection. The definition is adapted from the definition of projection of global types to local types, given in [DY13]. We denote by $\mathscr{P}(\mathtt{G})$ the set of roles in a protocol $\mathtt{G}$.

Definition 2.3.12 (Projection) *The projection of* $\mathtt{G}$ *onto* $\mathtt{A} \in \mathscr{P}(\mathtt{G})$*, written* $\mathtt{G} \downarrow_{\mathtt{A}}$*, is defined by induction on* $\mathtt{G}$ *as follows:*

$$(\mathtt{a}(\mathtt{S})\ \mathtt{from}\ \mathtt{B}\ \mathtt{to}\ \mathtt{C};\mathtt{G}') \downarrow_{\mathtt{A}} = \begin{cases} \mathtt{a}(\mathtt{S})\ \mathtt{from}\ \mathtt{B};(\mathtt{G}' \downarrow_{\mathtt{A}}) & \textit{if } \mathtt{A} = \mathtt{C} \\ \mathtt{a}(\mathtt{S})\ \mathtt{to}\ \mathtt{C};(\mathtt{G}' \downarrow_{\mathtt{A}}) & \textit{if } \mathtt{A} = \mathtt{B} \\ \mathtt{G}' \downarrow_{\mathtt{A}} & \textit{if } \mathtt{A} \neq \mathtt{B},\mathtt{C} \end{cases}$$

$$(\mathtt{choice\ at}\ \mathtt{B}\ \{\mathtt{a_i}(\mathtt{S_i})\ \mathtt{from}\ \mathtt{B}\ \mathtt{to}\ \mathtt{C};\mathtt{G_i}\}_{i\in I}) \downarrow_{\mathtt{A}} =$$
$$\begin{cases} \mathtt{choice\ at}\ \mathtt{B}\ \{\mathtt{a_i}(\mathtt{S_i})\ \mathtt{from}\ \mathtt{B};(\mathtt{G_i} \downarrow_{\mathtt{A}})\}_{i\in I} & \textit{if } \mathtt{A} = \mathtt{C} \\ \mathtt{choice\ at}\ \mathtt{B}\ \{\mathtt{a_i}(\mathtt{S_i})\ \mathtt{to}\ \mathtt{C};(\mathtt{G_i} \downarrow_{\mathtt{A}})\}_{i\in I} & \textit{if } \mathtt{A} = \mathtt{B} \\ \mathtt{choice\ at}\ \mathtt{D}\ (\sqcup\{(\mathtt{G_i} \downarrow_{\mathtt{A}})\}_{i\in I}) & \textit{if } \mathtt{A} \neq \mathtt{B},\mathtt{C};\ \mathtt{G_i} \downarrow_{\mathtt{A}} = \mathtt{a_i}(\mathtt{S_i})\ \mathtt{from}\ \mathtt{D};\mathtt{G}'_{\mathtt{i}} \downarrow_{\mathtt{A}}, \forall i \in I \end{cases}$$

$$(\mathtt{rec}\ \mathtt{t}\ \{\mathtt{G}'\}) \downarrow_{\mathtt{A}} = \begin{cases} \mathtt{rec}\ \mathtt{t}\ \{(\mathtt{G}' \downarrow_{\mathtt{A}})\} & \mathtt{G}' \neq \mathtt{continue}\ \mathtt{t} \\ \mathtt{end} & \textit{otherwise} \end{cases}$$

$$(\mathtt{continue}\ \mathtt{t}) \downarrow_{\mathtt{A}} = \mathtt{continue}\ \mathtt{t}$$
$$(\mathtt{end}) \downarrow_{\mathtt{A}} = \mathtt{end}$$

If no side condition applies then $\mathtt{G}$ is *not projectable* on $\mathtt{A}$ and the global protocol $\mathtt{G}$ is not well-formed. The case for $\mathtt{choice}$ uses the merge operator $\sqcup$ to ensure that (1) the locally projected behaviour is independent of the chosen branch (i.e $\mathtt{G_i} = \mathtt{G_j}$, for all $i, j \in I$), or (2) the chosen branch is identifiable by $\mathtt{A}$ via a unique label. The merge operator $\sqcup$ [DY13] is defined as a partial commutative operator over two types such that $\mathtt{T} \sqcup \mathtt{T} = \mathtt{T}$ for all types and that:

$$\begin{aligned}&\{\mathtt{a_i}(\mathtt{S_i})\ \mathtt{from}\ \mathtt{B};\mathtt{T_i}\}_{i\in I} \sqcup \{\mathtt{a'_j}(\mathtt{S'_j})\ \mathtt{from}\ \mathtt{B};\mathtt{T'_j}\}_{j\in J} = \\ &\{\mathtt{a_k}(\mathtt{S_k})\ \mathtt{from}\ \mathtt{B};\mathtt{T_k}\}_{k\in I\setminus J} \cup \{\mathtt{a'_j}(\mathtt{S'_j})\ \mathtt{from}\ \mathtt{B};\mathtt{T'_j}\}_{j\in J\setminus I} \\ &\cup\{\mathtt{a_k}(\mathtt{S_k})\ \mathtt{from}\ \mathtt{B};\mathtt{T_k} \sqcup \mathtt{T'_k}\}_{k\in I\cap J}\ \textit{where for each } k \in I \cap J, \mathtt{a_k} = \mathtt{a'_k}, \mathtt{S}_k = \mathtt{S}'_k\end{aligned}$$

and homomorphic for other types (i.e $\mathscr{E}[\mathtt{T_k}] \sqcup \mathscr{E}[\mathtt{T'_k}] = \mathscr{E}[\mathtt{T_k} \sqcup \mathtt{T'_k}]$, where $\mathscr{E}$ is a context for local protocols.). We say that $\mathtt{G}$ is *well-formed* if for all $\mathtt{A} \in \mathscr{P}(\mathtt{G})$, $\mathtt{G} \downarrow_{\mathtt{A}}$ is defined.

Note that a normal form is preserved during projection. More precisely, a Sribble global protocol in a normal form is projected to a Scribble local protocol in a normal form.

Next we give the encoding between Scribble local protocols and MPST local types. Hereafter we write Scribble local protocol when referring to LSNF protocols.

Definition 2.3.13 (Encoding of Local Protocols to Local Types) *The encoding* $(\!|\,|\!)$ *from (Scribble) local protocols to MPST local types is given below:*

$$
\begin{array}{l}
(\!|\mathtt{a(S)}\ \mathtt{to}\ \mathtt{B;T}|\!) = \mathtt{B!\{a : \langle S\rangle.(\!|T|\!)\}}\\
(\!|\mathtt{a(S)}\ \mathtt{from}\ \mathtt{B;T}|\!) = \mathtt{B?\{a : \langle S\rangle.(\!|T|\!)\}}\\
(\!|\mathtt{choice\ at\ A}\ \{\mathtt{T'_i}\}_{i\in I}|\!) = \begin{cases} \mathtt{B!}\{\mathtt{a}_i : \langle \mathtt{S}_i\rangle.(\!|\mathtt{T'_i}|\!)\}_{i\in I}\ \mathit{if}\ \mathtt{T'_i} = \mathtt{a_i(S_i)}\ \mathtt{to}\ \mathtt{B;T_i}\\ \mathtt{A?}\{\mathtt{a}_i : \langle \mathtt{S}_i\rangle.(\!|\mathtt{T'_i}|\!)\}_{i\in I}\ \mathit{if}\ \mathtt{T'_i} = \mathtt{a_i(S_i)}\ \mathtt{from}\ \mathtt{A;T_i}\end{cases}\\
(\!|\mathtt{rec\ t}\ \{\mathtt{T}\}|\!) = \mu\mathtt{t}.(\!|\mathtt{T}|\!)\\
(\!|\mathtt{continue\ t}|\!) = \mathtt{t}\\
(\!|\mathsf{end}|\!) = \mathsf{end}
\end{array}
$$

Proposition 2.3.14 (Correspondence of Local Protocols and Local Types) *Let* $\mathtt{T}$ *be a Scribble local protocol, then* $\mathtt{T} \approx (\!|\mathtt{T}|\!)$.

Proof. Below, for convenience, we recall the LTS for local session types (adapted from [DY13]):

$$
\begin{array}{lll}
\mathtt{B!}\{\mathtt{a}_i : \langle \mathtt{S}_i\rangle.T_i\}_{i\in I}) \xrightarrow{\mathtt{AB!a\langle S\rangle}} T_j\ \ (j\in I) & & \lfloor \text{LSEL} \rfloor\\
\mathtt{B?}\{\mathtt{a}_i : \langle \mathtt{S}_i\rangle.T_i\}_{i\in I}) \xrightarrow{\mathtt{AB?a\langle S\rangle}} T_j\ \ (j\in I) & & \lfloor \text{LBRA} \rfloor\\
T[\mu\mathtt{t}.T/\mathtt{t}] \xrightarrow{\ell} T'\ \ \mathit{imply}\ \ \mu\mathtt{t}.T \xrightarrow{\ell} T' & & \lfloor \text{LREC} \rfloor
\end{array}
$$

First, we consider $\mathtt{T} \lesssim [\![\mathtt{T}]\!]$.

The proof is done by induction (on the depth of the tree) on the transition rule applied.

1. (Base case) If $\mathtt{T} = \mathsf{end}$ then both $\mathtt{T}$ and $(\!|\mathtt{T}|\!)$ produce an empty set of traces.
2. (Inductive case) $\mathtt{T} \xrightarrow{\ell} \mathtt{T'}$ and we have to prove that $(\!|\mathtt{T}|\!) \xrightarrow{\ell} (\!|\mathtt{T'}|\!)$. We proceed by case analysis on the structure of $\mathtt{T}$
   a) if $\mathtt{T} = \mathtt{a(S)}\ \mathtt{to}\ \mathtt{B;T''} \xrightarrow{\mathtt{AB!a\langle S\rangle}} \mathtt{T''}$ by $\lfloor \text{SEND} \rfloor$
      $(\!|\mathtt{T}|\!) = \mathtt{B!\{a : \langle S\rangle.(\!|T''|\!)\}} \xrightarrow{\mathtt{AB!a\langle S\rangle}} (\!|\mathtt{T''}|\!)$ by $\lfloor \text{LSEL} \rfloor$
   b) if $\mathtt{T} = \mathtt{a(S)}\ \mathtt{from}\ \mathtt{B;T''} \xrightarrow{\mathtt{AB?a\langle S\rangle}} \mathtt{T''}$ by $\lfloor \text{RECV} \rfloor$
      $(\!|\mathtt{T}|\!) = \mathtt{B?\{a : \langle S\rangle.(\!|T''|\!)\}} \xrightarrow{\mathtt{AB?a\langle S\rangle}} (\!|\mathtt{T''}|\!)$ by $\lfloor \text{LBRA} \rfloor$
   c) if $\mathtt{T} = \mathtt{choice\ at\ A}\ \{\mathtt{T_i}\}_{i\in I}) \xrightarrow{\ell} \mathtt{T'}$

Depending on the structure of $\mathtt{T_i}$, this case folds back to previous cases a) and b).

if $\mathtt{T_i} = \mathtt{a_i(S_i)}$ $\mathtt{from}$ $\mathtt{B;T''} \xrightarrow{\mathtt{AB!a\langle S\rangle}} \mathtt{T''} = \mathtt{T'}$ then $(\!|\mathtt{T_i}|\!) \xrightarrow{\mathtt{AB!a\langle S\rangle}} (\!|\mathtt{T'}|\!)$ by $\lfloor$LSEL$\rfloor$

if $\mathtt{T_i} = \mathtt{B?\{a_i : \langle S_\mathit{i}\rangle.(\!|T''|\!)\}} \xrightarrow{\mathtt{AB?a\langle S\rangle}} = \mathtt{T''} = \mathtt{T'}$ then $(\!|\mathtt{T_i}|\!) \xrightarrow{\mathtt{AB?a\langle S\rangle}} (\!|\mathtt{T'}|\!)$ by $\lfloor$LBRA$\rfloor$

d) if $\mathtt{T} = \mu\mathtt{t.T''}$ the thesis directly follows by induction.

Now we consider $(\!|\mathtt{T}|\!) \lesssim \mathtt{T}$.

The proof is done by induction on transition rules applied to the encoding.

1. (Base case) If $(\!|\mathtt{T}|\!) = \mathsf{end}$ then both $(\!|\mathtt{T}|\!)$ and $\mathtt{T}$ produce an empty set of traces.
2. (Inductive case) $(\!|\mathtt{T}|\!) \xrightarrow{\ell} (\!|\mathtt{T'}|\!)$ and we have to prove that $\mathtt{T} \xrightarrow{\ell} \mathtt{T'}$. We proceed by case analysis on the structure of $(\!|\mathtt{T}|\!)$

- if $(\!|\mathtt{T}|\!) = \mathtt{B!\{a : \langle S\rangle.(\!|T''|\!)\}}$
  $\mathtt{B!\{a : \langle S\rangle.(\!|T''|\!)\}} \xrightarrow{\mathtt{AB!a\langle S\rangle}} (\!|\mathtt{T''}|\!)$ by $\lfloor$LSEL$\rfloor$
  $\mathtt{T} = \mathtt{a(S)}$ $\mathtt{to}$ $\mathtt{B;T''} \xrightarrow{\mathtt{AB!a\langle S\rangle}} (\!|\mathtt{T''}|\!)$ by $\lfloor$SEND$\rfloor$
- if $(\!|\mathtt{T}|\!) = \mathtt{B?\{a : \langle S\rangle.(\!|T''|\!)\}}$
  $\mathtt{B?\{a : \langle S\rangle.(\!|T''|\!)\}} \xrightarrow{\mathtt{AB?a\langle S\rangle}} (\!|\mathtt{T''}|\!)$ by $\lfloor$LBRA$\rfloor$
  $\mathtt{T} = \mathtt{a(S)}$ $\mathtt{from}$ $\mathtt{B;T''} \xrightarrow{\mathtt{AB?a\langle S\rangle}} \mathtt{T''}$ by $\lfloor$RECV$\rfloor$
- if $(\!|\mathtt{T}|\!) = \mathtt{B?\{a_\mathit{i} : \langle S_\mathit{i}\rangle.(\!|T_i|\!)\}}_{i\in I}$
  $\mathtt{B?\{a_\mathit{i} : \langle S_\mathit{i}\rangle.(\!|T_i|\!)\}}_{i\in I} \xrightarrow{\mathtt{AB?a\langle S\rangle}} (\!|\mathtt{T_j}|\!) (j \in I)$
  By $\lfloor$RECV$\rfloor$ and the structure of $\mathtt{T_i}$ we have that $\mathtt{a_i(T_i)}$ $\mathtt{to}$ $\mathtt{B;T_i} \xrightarrow{\mathtt{AB!a\langle S\rangle}} \mathtt{T_i}$ and therefore we can apply $\lfloor$CHOICE$\rfloor$
  Thus, $\mathtt{T} \xrightarrow{\mathtt{AB!a\langle S\rangle}} \mathtt{T_j}$
- if $(\!|\mathtt{T}|\!) = \mathtt{A!\{a_i : \langle S_\mathit{i}\rangle.(\!|T|\!)\}}_{i\in I}$ the case is analogical to the previous one.
- if $(\!|\mathtt{T}|\!) = \mu\mathtt{t.T''}$ the thesis directly follows by induction.

Proposition 2.3.15 (Correspondence of Configurations) *Let* $(\mathtt{T_1},\ldots,\mathtt{T_n},w)$ *be a configuration of Scribble local protocols, then* $(\mathtt{T'_1},\ldots,\mathtt{T'_n},w) \approx ((\!|\mathtt{T_1}'|\!),\ldots,(\!|\mathtt{T'_n}|\!),w')$.

Proof. The proof is by induction on the number of transition steps.
Inductive hypothesis: $(\mathtt{T_1},\ldots,\mathtt{T_n},w) \approx ((\!|\mathtt{T_1}|\!),\ldots,(\!|\mathtt{T_n}|\!),w)$
Now we want to prove that if $(\mathtt{T_1},\ldots,\mathtt{T_n},w) \xrightarrow{\ell} (\mathtt{T'_1},\ldots,\mathtt{T'_n},w')$ then $((\!|\mathtt{T_1}|\!),\ldots,(\!|\mathtt{T_n}|\!),w) \xrightarrow{\ell} ((\!|\mathtt{T'_1}|\!),\ldots,(\!|\mathtt{T'_n}|\!),w')$

We do a case analysis on the transition label ℓ:

(1) if $\ell = \mathtt{AB!a\langle S\rangle}$

By $\mathtt{T_B} \xrightarrow{\mathtt{AB!a\langle S\rangle}} \mathtt{T_B}$ and Lemma 3.3.5 it follows: $(\!|\mathtt{T_B}|\!) \xrightarrow{\mathtt{AB!a\langle S\rangle}} (\!|\mathtt{T_B}|\!)$

By definition of configuration of local protocols:

$w'_{\mathtt{AB}} = w_{\mathtt{AB}} \cdot \mathtt{a}(\mathtt{T}) \wedge (w_{ij} = w'_{ij})_{\text{for } ij \neq \mathtt{AB}}$.

(2) if $\ell = \mathtt{AB?a\langle S\rangle}$

By $\mathtt{T_B} \xrightarrow{\mathtt{AB?a\langle S\rangle}} \mathtt{T'_B}$ and Lemma 3.3.5 it follows: $(\!|\mathtt{T_B}|\!) \xrightarrow{\mathtt{AB?a\langle S\rangle}} (\!|\mathtt{T'_B}|\!)$

By definition of configuration of local protocols:

$w'_{\mathtt{AB}} = w_{\mathtt{AB}} \cdot \mathtt{a}(\mathtt{S}) \wedge (\Rightarrow w_{ij} = w'_{ij})_{ij \neq \mathtt{AB}}$

In (1) and (2) we have by definition that $\mathtt{T_i} = \mathtt{T'_i}($ for $\neq \mathtt{AB})$, which by the inductive hypothesis implies that $(\!|\mathtt{T_i}|\!) = (\!|\mathtt{T'_i}|\!)$

Then by the definition of configuration of local protocols (from (1) and (2)) it follows that $((\!|\mathtt{T_1}|\!), \ldots, , \ldots, (\!|\mathtt{T_n}|\!), w) \xrightarrow{\ell} ((\!|\mathtt{T'_1}|\!), \ldots, (\!|\mathtt{T'_n}|\!), w')$

2.3.4. Correspondence of Global and Local Protocols

Theorem 2.3.16 gives the correspondence between the traces produced by a global protocol $\mathtt{G}$ and those produced by the configuration that consists of the composition of the projections of $\mathtt{G}$ onto $\mathscr{P}(\mathtt{G})$.

Theorem 2.3.16 (Soundness of projection) *Let* $\mathtt{G}$ *be a Scribble global protocol and* $\{\mathtt{T_1}, \ldots, \mathtt{T_n}\} = \{\mathtt{G} \downarrow_{\mathtt{A}}\}_{\mathtt{A} \in \mathscr{P}(\mathtt{G})}$ *be the set of its projections, then*

$$\mathtt{G} \approx (\mathtt{T_1}, \ldots, \mathtt{T_n}, \vec{\varepsilon})$$

Theorem 2.3.16 directly follows by: (i) the correspondence between (Scribble) global protocols and MPSTs global types given in Section 2.3; (ii) trace equivalence between global types and configuration of projected global types (Theorem 3.1 in [DY13]); (iii) the correspondence between configurations of MPSTs local types and configurations of Scribble local protocols given in 2.3.

2.4. From Scribble to CFSMs

This section gives the translation of local protocols to CFSMs [BZ83]. First, we start from some preliminary notations. ε is the empty word. A is a finite alphabet and A^* is the set of all finite words over A. $|x|$ is the length of a word x and $x.y$ or xy the concatenation of two words x and y. Let $\mathscr{P}$ be a set of participants fixed throughout the section: $\mathscr{P} = \{\mathtt{A}, \mathtt{B}, \mathtt{C}, \ldots \mathtt{p}, \mathtt{q}, \ldots\}$.

Definition 2.4.1 (CFSM) *A communicating finite state machine is a finite transition system given by a 5-tuple $M = (Q, C, q_0, A, \delta)$ where (1) Q is a finite set of states; (2) $C = \{\mathtt{AB} \in \mathscr{P}^2 | \mathtt{A} \neq \mathtt{B}\}$ is a set of channels; (3) $q_0 \in Q$ is an initial state; (4) A is a finite alphabet of message labels, and (5) $\delta = Q \times (C \times \{!, ?\} \times A) \times Q$ is a finite set of transitions.*

A state $q \in Q$, which does not have any outgoing transitions, is called a final state. Note that in the above definition δ can be the empty relation when all states in Q are final states.

Definition 2.4.2 (CS) *A (communicating) system S is a tuple $S = (M_p)_{p\in\mathscr{P}}$ of CFSMs such that $M_p = (Q_p, C, q_{0_p}, A, \delta_p)$*

For $M_p = (Q_p, C, q_{0_p}, A, \delta_p)$ we define a configuration $S = (M_p)_{p\in\mathscr{P}}$ to be a tuple $s = (\vec{q}, \vec{w})$ where $\vec{q} = (q_p)_{p\in\mathscr{P}}$ and where $w = (w_{\mathtt{pq}})_{p\neq q\in\mathscr{P}}$ with $w_{pq} \in A^*$.

A *path* in M is a finite sequence of $q_0, \ldots, q_n (n \geq 0)$ such that $(q_i, \ell, q_{i+1}) \in \delta (0 \leq i \leq n-1)$ and we write $q \xrightarrow{\ell} q'$ if $(q, \ell, q') \in \delta$.

To define the translation we introduce

$$\mathtt{rec}\ \vec{\mathtt{t}}\ \{\mathtt{T}\} = \begin{cases} \mathtt{T} & \text{if } |\vec{\mathtt{t}}| = 0 \\ \mathtt{rec}\ \mathtt{t}_0\{\ldots\mathtt{rec}\ \mathtt{t_n}\ \{\mathtt{T}'\}\ldots\} & \text{if } \vec{\mathtt{t}} = (\mathtt{t}_0, \ldots, \mathtt{t}_n), \mathtt{T}' \neq \mathtt{rec}\ \mathtt{t}\ \{\mathtt{T}''\} \end{cases}$$

$$\mathtt{body}(\mathtt{T}) = \begin{cases} \mathtt{T}' & \mathtt{if}\ \mathtt{T} = \mathtt{rec}\ \vec{\mathtt{t}}\ \{\mathtt{T}'\} \\ \mathtt{T} & \mathtt{otherwise} \end{cases}$$

We remind that recursive variables are guarded in the standard way, i.e. they only occur under a prefix and therefore $\mathtt{body}(\mathtt{T})$ cannot be $\mathtt{continue}\ \mathtt{t}$.

Next we show how to algorithmically go from local Scribble protocols in a normal form to CFSMs. Note that, for convenience, we do not separate a label from a payload and we write $\mathtt{msg}$ instead of $\mathtt{a(S)}$. Without loss of generality we assume all nested recursive types are written in the form $\mathtt{rec}\ \vec{\mathtt{t}}\ \{\mathtt{T}\}$.

Definition 2.4.3 (translation from local types to CFSMs.) *We write $\mathtt{T}' \in \mathtt{T}$ if $\mathtt{T}'$ occurs in $\mathtt{T}$. Let $\mathtt{T}$ be the normal form of the local type of participant $\mathtt{A}$ projected from $\mathtt{G}$. The automaton corresponding to $\mathtt{T}$ is $\mathscr{A}(\mathtt{T}) = (Q, C, q_0, A, \delta)$ where: (1) $Q = \{\mathtt{T}' | \mathtt{T}' \in \mathtt{T}, \mathtt{T}' \neq \mathtt{continue}\ \mathtt{t}\} \setminus (\{\mathtt{T}' | \mathtt{rec}\ \vec{\mathtt{t}}\ \{\mathtt{T}'\} \in \mathtt{T}\} \cup \{\mathtt{T_i} | \mathtt{choice\ at}\ \mathtt{A}\ \{\mathtt{T_i}\}_{i\in I} \in \mathtt{T}\})$; (2) $q_0 = \mathtt{T}$ (3) $C = \{\mathtt{AB}\ |\ \mathtt{A}, \mathtt{B} \in \mathtt{G}\}$; (4) $A = \{\mathtt{msg} \in \mathtt{G}\}$ is the set of labels $\mathtt{msg}$ in $\mathtt{G}$; and (5) δ is defined below:*

1. *if* $\mathtt{body}(\mathtt{T}) = \mathit{msg}\ \mathtt{to}\ \mathtt{B};\mathtt{T}' \in Q$, *then*

$$\begin{cases} 1)(\mathtt{T},(\mathtt{AB}!\mathit{msg}),\underset{\mathtt{t}\in\vec{\mathtt{t}}}{\mathtt{rec}}\ \vec{\mathtt{t}}\ \{\mathtt{T}''\}) \in \delta & \textit{if}\ \mathtt{T}' = \mathtt{continue}\ \mathtt{t}, \underset{\mathtt{t}\in\vec{\mathtt{t}}}{\mathtt{rec}}\ \vec{\mathtt{t}}\ \{\mathtt{T}''\} \in \mathtt{T}_0 \\ 2)(\mathtt{T},(\mathtt{AB}!\mathit{msg}),\mathtt{T}') \in \delta & \textit{otherwise} \end{cases}$$

2. *if* $\mathtt{body}(\mathtt{T}) = \mathit{msg}\ \mathtt{from}\ \mathtt{B};\mathtt{T}' \in Q$, *then*

$$\begin{cases} 1)(\mathtt{T},(\mathtt{BA}?\mathit{msg}),\underset{\mathtt{t}\in\vec{\mathtt{t}}}{\mathtt{rec}}\ \vec{\mathtt{t}}\ \{\mathtt{T}''\}) \in \delta & \textit{if}\ \mathtt{T}' = \mathtt{continue}\ \mathtt{t}, \underset{\mathtt{t}\in\vec{\mathtt{t}}}{\mathtt{rec}}\ \vec{\mathtt{t}}\ \{\mathtt{T}''\} \in \mathtt{T}_0 \\ 2)(\mathtt{T},(\mathtt{BA}?\mathit{msg}),\mathtt{T}') \in \delta & \textit{otherwise} \end{cases}$$

3. *if* $\mathtt{T} = \mathtt{choice\ at}\ \mathtt{A}\ \{\mathtt{T_i}\}_{i\in I}$, *then:*

   a) *if* $\mathtt{T_i} = \mathit{msg}_i\ \mathtt{to}\ \mathtt{B};\mathtt{T}'$

$$\begin{cases} 1)\mathtt{T},(\mathtt{AB}!\mathit{msg}_i),\underset{\mathtt{t}\in\vec{\mathtt{t}}}{\mathtt{rec}}\ \vec{\mathtt{t}}\ \{\mathtt{T}''\}) \in \delta & \textit{if}\ \mathtt{T}' = \mathtt{continue}\ \mathtt{t}, \underset{\mathtt{t}\in\vec{\mathtt{t}}}{\mathtt{rec}}\ \vec{\mathtt{t}}\ \{\mathtt{T}''\} \in Q, \\ 2)\mathtt{T},(\mathtt{AB}!\mathit{msg}_i),\mathtt{T}') \in \delta & \textit{otherwise} \end{cases}$$

   b) *if* $\mathtt{T_i} = \mathit{msg}_i\ \mathtt{from}\ \mathtt{A};\mathtt{T}'$

$$\begin{cases} 1)(\mathtt{T},(\mathtt{BA}?\mathit{msg}_i),\underset{\mathtt{t}\in\vec{\mathtt{t}}}{\mathtt{rec}}\ \vec{\mathtt{t}}\ \{\mathtt{T}''\}) \in \delta & \textit{if}\ \mathtt{T}' = \mathtt{continue}\ \mathtt{t}, \underset{\mathtt{t}\in\vec{\mathtt{t}}}{\mathtt{rec}}\ \vec{\mathtt{t}}\ \{\mathtt{T}''\} \in Q \\ 2)(\mathtt{T},(\mathtt{BA}?\mathit{msg}_i),\mathtt{T}') \in \delta & \textit{otherwise} \end{cases}$$

We give two examples, Fig. 2.10 and Fig. 2.11, as to illustrate the translation.

```
rec t1 {
  m1 to B;
  m2 to B;
  continue t1}
```

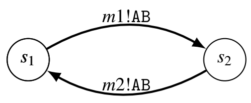


Figure 2.10.: Example of a local protocol $\mathtt{T}$ (left) and its CFSM translation (right).

The CFSM $\mathscr{A}(\mathtt{T})$ for the local protocol $\mathtt{T}$ from Fig. 2.10 is $\mathscr{A}(\mathtt{T}) = (Q, C, q_0, A, \delta)$. We first generate the states Q of $\mathscr{A}(\mathtt{T})$ from the suboccurrences of the initial local protocol $\mathtt{T}$. We name the sates s_1 and s_2 where

$$s_1 = \mathtt{T}$$
$$s_2 = (\mathtt{m2\ to\ B;\ continue\ t1;}).$$

Then $\mathscr{A}(\mathtt{T})$ is defined as follows: 1) $Q = \{s_1, s_2\}$; 2) $C = \{\mathtt{AB}, \mathtt{BA}\}$ 3) $q_0 = s_1$; 4) $A = \{\mathtt{m1},\ \mathtt{m2}\}$; 5) $\delta = \{(s_1, \mathtt{m1!AB}, s_2), (s_2, \mathtt{m2!AB}, s_1)\}$

```
rec t1 {
  m1 to B; rec t2 { m2 to B;
  choice at A {m3 to B;
               continue t2}
  or {m4 to B; continue t1}}}
```

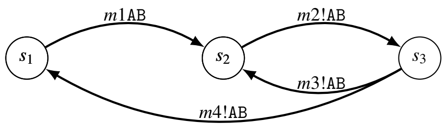


Figure 2.11.: Example of Local protocol $\mathtt{T}$ (left) and its CFSM translation (right).

Next we consider the local type $\mathtt{T}$, given on Fig. 2.11.

From the suboccurrences of the local protocol $\mathtt{T}$ we generate three states:

$$
\begin{aligned}
s_1 &= \mathtt{T} \\
s_2 &= \mathtt{rec\ t2\ \{\ m2\ to\ B;\ choice\ at\ A\ \{m3\ to\ B;\ continue\ t2\}\ or\ \{m4\ to\ B;\ continue\ t1\}} \\
s_3 &= \mathtt{choice\ at\ A\ \{m3\ to\ B;\ continue\ t2\}\ or\ \{m4\ to\ B;\ continue\ t1\}}
\end{aligned}
$$

Then the $\mathscr{A}(\mathtt{T}) = (Q, C, A, q_0, \delta)$ is defined as follows 1) $Q = \{s_1, s_2, s_3\}$; 2) $C = \{\mathtt{AB}, \mathtt{BA}\}$; 3) $q_0 = s_1$ 4) $A = \{\mathtt{m1,\ m2,\ m3,\ m4}\}$; 5) $\delta = \{(s_1, \mathtt{m1!AB}, s_2), (s_2, \mathtt{m2!AB}, s_3), (s_3, \mathtt{m3!AB}, s_2), (s_3, \mathtt{m4!AB}, s_1)\}$.

Before proving operational correspondence between a local type $\mathtt{T}$ and its corresponding $\mathscr{A}(\mathtt{T})$, we give a few auxiliary definitions.

Definition 2.4.4 (Sigma-unfold) *We define a function* $\mathtt{sigmaUnf} : \mathtt{T} \times \sigma \to \sigma$, *where* $\mathtt{T}$ *is a type and* σ *is a mapping from recursive variables to types.*

$$
\mathtt{sigmaUnf}(\mathtt{T}, \sigma) =
\begin{cases}
\bigcup\limits_{i \in I} \mathtt{sigmaUnf}(\mathtt{T_i}, \sigma) \text{ if } \mathtt{T} = \mathtt{choice\ at\ A}\ \{\mathtt{T_i}\}_{i \in I} \\
\mathtt{sigmaUnf}(\mathtt{T'}, \sigma) \text{ if } \mathtt{T} = msg\ \mathtt{to\ B}; \mathtt{T'} \\
\mathtt{sigmaUnf}(\mathtt{T'}, \sigma) \text{ if } \mathtt{T} = msg\ \mathtt{from\ B}; \mathtt{T'} \\
\mathtt{sigmaUnf}(\mathtt{T'}[\mathtt{rec}\ \vec{\mathtt{t}}\ \{\mathtt{T'}\} / \mathtt{continue\ t}]_{\forall \mathtt{t} \in \vec{\mathtt{t}}}, \sigma \bigcup\limits_{\mathtt{t}_i \in \vec{\mathtt{t}}} \{\mathtt{t}_i \mapsto \mathtt{T}\}) \\
\qquad\qquad \text{if } \mathtt{T} = \mathtt{rec}\ \vec{\mathtt{t}}\ \{\mathtt{T'}\} \text{ and } \forall \mathtt{t}_i \in \vec{\mathtt{t}}, \mathtt{t}_i \notin \sigma \\
\sigma \text{ if } \mathtt{T} = \mathtt{rec}\ \vec{\mathtt{t}}\ \{\mathtt{T'}\} \text{ and } \exists \mathtt{t'} \in \vec{\mathtt{t}} : \mathtt{t'} \in \sigma \\
\sigma \text{ if } \mathtt{T} = \mathtt{end}
\end{cases}
$$

We assume all recursive variables are distinct and also $\mathtt{rec}\ \vec{\mathtt{t}}\ \{\mathtt{T'}\}\sigma = \mathtt{T'}\sigma$. Hence, σ can contain $\mathtt{t} \in \vec{\mathtt{t}}$ and we apply the substitution σ without α-renaming.

Lemma 2.4.5 (Suboccurrences) *Given a local type* $\mathtt{T}$*, a suboccurrence* $\mathtt{rec}\ \vec{\mathtt{t}}\ \{\mathtt{T'}\}_{(t \in \vec{t})} \in$ *T and a substitution* σ *s.t* $\sigma = \mathtt{sigmaUnf}((\mathtt{T}, \varnothing))$*,then*

$$\mathtt{rec}\ \vec{\mathtt{t}}\ \{\mathtt{T'}\}_{(t \in \vec{t})}\sigma = \mathtt{T''} \text{ with } \{t \mapsto \mathtt{T''}\} \in \sigma$$

Lemma 2.4.6 (Soundness of the translation) *If* $\mathtt{T}$ *is a local Scribble protocol, then* $\mathtt{T} \approx \mathscr{A}(\mathtt{T})$.

Proof. In the proof we assume $\sigma = \mathtt{sigmaUnf}(\mathtt{T}, \varnothing)$. Also we assume $\mathtt{T} \neq \mathtt{end}$. When $\mathtt{T} = \mathtt{end}$ the lemma is trivially true since $\mathtt{T}$ produces an empty set of traces, δ is an empty relation and q_0, the initial state, is also a final state.
First, we consider $\mathtt{T} \lesssim \mathscr{A}(\mathtt{T})$. Next we prove that if $\mathtt{T} \xrightarrow{\ell} \mathtt{T}'$ then $\exists \mathtt{T_A}, \mathtt{T'_A} \in Q$ such that $\mathtt{T_A}\sigma = \mathtt{T}$ and $\mathtt{T'_A}\sigma = \mathtt{T}'$, and $(\mathtt{T_A}, \ell, \mathtt{T'_A}) \in \delta$.

The proof is by induction on the transition relation for local types. In all cases we assume that $\mathtt{T} = \mathtt{T_A}\sigma$.

- $\lfloor\text{SEND}\rfloor$ if the reduction is by $\lfloor\text{SEND}\rfloor$ we have
  $\mathtt{T_A}\sigma = (\mathtt{msg}\ \mathtt{to}\ \mathtt{B}; \mathtt{T'_A})\sigma = \mathtt{msg}\ \mathtt{to}\ \mathtt{B}; (\mathtt{T'_A}\sigma)$.
  Thus, $\mathtt{T_A}\sigma \xrightarrow{\ell} \mathtt{T'_A}\sigma$ where $\ell = \mathtt{msg!AB}$.
  Since $\mathtt{body}(\mathtt{T_A}) = \mathtt{msg}\ \mathtt{to}\ \mathtt{B}; \mathtt{T'_A}$ we proceed by case analysis on $\mathtt{T'_A}$.
  Case 1: $\mathtt{T'_A} \neq \mathtt{continue}\ \mathtt{t}$;
  By Def. 2.4.3(1-2) and $\mathtt{body}(\mathtt{T_A}) = \mathtt{msg}\ \mathtt{to}\ \mathtt{B}; \mathtt{T'_A} \Rightarrow (\mathtt{T_A}, \ell, \mathtt{T'_A}) \in \delta$.
  Case 2: $\mathtt{T'_A} = \mathtt{continue}\ \mathtt{t}$;
  We have that $\mathtt{T'_A}\sigma = \mathtt{continue}\ \mathtt{t}\ \sigma = \mathtt{T}''$, where $\{\mathtt{t} \mapsto \mathtt{T}''\} \in \sigma$.
  By $\lfloor\text{SEND}\rfloor$ we have $\mathtt{T_A}\sigma \xrightarrow{\ell} \mathtt{T}''$.
  By Def. 2.4.3(1-1) and $\mathtt{body}(\mathtt{T_A}) = \mathtt{msg}\ \mathtt{to}\ \mathtt{B}; \mathtt{T'_A}$ it follows that $(\mathtt{T_A}, \ell, \mathtt{rec}\ \vec{\mathtt{t}}\ \{\mathtt{T''_A}\}) \in \delta$ with $\mathtt{rec}\ \vec{\mathtt{t}}\ \{\mathtt{T''_A}\} \in \mathtt{T_0}$.
  By Lemma 2.4.5 we have $\mathtt{rec}\ \vec{\mathtt{t}}\ \{\mathtt{T''_A}\}\sigma = \mathtt{T}''$ and we conclude the case.
- $\lfloor\text{RECV}\rfloor$ is similar to Case $\lfloor\text{SEND}\rfloor$ and thus we omit.
- $\lfloor\text{CHOICE}\rfloor$ if the reduction is by $\lfloor\text{CHOICE}\rfloor$ we have
  $\mathtt{T_A}\sigma = (\mathtt{choice}\ \mathtt{at}\ \mathtt{A}\{\mathtt{T_{Ai}}\}_{i\in I})\sigma = \mathtt{choice}\ \mathtt{at}\ \mathtt{A}\{(\mathtt{T_{Ai}}\sigma)\}_{i\in I}$.
  Case 1 if $\mathtt{T_{Ai}}\sigma$ has the shape $(\mathtt{msg}_i\ \mathtt{to}\ \mathtt{B}; \mathtt{T'_{Ai}})\sigma = \mathtt{msg}_i\ \mathtt{to}\ \mathtt{B}; (\mathtt{T'_{Ai}}\sigma), \forall i \in I$ then we have $\mathtt{T_A}\sigma \xrightarrow{\ell} \mathtt{T'_{Aj}}\sigma$ for some $j \in I$ with $\ell = \mathtt{msg}_j\mathtt{!AB}$.
  Since $\mathtt{body}(\mathtt{T_A}) = \mathtt{T_A}$, we proceed by case analysis on $\mathtt{T'_{Aj}}$.
  Case 1.1: $\mathtt{T'_{Aj}} \neq \mathtt{continue}\ \mathtt{t}$;
  By Def. 2.4.3(3-a-2) and $\mathtt{body}(\mathtt{T_A}) = \mathtt{T_A}$ we have $(\mathtt{T_A}, \ell, \mathtt{T'_{Aj}}) \in \delta$.
  Case 1.2: $\mathtt{T_{Aj}}' = \mathtt{continue}\ \mathtt{t}$;
  (1*) We have that $\mathtt{T'_{Aj}}\sigma = \mathtt{continue}\ \mathtt{t}\ \sigma = \mathtt{T}''$, where $\{\mathtt{t} \mapsto \mathtt{T}''\} \in \sigma$.
  (2*) By $\lfloor\text{CHOICE}\rfloor$ we have $\mathtt{T_A}\sigma \xrightarrow{\ell} \mathtt{T}''$.
  By Def. 2.4.3(3-a-2) and $\mathtt{body}(\mathtt{T_A}) = \mathtt{T_A} \Rightarrow (\mathtt{T_A}, \ell, \mathtt{rec}\ \vec{\mathtt{t}}\ \{\mathtt{T''_A}\}) \in \delta$ with $\mathtt{rec}\ \vec{\mathtt{t}}\ \{\mathtt{T''_A}\} \in \mathtt{T_0}$.
  By Lemma 2.4.5 we have $\mathtt{rec}\ \vec{\mathtt{t}}\ \{\mathtt{T''_A}\}\sigma = \mathtt{T}''$.

Applying the IH to (1*) and (2*) we conclude the case.

Case 2 if $\mathtt{T_{Ai}}\sigma$ has the shape $(\mathtt{msg}_i\ \mathtt{to}\ \mathtt{B};\mathtt{T'_{Ai}})\sigma = \mathtt{msg}_i\ \mathtt{to}\ \mathtt{B};(\mathtt{T'_{Ai}}\sigma)$ this case is similar to Case 1 and thus we omit.

Note that since the normal form of local types does not allow for unguarded choice, hence, all possible transitions of $\mathtt{T_A}\sigma$ are the transitions from Case 1 and Case 2.

- $\lfloor\textsc{rec}\rfloor$ if the reduction is by $\lfloor\textsc{rec}\rfloor$ we have then $\mathtt{T_A}\sigma = (\mathtt{rec}\ \vec{\mathtt{t}}\ \{\mathtt{T'_A}\})\sigma = \mathtt{T'_A}\sigma$. We note that $\mathtt{T'_A}\sigma$ cannot be $\mathtt{continue}\ \mathtt{t}$ since unguarded recursive variables are not allowed. Hence, $\mathtt{T'_A}\sigma$ is either send, receive or choice and by IH and $\lfloor\textsc{send}\rfloor$, $\lfloor\textsc{recv}\rfloor$, $\lfloor\textsc{choice}\rfloor$ we conclude this case.

We next consider $\mathscr{A}(\mathtt{T}) \lesssim \mathtt{T}$. We prove that given a local protocol $\mathtt{T_0}$ if $(\mathtt{T_A},\ell,\mathtt{T'_A}) \in \delta$ then $\exists\mathtt{T}$ s.t. $\mathtt{T} = \mathtt{T_A}\sigma$ and $\mathtt{T} \xrightarrow{\ell} \mathtt{T'}$ and $\mathtt{T'} = \mathtt{T'_A}\sigma$ with $\sigma = \mathtt{sigmaUnf}(\mathtt{T_0},\varnothing)$. We proceed by case analysis on the transitions in δ.

Case 1 $\mathtt{T_A} = \mathtt{msg}\ \mathtt{to}\ \mathtt{B};\mathtt{T''_A}$ and $\ell = \mathtt{msg?AB}$.

Then $\mathtt{T'} = \mathtt{T_A}\sigma$ and we have by $\lfloor\textsc{send}\rfloor$ $\mathtt{T_A}\sigma \xrightarrow{\ell} \mathtt{T''_A}\sigma$.

Case 1.1 if $\mathtt{T''_A} = \mathtt{T'_A} \neq \mathtt{continue}\ \mathtt{t}$

The hypothesis follows from $\mathtt{T_A}\sigma \xrightarrow{\ell} \mathtt{T'_A}\sigma$.

Case 1.2 if $\mathtt{T''_A} = \mathtt{continue}\ \mathtt{t}$

By Def. 2.4.3 $\mathtt{T'_A} = \mathtt{rec}\ \vec{\mathtt{t}}\ \{\mathtt{T'''_A}\} \in \mathtt{T_0}, \mathtt{t} \in \vec{\mathtt{t}}$.

By Def. 2.4.4 and Lemma 2.4.5 we have $\mathtt{t} \mapsto \mathtt{T''}$ s.t. $\mathtt{rec}\ \vec{\mathtt{t}}\ \{\mathtt{T'''_A}\}\sigma = \mathtt{T''}$.

From IH and $\mathtt{T_A}\sigma \xrightarrow{\ell} \mathtt{T''_A}\sigma = \mathtt{T''} = \mathtt{rec}\ \vec{\mathtt{t}}\ \{\mathtt{T'''_A}\}\sigma = \mathtt{T'_A}\sigma$ we conclude the case.

Case 2 $\mathtt{T_A} = \mathtt{msg}\ \mathtt{from}\ \mathtt{B};\mathtt{T''_A}$ and $\ell = \mathtt{msg!AB}$.

Proceeds in a similar way as Case 2 and thus we omit.

Case 3 $\mathtt{T_A} = \mathtt{choice\ at}\{\mathtt{msg}_i\ \mathtt{to}\ \mathtt{B};\mathtt{T_{Ai}}\}_{i\in I}$

Then we have by $\lfloor\textsc{choice}\rfloor$

$\mathtt{T_A}\sigma = \mathtt{choice\ at}\{\mathtt{msg}_i\ \mathtt{to}\ \mathtt{B};\mathtt{T_{Ai}}\sigma\}_{i\in I} \xrightarrow{\mathtt{msg!AB}} \mathtt{T_{Aj}}\sigma$ for some $j \in I$.

Case 3.1 if $\mathtt{T_{Aj}} = \mathtt{T'_A} \neq \mathtt{continue}\ \mathtt{t}$

From IH and $\mathtt{T_A}\sigma \xrightarrow{\ell} \mathtt{T'_A}\sigma$ we conclude the case.

Case 3.2 if $\mathtt{T_{Aj}} = \mathtt{continue}\ \mathtt{t}$

By Def. 2.4.3 $\mathtt{T_{Aj}} = \mathtt{rec}\ \vec{\mathtt{t}}\ \{\mathtt{T''_A}\} \in \mathtt{T_0}, \mathtt{t} \in \vec{\mathtt{t}}$

By Def. 2.4.4 and Lemma 2.4.5 we have $\mathtt{t} \mapsto \mathtt{T''}$ s.t. $\mathtt{rec}\ \vec{\mathtt{t}}\ \{\mathtt{T'''_A}\}\sigma = \mathtt{T''}$.

We have that $\mathtt{T_A}\sigma \xrightarrow{\ell} \mathtt{T_{Aj}}\sigma = \mathtt{T''} = \mathtt{rec}\ \vec{\mathtt{t}}\ \{\mathtt{T'''_A}\}\sigma = \mathtt{T'_A}\sigma$, hence we conclude the case.

Case 4 $\mathtt{T_A} = \mathtt{choice\ at}\{\mathtt{msg}_i\ \mathtt{from}\ \mathtt{B};\mathtt{T_{Ai}}\}_{i\in I}$

Proceeds in a similar way as Case 3 and thus we omit.

Case 5 $\mathtt{T_A} = \mathtt{rec}\ \vec{\mathtt{t}}\ \{\mathtt{T''_A}\}$

Note that the $\mathtt{T}''_{\mathtt{A}}$ is either message send or message receive. Hence, By applying the IH and Case 1, 2 we conclude the case.

2.5. Concluding Remarks

In this chapter, we gave an overview of the protocol description language Scribble. We explained its formal foundations by proving correspondence between Scribble and MPST. Most importantly, we show that a global protocol corresponds to a system of CFSMs. This result has an important implication when building and verifying distributed systems. Mainly, it guarantees that global safety properties can be ensured through local, e.g, decentralised, verification. Such a setting thus not require synchronisation at runtime and therefore is more efficient to implement than a centralised approach. In the next chapter we build on this result to design and build a sound Scribble-based framework for runtime verification.

3. From Scribble Specifications to Runtime Monitors

This chapter presents a toolchain for session-based programming (hereafter conversation programming) in Python. Communication safety of a system of Python programs is guaranteed at runtime by monitors that check the execution traces comply with an associated protocol. Protocols are written in the protocol description language Scribble, presented in Chapter 2, with addition of logic formulas for more precise behaviour properties. We demonstrate the advantages, expressiveness and performance of dynamic protocol checking through use cases and benchmarks.

Framework overview. Fig. 3.1 illustrates the methodology of our framework. The development of a communication-oriented application starts with the specification of the intended interactions (the choreography) as a *global protocol* using the Scribble protocol description language, presented in Chapter 2.

Our toolchain validates that the global protocol satisfies MPST well-formedness properties, such as coherent branches (no ambiguity between participants about which branch to follow) and deadlock-freedom (between parallel flows). From a well-formed global protocol, the toolchain mechanically generates a Scribble *local protocol* (called a *projection*) for each participant (abstracted as a *role*) that is involved.

As a session is conducted at run-time, the monitor at each endpoint uses a finite state machine (FSM) representation of the local communication behaviour, generated from the local protocol for its role, to track its progress in the session. In our implementation, the FSM generation follows the correspondence between Scribble local protocols and communication automata, given in Section 2.4, and extended for handling parallel branches, as presented in [DY12]. In [DY12] the treatment of parallel branches in a protocol results in a state explosion. In our work, we avoid this problem, treating parallel branches by generating nested FSM structures. The

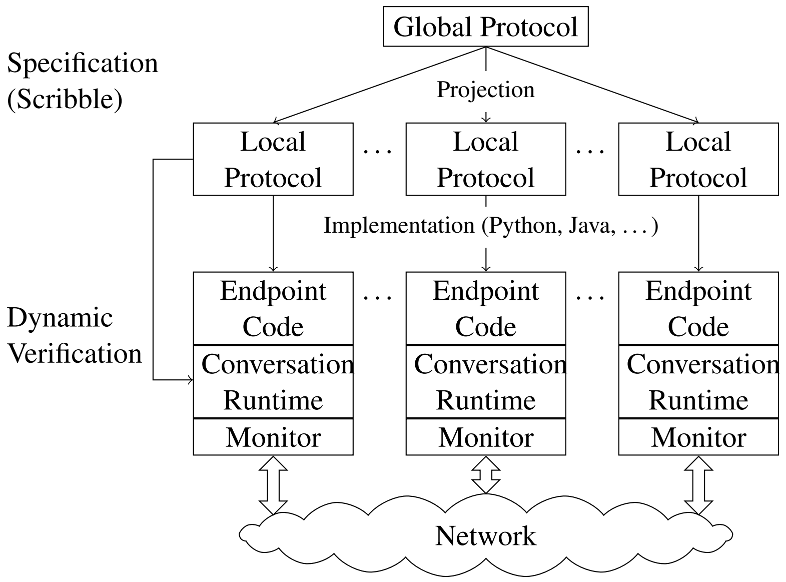


Figure 3.1.: Scribble methodology from global specification to local runtime verification

monitor validates the communication actions performed by the local endpoint, and the messages that arrive from the other endpoints, against the transitions permitted by the monitor's FSM. Each monitor thus works to protect both the endpoint from invalid actions by the network environment, and the network from incorrectly implemented endpoints.

Our dynamic MPST framework is designed in this way to ensure, via the decentralised monitoring of each local endpoint, that the progress of the session as a whole conforms to the original global protocol [BCD$^+$13], and that unsafe actions by a bad endpoint cannot corrupt the protocol state of other compliant endpoints.

We have integrated our framework into the Python-based runtime platform developed by Ocean Observatories Initiative (OOI) [OOIa]. The OOI is a project to establish a cyberinfrastructure for the delivery, management and analysis of scientific data from a large network of ocean sensor systems. Its architecture relies on the combination of high-level protocol specifications of network services (expressed as Scribble protocols [OOIb]) and distributed runtime monitoring to regulate the behaviour of third-party applications within the system [OOId]. Note that although this work is in collaboration with OOI, our implementation can be used orthogonally as a standalone monitoring framework for distributed Python applications.

Contributions and outline. We outline the structure of this chapter, summarising the contributions of each part:

Section 3.1 demonstrates our dynamic MPST framework through an example. We present a global-to-local projection of Scribble protocols, endpoint implementations, and local FSM generation.

Section 3.2 demonstrates an API for conversation programming in Python that supports standard socket-like operations, as well as event-driven interface. The API decorates conversation messages with meta information required by the monitors to perform dynamic verification.

Section 3.3 discusses the monitor implementation, focusing on key architectural requirements of our framework.

Section 3.4 evaluates the performance of the monitor implementation through a collection of benchmarks. The results show that conversation programming and run-time monitoring can be realised with low overhead.

Section 3.5 surveys theoretical works for MPST verification and explains the correspondence between a theoretical model and our implementation. We also discuss related and future works.

3.1. Scribble-based Dynamic Verification

This section illustrates the stages of our framework and its implementation through an use case, emphasising the properties verified at each verification stage. The presented use case is obtained from our industrial partners Ocean Observatory Institute (OOI) [OOIa] (use case UC.R2.13 ”Acquire Data From Instrument”). Fig. 3.3 shows a summary of the verification methodology, applied to this particular example.

3.1.1. Verification Steps

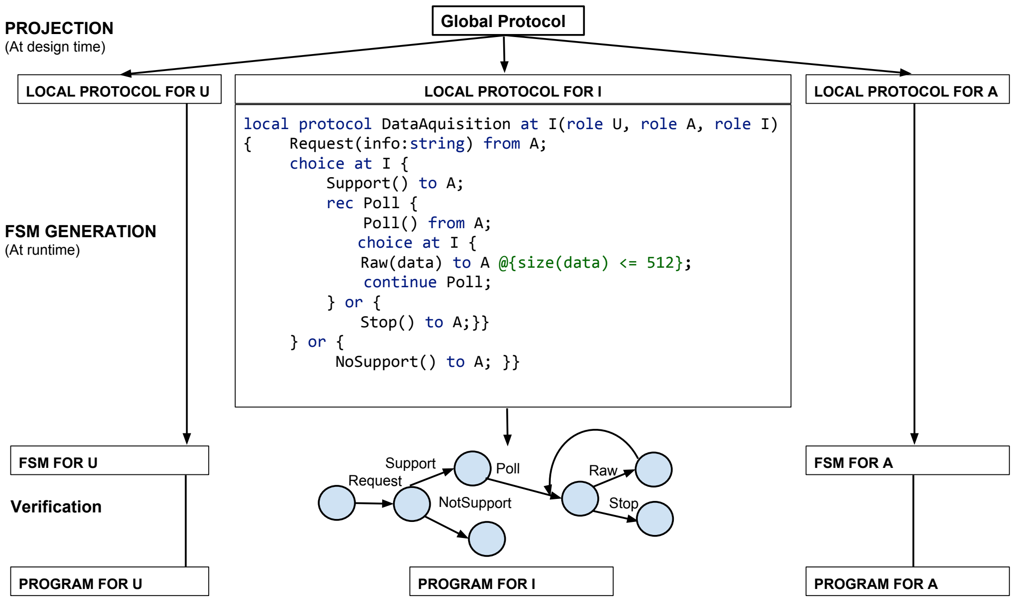
Global Protocol Correctness. The first level of verification is when designing a global protocol. A Scribble global protocol for the use case is listed in Fig. 3.2. Scribble describes interactions between session participants through message passing sequences, branches and recursion (as explained in Chapter 2). Each message has a label (an operator) and a payload. The Scribble protocol in Fig. 3.2

```
 1 global protocol DataAcquisition(  local protocol DataAcquisition
 2 role U, role A, role I) {           at U
 3 Request(string:info) from U to A;   (role U, role A, role I)}
 4 Request(string:info) from A to I;   Request(string:info) to A;
 5   choice at I {                     choice at I {
 6     Support() from I to A;
 7     rec Poll {                      rec Poll{
 8       Poll() from A to I;
 9       choice at I {                   choice at I {
10       Raw(data) from I to A
11       @{size(data) <= 512};           @{size(data) <= 512};
12       Formatted(data) from A to U;    Formatted(data) from A;
13       continue Poll;                  continue Poll;
14   } or {                              } or {
15     Stop() from I to A;
16     Stop() from A to U;}}               Stop() from A;}}
17   } or {                              } or {
18     NotSupported from I to A;
19     Stop() from A to I;
20     Stop from A to U;}}               Stop() from A;}}
```

Figure 3.2.: Global Protocol (left) and Local Protocol (right)

starts by protocol declaration, which specifies the name of the protocol, `Data Acquisition`, and its participating roles - a User (`U`), an Agent service (`A`) and an Instrument (`I`). The overall scenario is as follows: `U` requests via `A` to start streaming a list of resources from `I` (line 2–3). At line 4 `I` makes a choice whether to continue the interaction or not. If `I` supports the requested resource, `I` sends a message `Support` and the communication continues by `A` sending a `Poll` request to `I`. The raw resource data is sent from `I` to `A`, at `A` the data is formatted and forwarded to `U` (line 6–15). Line 10 demonstrates an *assertion* construct specifying that `I` is allowed to send data packages that are less than 512MB.

The Scribble toolchain validates that a protocol is well-formed and thus projectable for each role. For example, in each case of a choice construct, the deciding party (e.g. at `I`) must correctly communicate the decision outcome unambiguously to all other roles involved; a choice is badly-formed if the actions of the deciding party would cause a race condition on the selected case between the other roles, or if it is ambiguous to another role whether the decision has already been made or is still pending. A comprehensive overview of the Scribble syntax, and further references to the formal conditions for protocol correctness are described in Chapter 2.



```
local protocol DataAquisition at I(role U, role A, role I)
{    Request(info:string) from A;
     choice at I {
         Support() to A;
         rec Poll {
             Poll() from A;
             choice at I {
             Raw(data) to A @{size(data) <= 512};
              continue Poll;
         } or {
             Stop() to A;}}
     } or {
          NoSupport() to A; }}
```

Figure 3.3.: Verification Methodology for DataAcquisition protocol

Local protocol conformance. The second level of verification is performed at runtime and ensures that each endpoint program conforms to the local protocol structure. Local protocols specify the communication behaviour for each conversation participant. Local protocols are mechanically projected from a global protocol. A local protocol is essentially a view of the global protocol from the perspective of one participant role. Projection basically works by identifying the message exchanges where the participant is involved, and disregarding the rest, while preserving the overall interaction structure of the global protocol. Fig. 3.3 presents the local protocol for role `I`.

From the local protocols, an FSM is generated. The FSM, generated from the local protocol for `I` is shown in Fig. 3.3. At runtime, the endpoint program is validated against the FSM states. There are two main checks that are performed. First, we verify that the type (a label and payloads) of each message matches its specification (labels can be mapped directly to message headers, or to method calls, class names or other relevant artefacts in the program). Second, we verify that the flow of interactions is correct, i.e. interaction sequences, branches and recursions proceed as expected, respecting the explicit dependencies (e.g. `m1() from A to B; m2() from B to C`; imposes a causality at `B`, which is obliged to receive the messages from `A` before sending a message to `C`).

Policy validation. The final level of verification enables the elaboration of Scribble protocols using annotations (@{} in Fig. 3.2). The annotations function as API hooks to the verification framework: they are not verified by the MPST monitor itself, but are, instead, delegated to a third-party engine. Various policy domains (e.g. security policies) can be enforced by integrating engines for predicates on endpoint state, automata-based properties, etc., as extensions to the core protocol monitor. Our current implementation uses a Python library for evaluating basic predicates (e.g. the size check in Fig. 3.2), which is sufficient for the application protocols we have developed with [OOIa]. At runtime, the monitor passes the annotated information, along with the FSM state information, to the appropriate policy engine to perform the additional checks or calculations. To plug in an external validation engine, our toolchain API requires modules for parsing and evaluating the annotation expressions specified in the protocol.

3.1.2. Monitoring Requirements

Positioning. In order to guarantee global safety, our monitoring framework imposes complete mediation of communications: communication actions should not have an effect unless the message is mediated by the monitor. The tool implements this principal for both inline and outline monitor configurations. Inline monitoring relies on internal message interception: the local conversation runtime, in place at each endpoint, synchronously passes every message (on arrival or prior to dispatch) through a monitor component. Outline monitoring is realised by dynamically modifying the application-level network configuration to (asynchronously) route every message through a monitor. Our prototype is built over an Advance Messaging Queue Protocol (AMQP) [1] transport, where we use the AMQP exchange-to-exchange binding functionality to perform message rerouting. A monitor dispatcher is assigned to each network endpoint as a conversation gateway. The dispatcher can create new routes and spawn new monitor processes if needed, to ensure the scalability of this approach.

Message format. To monitor Scribble conversations, our toolchain relies on a small amount of message meta data, refered to as Scribble header, and embedded into the message payload. Messages are processed depending on their kind, as recorded in the first field of the Scribble header. There are two kinds of conversation messages: initialisation (exchanged when a session is started, carrying

Conversation API operation	Purpose
`create(protocol_name, config.yml)`	Initiate conversation, send invitations
`join(self, role, principal_name)`	Accept invitation
`send(role, op, payload)`	Send a message
`recv(role)`	Receive message from role
`recv_async(self, role, callback)`	Asynchronous receive

Figure 3.4.: The core Python Conversation API operations

information such as the protocol name and the role of the monitored process) and in-session (carrying the message operation and the sender/receiver roles). Initialisation messages are used for routing reconfiguration, while in-session messages are the ones checked for protocol conformance.

Principals and Conversation runtime. A principal (an application) implements a protocol behaviour using the Conversation API. The API is built on top of a Conversation Runtime. The runtime provides a library for instantiating, managing and programming Scribble protocols and serialising and (de)serializing conversation messages. The library is implemented as a thin wrapper over an existing transport library. The API provides primitives for creating and joining a conversation, as well as primitives for sending and receiving messages.

3.2. Conversation Programming in Python

The Python Conversation API, a new message passing API, offers a high-level interface for safe conversation programming, mapping the interaction primitives of session types to lower-level communication actions on concrete transports. The API primitives are displayed in Fig. 3.4. In summary, the API provides functionality for (1) session initiation and joining, (2) basic send/receive.

3.2.1. Conversation Initiation

The `Conversation.create` method initiates a new conversation. It creates a fresh conversation id and the required AMQP objects (principal exchange and queue), and sends an invitation for each role specified in the protocol. Invitations are sent to principals. Principal names direct the routing of invitation message to the right endpoint. Each invitation carries a role, a principal name and a name for a Scribble local specification file. For simplicity, we use a configuration file to provide the mapping between roles and principals; another option would be a naming registry. An example of the configuration file (invitation section) for the `DataAcquisition` example is:

```
...
invitations:
  -role: U
   principal name: alice
   local capability: DataAcquisition.spr
  -role: A
   principal name: bob
   local capability: DataAcquisition.spr
  -role: I
   principal name: carol
   local capability: DataAcquisition.spr
```

An invitation is accepted using the `Conversation.join` method. It establishes an AMQP connection and, if one does not exist, creates an invitation queue for receiving invitations.

We demonstrate the usage of the API in a Python implementation of the local protocol projected for the User role. Listing 3.1 gives the User Role implementation. First, the `create` method of the Conversation API initiates a new conversation instance of the `DataAcquisition` protocol (Fig. 3.2), and returns a token that is used to join the conversation locally. The `config.yml` file specifies which network principals will play which roles in this session and the runtime sends invitation messages to each principal. The `join` method confirms that the endpoint is joining the conversation as the principal `alice` playing the role `User`. Once the invitations are sent and accepted (via `Conversation.join`), the conversation is established and the intended message exchange can proceed. As a result of the initiation procedure, the runtime at every participant has a mapping (conversation table) between each role and their AMQP addresses.

3.2.2. Conversation Message Passing

The API provides standard send/receive primitives. Send is asynchronous, meaning that a basic send does not block on the corresponding receive; however, the basic receive does block until the complete message has been received. An asynchronous receive (`recv_async`) is also provided to support event-driven usage of the conversation API. These asynchronous features map closely to those supported

Listing 3.1: Python program for A: single threaded

```
class ClientApp(BaseApp):
  def start(self):
    c = Conversation.create('DataAcquisition',
'config.yml')
    c.join('A', 'alice')

    resource_request = c.recv('U')
    c.send('I', resource_request)
    req_result = c.recv('I')

    if (req_result == SUPPORTED):
      c.send('I', 'Poll')
      op, data = c.recv('I')

      while (op !=  'Stop'):
        formatted_data = format(data)
        c.send('U', formatted_data)
      c.send('U', stop)
    else:
        c.send(['U', 'I'], stop) #broadcast
        c.stop()
```

by Pika [1], a Python transport library that adopts continuation-passing style even for synchronous (blocking) calls. Pika is used as the underlying transport library in our implementations.

Each message signature in a Scribble specification contains an operation and payload (message arguments). The API does not mandate how the operation field should be treated, allowing the freedom to interpret the operation name in various ways, e.g. as a plain message label, an RMI method name, etc. We treat the operation name as a plain label.

Following its local protocol, the program for `A` receives a request from `U` and forwards the message to `I`. The `recv` returns a pair, (label, payload) of the message. When the message does not have a payload, only the label is returned (`req_result = c.recv('I')`). The `recv` method can also take the source role as a single argument (`c.recv('I')`), or additionally the label of the desired message (`c.recv('I', 'Request')`). The `send` method called on the conversation channel `c` takes, in this order, the destination role, message operator and payload values as arguments. In our example, the received payload `resource_request` is forwarded without modifications to `I`. Send is asyn-

[1] http://pika.readthedocs.org/

Listing 3.2: Python program for A: event-driven

```
class ClientApp(BaseApp):
  def start(self):
    c = Conversation.create('DataAcquisition',
'config.yml')
    c.join('A', 'alice')
    c.recv_async('U', on_request_received)

  def on_request_received(self, conv, op, msg):
    if (op == SUPPORTED):
      conv.send('I', 'Poll')
      conv.recv_async('I', 'on_data_received')
    else: conv.send(['I', 'U'], 'Stop')# broadcast


  def on_data_receive(self, conv, op, payload):
    if (op != 'Stop'):
      formatted_data = format(payload)
      c.send('U', formatted_data)
    else:
        conv.send('U', 'Stop')
        conv.stop()
```

chronous, meaning that the operation does not block on the corresponding receive; however, the basic receive does block until the complete message has been received. After `A` receives the reply from `I`, the program checks the label value `req_result` using conditional statements, `if (req_result=='Supported')`. If `I` replies with `'Supported'`, `A` enters a loop, where it continuously sends a `'Poll'` requests to `I` and after receiving the result from `I`, formats the received data (`format(data)`) and resends the formatted result to `U`.

Event-driven conversations. For asynchronous, non-blocking receives, the Conversation API provides `recv_async` to be used in an event-driven style. Listing 3.2 shows an alternative implementation of the `user` role using callbacks. The method accepts as arguments a callback to be invoked when a message is received.

We first create a conversation variable similar to the single threaded implementation. After joining the conversation, `A` registers a callback to be invoked when a message from `U` is received (`on_request_received`). The callback executions are linked to the flow of the protocol by taking the conversation id as an argument (e.g. `conv`). It also accepts as arguments the label for the message (`op`) and the payload (`msg`). In the message handler for `Request`, `A` forwards the received payload to `I` and registers a new message handler for the next message. Although the

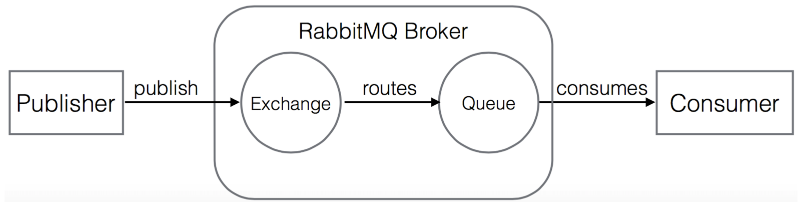


Figure 3.5.: AMQP networks

event-driven API promotes a notably different programming style, our framework monitors both this implementation and that in Listing 4.4 transparently without any modifications.

3.3. Runtime Monitors

3.3.1. Monitoring AMQP Networks

First, we explain the basic model of AMQP middlewares. An AMQP network consists of a federation of distributed message brokers (we use RabbitMQ) supporting transparent store-and-forward unicast and multicast messaging. AMQP brokers host two basic messaging abstractions, exchanges and queues. Fig. 3.5 illustrates a workflow for sending a message from a producer to a consumer. A producer sends messages to an exchange at its broker, from the exchange messages are routed to a queue, which is linked to a message consumer. Every message has an associated *routing key*, which is specified in the AMQP header of the message. When a queue is created, it is *bound* to an exchange via a *binding key*. An exchange delivers a message to a queue if the routing key in the message header corresponds to the binding key between the queue and the exchange.

Next we explain how we use AMQP abstractions to route messages. As explained in Section 3.1.2, there are two kinds of messages in our system: *invitation* messages (and corresponding acknowledgements) and *in-session* (*conversation*) messages. The former are used to establish new sessions between endpoints, while the latter belong to specific sessions. Depending on the message type either shared channels or session channels are used to route the messages. Channels are implemented in AMQP as a tuple of an exchange and a routing key. Session channels have an exchange and a routing key that depend on the conversation id, sender and receiver roles. Shared channels use the default or a well-known exchange with routing keys depending on principal names (see Fig. 3.6 for an illustration of channel implementations).

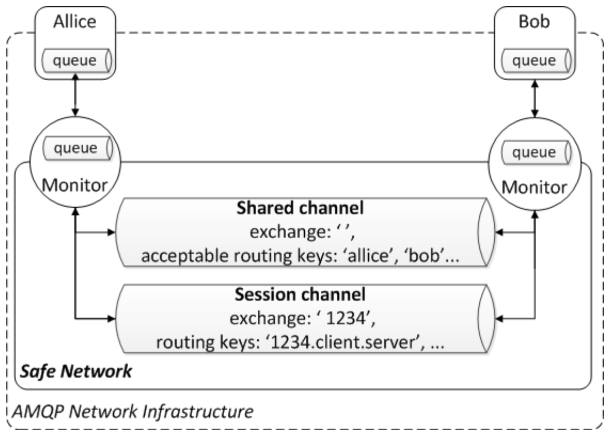


Figure 3.6.: Configuration of distributed session monitors for an AMQP-based network

Fig. 3.6 depicts our outline monitor configuration. The interception mechanism is based on message forwarding. A principal has at least one queue for consuming messages, although the number of queues can be tuned to use separate queues for invitations and roles.

We outline a concrete scenario. Principal Alice is authenticated and connected to her local broker.

1. Authentication creates a network access point for Alice (the Monitor circle in Fig. 3.6). The access point consists of a new conversation monitor instance, monitor queues (monitor as a consumer), and an exchange. Alice is only permitted to send messages to this exchange.
2. Alice initiates a new session (creates an exchange with id 1234 in Fig. 3.6) and dispatches an invitation to principal Bob. The invitation is received and checked by Alice's monitor and then dispatched on the shared channel, from where it is rerouted to Bob's Monitor.
3. Bob's monitor checks the invitation, generates the local FSM and session context for Bob and Bob role (for example client), and allocates a session channel (with exchange: 1234 and routing keys matching Bob's role ($1234.client.*$ and $1234.*.client$). The invitation is delivered to Bob's queue.

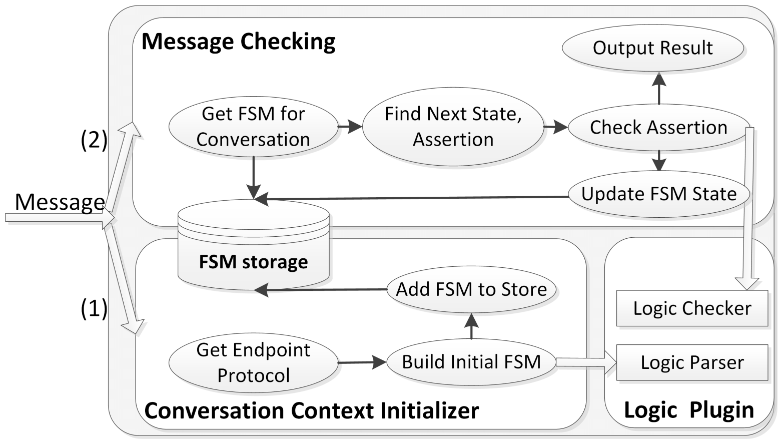


Figure 3.7.: Monitor workflow for (1) invitation and (2) in-conversation messages

4. Any message sent by Alice (e.g. to Bob) in this session is similarly passed by the monitor and validated. If valid, the message is forwarded to the session channel to be routed. The receiver's monitor will similarly but independently validate the message.

The monitor implementation is compatible with a range of monitor configurations. At one end of the spectrum is *inline monitoring*, where the monitor is embedded into the endpoint code. Then there are various configurations for *outline monitoring*, where the monitor is positioned externally to its component. In the OOI project, our focus has been to integrate our framework for inline monitoring due to the architecture of the OOI message interceptor stack [OOId].

3.3.2. Monitor Implementation

Fig. 3.7 depicts the main components and internal workflow of our prototype monitor. The lower part relates to conversation initiation. The *invitation* message carries (a reference to) the local protocol for the invitee and the conversation id (global protocols can also be exchanged if the monitor has the facility for projection.)

We use a parser generator (ANTLR) to produce, from a Scribble local protocol, an abstract syntax tree with MPST constructs as nodes. The tree is traversed to generate a finite state machine, represented in Python as a hash table, where each entry has the shape:

$$(\mathit{current\_state},\ \mathit{transition})\ \mapsto\ (\mathit{next\_state},\ \mathit{assertion},\ \mathit{var})$$

where *transition* is a quadruple $(interaction\ type, label, sender, receiver)$, *interaction type* is either *send* or *receive* and *var* is a variable binder for a message payload. We number the states using a generator of natural numbers. Note that the FSM generation is based on the translation of local Scribble protocols to FSMs, presented in Chapter 2.

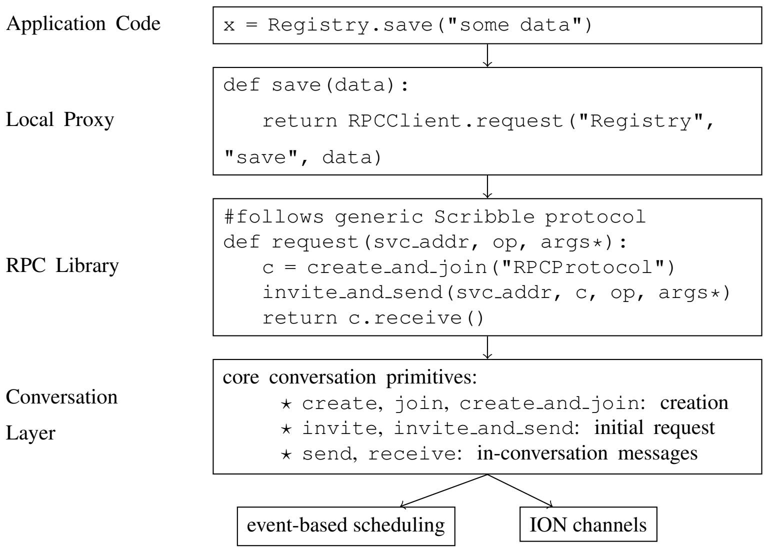
When a *send/receive* node is visited, two states are generated, and a linking transition of type send or receive is added to the transition table. Dummy states are created on entering a *choice* subtree and then again for each of the choice branches. An empty transition connects a dummy *initial choice* state and *branch* states. A recursion is handled by keeping a mapping between a *recursion label* and its *recursion start* state. Processing a node *continue* results in an empty transition between a current state and a corresponding recursion start state.

The algorithm for generating FSM from a MPST protocol is presented formally in Chapter 2. The implementation differs in the treatment of parallel sub-protocols (i.e. unordered message sequences). For efficiency, we implement the translation to generate a nested FSM for each conversation thread, avoiding the potential state explosion that comes from constructing their product. This allows FSM generation in polynomial time and space in the length of the local protocol. The (nested) FSMs are stored in a hash table with conversation id as the key. Due to standard MPST well-formedness (message label distinction), any nested FSM is uniquely identifiable from any unordered message, i.e. message-to-transition matching in a conversation FSM is deterministic.

The upper part of Fig. 3.7 relates to *in-conversation* messages, which carry the conversation id (matching an entry in the FSM hash table), sender and receiver fields, and the message label and payload. This information allows the monitor to retrieve the corresponding FSM (by matching the message signature to the FSM's transition function). Assertions associated to communication actions are evaluated by invoking a library for Python predicate evaluation.

3.4. Evaluation

Our dynamic MPST verification framework has been implemented and integrated into the current release of the Ocean Observatories platform [OOIc]. This section reports on our integration efforts and the performance of our framework.



```
x = Registry.save("some data")
```

```
def save(data):
   return RPCClient.request("Registry",
"save", data)
```

```
#follows generic Scribble protocol
def request(svc_addr, op, args*):
   c = create_and_join("RPCProtocol")
   invite_and_send(svc_addr, c, op, args*)
   return c.receive()
```

core conversation primitives:
- ⋆ `create, join, create_and_join`: creation
- ⋆ `invite, invite_and_send`: initial request
- ⋆ `send, receive`: in-conversation messages

Figure 3.8.: Translation of an RPC command into lower-level conversation calls

3.4.1. Experience: OOI Integration

The current release of OOI is based on a Service-Oriented Architecture, with all of the distributed system services accessible by RPC. As part of their efforts to move to agent-based systems in the next release, and to support distributed governance for more than just individual RPC calls, we engineered the following step-by-step transition. The first step was to add our Scribble monitor to the message interceptor stack of their middleware [OOId]. The second was to propose our conversation programming interface to the OOI developers. To facilitate the use of session types without obstructing the existing application code, we preserved the interface of the RPC libraries but replaced the underlying machinery with the distributed runtime for session types (as shown in Fig. 3.8, the RPC library is now realised on top of the Conversation Layer). As wrappers to the conversation primitives, all RPC calls are now automatically verified by the inline MPST monitors. This approach was feasible because no changes were required to existing application code, but at the same time, developers now have the option to use the Conversation API directly for conversations more complex than RPC. The next step in this ongoing integration work involves porting higher-level and more complex OOI application protocols, such as distributed agent negotiation [OOIb], to Scribble specifications and Conversation API implementations.

	10 RPCs (s)	
RPC Lib	0.103	
No Monitor	0.108	+4%
Monitor	0.122	+13%

Table 3.1.: Original OOI RPC vs. conversation-based RPC with monitoring disabled/enabled

Seq States	No-Mon (s)	Mon (s) (s)	
10	0.92	0.95	+3.2%
100	8.13	8.22	+1.1%
1000	80.31	80.53	+0.8%

Par States	No-Mon (s)	Mon (s)	
10	0.45	0.49	+8%
100	4.05	4.22	+4.1%
1000	40.16	41.24	+2.7%

Table 3.2.: Conversation execution time for an increasing number of sequential and parallel states

3.4.2. Monitor Performance

The potential performance overhead that the Conversation Layer and monitoring could introduce to the system is an important consideration. The following performance measurements for the current prototype show that our framework can be realised at a reasonable cost. Table 3.1 presents the execution time comparing RPC calls using the original OOI RPC library implementation and the conversation-based RPC with and without monitor verification. The RPC protocol is shown below.

```
global protocol RPC (role R, role P){
request(string) from R to P;
reply(string) from P to R;
}
```

The protocol represents a `request` message and a subsequent `reply` message, the string payload is the string representation of the labels, and hence is negligible. On average, 13% overhead is recorded for conversations of 10 consecutive RPCs, mostly due to the FSM generation from the textual local Scribble protocol (our implementation currently uses Python ANTLR); the cost of message validation itself is negligible in comparison.

The second benchmark gives an idea of how well our framework scales beyond basic RPC patterns. The complexity of the internal monitor overhead is bound by the number of FSM states, because the most computation intensive operation of the monitor checking process is a search in the hashtable of all protocol states as to find the next state. We measure two scenarios, which we believe, are representative to demonstrate the overhead incurred by checking.

- Increasing session length (number of messages), for protocols. This case is marked as “Seq States” in Table 3.2. This is a best-case scenario. The FSM table for each role contains only two states, hence searching for the next FSM state is negligible. We use the protocol pattern, given below, and the results in the table represent the protocol execution time when repeating the recursive body 10, 100 or 1000 times respectively.

```
rec Loop {
choice at R {
  request(string) from R to P;
  reply(string) from P to R;
  continue Loop;
  } or {
  stop(string) from R to P;
  ok(string) from P to R;
}
```

- Increasing protocol size (increasing number of states in the protocol). The protocol is parameterised on an integer number *n*, which determines the number of protocol states. This case is marked as “Par States” in Table 3.2. We measure the protocol execution time for the pattern, given below, and show the results for *n* being 10, 100 and 1000.

```
par {
  request_1(string) from R to P;
  reply_1(string) from P to R; }
and {
  request_2(string) from R to P;
  reply_2(string) from P to R; }
  }
...
and {
  request_n(string) from R to P;
  reply_n(string) from P to R; }
```

  `Par` blocks in Scribble represent unordered message exchange. Hence, ordering of messages is preserved only inside a parallel branch, and not between the branches. For example, in the above protocol `R` can send messages `request_1`, ..., `request_n` in any order and similarly `P` is allowed to receive them in any order. Therefore, for the monitor to verify that a given message is correct (in the worst case) it should traverse all protocol states.

Hence, this pattern demonstrates a worst case scenario, where the complexity of monitor checking is $\mathcal{O}(n)$.

The results, displayed in Table 3.2, show that the overall verification architecture (Conversation Layer and inline monitor) scales reasonably with increasing session length (number of message exchanges) and increasing protocol states. Two benchmark cases are compared. The main case “Monitor” (Mon) is fully monitored, i.e. FSM generation and message validation are enabled for both the client and server. The base case for comparison “No Monitor” (No-Mon) has the client and server in the same configuration, but monitors are disabled (messages do not go through the interceptor stack). As above, we found that the overhead introduced by the monitor when executing conversations of increasing number of recursive and parallel states is again mostly due to the cost of the initial FSM generation. We also note that the relative overhead decreases as the session length increases, because the one-time FSM generation cost becomes less prominent. For dense FSMs, the worse case scenario results in linear overhead growth w.r.t. the number of parallel branches, i.e the number of states that can be executed in any order.

In both of the above tables, the presented figures are the mean time for the client and server, connected by a single-broker AMQP network, to complete one conversation after repeating the benchmark 100 times for each parameter configuration. The client and server Python processes (including the conversation runtime and monitor) and the AMQP broker were each run on separate machines (Intel Core2 Duo 2.80 GHz, 4 GB memory, 64-bit Ubuntu 11.04, kernel 2.6.38). Latency between each node was measured to be 0.24 ms on average (ping 64 bytes). The full source code of the benchmark protocols and applications and the raw data are available from the project page [seta].

3.4.3. Use Cases

We conclude our evaluation with some remarks on use cases we have examined. Table 3.3 features a list of protocols, sourced from both the research community and our industry partner [OOIa]. We have written protocols in Scribble and have tested our monitor implementation on more realistic protocol specifications. A natural question for our methodology, being based on explicit specification of protocols, is the overhead imposed on developers w.r.t. writing protocols, given that a primary motivation for the development of Scribble is to reduce the design and testing effort for distributed systems. Among these use cases, we found the aver-

Use Cases from research papers	Global Scribble (LOC)	FSM Memory (B)	Generation Time (s)
A vehicle subsystem protocol [KMM07]	8	840	0.006
Map web-service protocol [GG07]	10	1040	0.010
A bidding protocol [LHJ05]	26	1544	0.020
Amazon search service [HBH$^+$10]	12	1088	0.010
SQL service [Sal10]	8	1936	0.009
Online shopping system [GCN$^+$07]	10	1024	0.008
Travel booking system [GCN$^+$07]	16	1440	0.013
Use Cases from OOI and Savara			
A purchasing protocol [Se]	11	1088	0.010
A banking example [OOIb]	16	1564	0.013
Negotiation protocol [OOIb]	20	1320	0.014
RPC with timeout [OOIb]	11	1016	0.013
Resource Access Control [OOIb]	21	1854	0.018

Table 3.3.: Use case protocols implemented in Scribble

age Scribble global protocol is roughly 10 LOC, with the longest one at 26 LOC, suggesting that Scribble is reasonably concise.

The main factors that may affect the performance and scalability of our monitor implementation, and which depend on the shape of a protocol, are (i) the time required for the generation of FSMs and (ii) the memory overhead that may be induced by the generation of nested FSMs in case of parallel blocks and interrupts. Table 3.3 measures these factors for each of the listed protocols. The time required for FSM generation remains under 20 ms, measuring on average to be around 10 ms. The memory overhead also remains within reasonable boundaries (under 2.0 KB), indicating that FSM caching is a feasible optimisation approach. The full Scribble protocols can be found at [seta].

From our experience of running our conversation monitoring framework within the OOI system, we expect that, in many large distributed systems, the cost of a decentralised monitoring infrastructure would be largely overshadowed by the raw cost of communication (latency, routing) and other services running at the same time. Considering the presented results, we thus believe the important benefits in terms of safety and management of high-level applications come at a reasonable cost and would be a realistic mechanism in many distributed systems.

3.5. Related work

In this section, we first explain the correspondence between a theoretical model for MPST-based monitoring and the implementation, presented in this chapter. Then we discuss two alternative approaches for MPST verification of dynamic languages. Then we give an overview of tools and frameworks for runtime verification, targeted at checking behavioural constraints.

3.5.1. Formal Foundations of MPST-based Runtime Verification

Our implementation is formalised in a theory for MPST-based verification of networks, first proposed in [CBD$^+$12], and later extended in [BCD$^+$13], and presented in detail in [Che13]. [CBD$^+$12] only gives an overview of the desired properties, and requires all local processes to be dynamically verified through the protections of system monitors, while [BCD$^+$13] presents a framework for semantically precise decentralised run-time verification, allowing mixing of statically and dynamically components. In addition, the routing mechanism of AMPQ networks is explicitly presented in [BCD$^+$13], while in [CBD$^+$12] it is implicit.

In summary, [BCD$^+$13] formalises specifications (based on local types) used to guard the runtime behaviour of processes in a network. Processes are given as asynchronous π-calculus with fine grained primitives for session initiation. Specifications are embedded into system monitors, each wrapping a principal to ensure that the ongoing communication conforms to the given specification. Reduction semantics consist of reduction rules for monitors, processes, and transport. Monitors communicate through the use of global queues; if the message is correct it is handled by the process, otherwise the process is unaffected.

Next we explain the correspondence between the asynchronous π-calculus with fine-grained primitives for session initiation and our Python API, and how the routing mechanism of AMQP brokers and queues is formalised in the theory as a network transport of routers and a global queue. Regarding the specification language, in [BCD$^+$13] specifications are given as local types. Instead, for efficient checking, we use the generation of communicating finite state machines (CFSMs) from local Scribble protocols, which are equivalent to local types, as has been shown in Chapter 2.

Networks. AMQP networks with Python monitors are formalised, in [BCD$^+$13], as a collection of principals $[P]_\alpha$, representing Python processes, a monitor, which is augmented with a session environment for specification checking and corresponds to the FSM checking in the implementation, and with a global transport $\langle r; h \rangle$, which is a pair a routing table r and a queue h, and abstracts the usage of AMQP middleware.

More precisely, the AMQP brokers is abstracted in the theory as a routing table. Messages are delivered to monitors based on mappings in the routing table as to reflect the forwarding of messages to principal queues by the AMQP broker. The routing table is a finite map from roles to principals. The global queue used as part of the transport is indexed on a session name, a message sender and a receiver and reflects the binding keys, in the implementation, which link AMQP exchanges and queues (see Fig. 3.6).

A delicate technical difference between the theory and the implementation lies in handling of out-of-order delivery of messages when messages are sent from different senders to the same receiver. Asynchrony poses a challenge in the treatment of out-of-order asynchronous message monitoring, and thus, to prevent false positive results, in the theoretical model, a type-level permutations of actions is required, e.g a monitor checks messages up to permutations. The use of global queues and local permutations is inefficient in practice, and thus we have implemented the theoretical model of a global queue as different physical queues. Specifically, we introduce a queue per a pair of roles, which ensures messages from the same receivers are delivered in order and are not mixed with messages from other roles. This model is semantically equivalent to a model of a global indexed queue, permitting permutation of messages.

Processes. Our Python API embodies the primitives of the asynchronous π-calculus with fine grained primitives for session initiation, presented in [BCD$^+$13]. The correspondence is given in Fig. 3.9. Note that the API does not stipulate the use of a recursion and a conditional, which appear in the syntax of session π-calculus, since these constructs are handled by native Python constructs. The `create` method, which, we remind, creates a fresh conversation id and the required AMQP objects (principal exchange and queue), and sends an invitation for each role specified in the protocol, corresponds to the action $\overline{a}\langle s[r] : T \rangle$, which sends on the shared channel a, an invitation to join the fresh conversation s as the role of r with a specification T. In the implementation, this information is codified in the message header, which as we have explained contains the new session id (abstracted as s), the name of the local Scribble protocol (e.g T) and the role (e.g

Conversation API operation	Purpose
`create(protocol_name, config.yml)`	$\overline{a}\langle s[r] : T\rangle$
`join(self, role, principal_name)`	$a(y[r] : T).P$
`send(role, op, payload)`	$k[r_1, r_2]!l\langle e\rangle$
`recv(role)`	$k[r_1, r_2]?\{l_i(x_i).P_i\}_{i\in I}$
`recv_async(self, role, callback)`	–

Figure 3.9.: The core Python Conversation API operations and their session *pi-calulus* counterparts

r). The invitation action $a(y[r] : T).P$ models session `join`. As a result of `join` new queues and a routing bindings are created. For example, when Bob joins a conversation with id of 1234 as the role of `client`, as shown in Fig. 3.6, an AMQP binding `1234.client.`$*$ is created , which ensures all messages to the role of a `client` are delivered to Bob. The reduction rule for $a(y[r] : T).P$, in the semantics in [BCD+13], reflects this behaviour by adding a record in the routing table. The primitive for sending a message $k[r_1, r_2]!l\langle e\rangle$ corresponds to the API call `send`, and results in sending a message of type $s[r_1, t_2]!l\langle e\rangle$, which in the implementation is codified in the message header, consisting of session id k, sender r_1, receiver r_2. label l and a payload e.

Properties of monitored networks. Finally, we give an overview of the fundamental properties of monitored networks as presented in [BCD+13]. Due to the correspondence explained above, these properties are preserved in the context of the monitor implementation, presented in this chapter.

Local safety states that a monitored process respects its local protocol, i.e. that dynamic verification by monitoring is sound.

Local transparency states that a monitored process has equivalent behaviour to an unmonitored but well-behaved process, e.g. statically verified against the same local protocol.

Global safety states that a system satisfies the global protocol, provided that each participant behaves as if monitored

Global transparency states that a fully monitored network has equivalent behaviour to an unmonitored but well-behaved network, i.e. in which all local processes are well-behaved against the same local protocols.

Session fidelity states that, as all message flows of a network satisfy global specifications, whenever the network changes because some local processes take actions, all message flows continue to satisfy global specifications.

3.5.2. Alternative Approaches

In this subsection, we survey two recent works which offer alternative approaches and verification of session types systems with dynamically typed participants and give valuable insights for future directions of our work.

A recent work [JGP16] investigates dynamically monitoring communication to enforce adherence to session types in a higher-order setting. In addition, a more advanced treatment of errors is presented, through a system of blame assignment. The architecture and the message passing model assumed in the paper are similar to ours. Specifically, monitors are assigned for each end of a communication channel similarly to our assignment of monitors and roles, and monitors enforce the order of which messages are sent and received according to their session type. They present a system of polarized channels, insisting message queues to have a definite direction, which reflects the queue creation in our system. The monitor only checks outgoing messages since the incoming messages are already verified by a monitor. [2] After surveying different monitoring approaches, authors justify the placement of a monitor at the end of the queue as the more convenient one to provide relatively precise blame assignment. To assign blame, in [JGP16], the monitor maintains a graph data structure that records process spawns through the execution of the entire system. When an alarm is raised a monitor assigns blame *jointly* to all direct ancestor of the failed participant in the graph. The work is theoretical and thus is not strictly related to our implementation. However, extending our implementation with blame assignment capabilities is an interesting topic for future investigation. Without delegation, blame assignment is trivial since the process to blame is the one producing the wrong type. We have already planned to add delegation to the monitor and this feature has already been implemented in Scribble. Unfortunately, the addition of delegation complicates significantly the blame assignment process. The results in [JGP16] are a promising first step towards this extension.

Another work that explores the safe interplay of statically and dynamically typed program fragments in the context of session types is [Thi14]. It presents a session typed functional calculus with synchronous communication and augments it with a dynamic type. It integrates gradual typing and binary session types. The proxy

[2]This is also a viable option in our system if we assume trusted monitors at all endpoints.

process introduced in their system to mediate the communication between dynamically forked processes can be seen as a monitor. The proxy forwards the operations between the channels prescribed by the session types. It applies the cast operations from the session coercion before writing to the other end of the channel. Although the work presents a theoretical fragment, the recent proposal for Python 3.5 for addition of typing hints through gradual typing suggests a promising future direction for the integration of their calculus in Python. However, there are numerous challenges left, most importantly in their system the communication is synchronous and the session types used are binary. It also permits more advanced treatment of choice, which allows a receiver to accept fewer alternatives than the ones provided by the sender.

3.5.3. Runtime Verification

Distributed runtime verification. The work in [ADM12] explores runtime monitoring based on session types as a test framework for multi-agent systems (MAS). Global session types are specified as cyclic Prolog terms in Jason (a MAS development platform). Their monitor is centralised (thus no projection facilities are discussed), and neither formalisation, global safety property nor proof of correctness is given in [ADM12]. The approach is later extended in [BMA15], where no changes are required to the monitored agents and hence existing MASs can be monitored without accessing to their code, but the monitor detects a violation only after it took place and even in case of a protocol violation the MAS execution goes on.

Other works, notably from the multi-agent community, have studied distributed enforcement of global properties through monitoring. A distributed architecture for local enforcement of global laws is presented by Zhang et al. [ZSM07], where monitors enforce *laws* expressed as event-condition-action. In [MU00], monitors may trigger sanctions if agents do not fulfil their obligations within given deadlines. Unlike such frameworks, where all agents belonging to a group obey the same set of laws, our approach asks agents to follow personalised laws based on the role they play in each session.

In runtime verification for Web services, the works [LHJ05, LJH06] propose FSM-based monitoring using a rule-based declarative language for specifications. These systems typically position monitors to protect the safety of service interfaces, but do not aim to enforce global network properties. The work of Cambronero et al. [CDVM11] transforms a subset of Web Services Choreography Description Language into timed-automata and prove their transformation is correct with respect to timed traces. Their approach is model-based, static and centralised, and does not treat either the runtime verification or interrupts. Baresi et al. [BGG04] develop a runtime monitoring tool for BPEL with assertions. A major difference is that BPEL approaches do not treat or prove global safety. BPEL is expressive, but does not support distribution and is designed to work in a centralised manner. Kruger et al. [KMM10] propose a runtime monitoring framework, projecting MSCs to FSM-based distributed monitors. They use aspect-oriented programming techniques to inject monitors into the implementation of the components. Our outline monitoring verifies conversation protocols. Gan [GCN$^+$07] follows a similar but centralised approach of [KMM10]. As a language for protocol specification, a main advantage of Scribble (i.e. MPST) over alternatives, such as message sequence charts (MSC), CDL and BPML, is that MPST has both a formal basis and an in-built mechanism (projection) for decentralisation, and is easily integrated with the language framework, as demonstrated for Python in this chapter.

Language-based monitoring tools. A survey on monitoring tools can be found in [FHR13]. Here we list only a few tools, that we consider, are most related. Jass [Jas] is a precompiler tool for monitoring the dynamic behaviour of sequential objects and the ordering of method invocations by annotating Java programs with specifications that can be checked at runtime. Other approaches to runtime verification of program execution by monitors generated from language-based specifications include: aspect-oriented programming [Lar]; other works that use process calculi formalisms, such as CSP [Jas]; monitors based on FSM skeletons associated to various forms of underlying patterns [AAC$^+$05, ATdM07]; and the analysis of dynamic parametric traces [ATdM07]. Our monitor framework has been influenced by these works and shares similarities with some of the presented runtime verification techniques. However, the target program domain and focus of our work are different. Our framework is specifically designed for decentralised monitoring of distributed programs with diverse participants and interleaving sessions, as opposed to monitoring the execution of a single program and verifying its local properties. The basis of our design and implementation is the theory of multiparty session types, over which we have developed practically motivated extensions to the type language and the methodology for runtime verification.

3.6. Concluding Remarks

We have implemented a dynamic verification framework for Python programs based on Scribble protocols. Our implementation automates distributed monitoring by generating FSMs from local protocols. We have integrated our framework in a large cyberinfrastructure project [OOIa] and we have demonstrated monitoring overhead induces a reasonable overhead. This work is the first step towards methodologies for better specification and more rigorous governance of network conversations in distributed systems. In the next two chapters we extend and improve the framework, adding features for exception handling and time constraints.

4. Monitoring Interruptible Systems

This chapter presents the implementation and formalisation of a new construct for verifying *asynchronous multiparty session interrupts*. Asynchronous session interrupts express communication patterns in which the behaviour of the roles following the default flow through a protocol segment may be overruled by one or more other roles concurrently raising asynchronous interrupt messages. Previous attempts at incorporating exception-like constructs into session type theory have been limited in their practical application and cannot express our use case patterns: [CHY08] is restricted to binary session types, [CHY08, Car09] do not support nested interrupts or continuations, and [CGY10, CVB^{+}16], although multiparty, relies on synchronous exception flags which are not feasible in general distributed systems.

Extending MPST with asynchronous interrupts is challenging because the inherent “communication race conditions“ that may arise conflict with the MPST safety properties. Taking a continuous stream of messages from a producer to a consumer as a simple example: if the consumer sends an interrupt message to the producer to pause or end the stream, stream messages (those already in transit or subsequently dispatched before the interrupt arrives at the producer) may well continue arriving at the consumer for some time after the interrupt is dispatched. This scenario is in contrast to the patterns permitted by standard session types, where the safety properties guarantee that no message is ever lost or redundant by virtue of disallowing all protocols with potential races.

This chapter introduces a novel approach based on reifying the concept of *scopes* within a protocol at the runtime level when an instance of the protocol is executed. A scope designates a sub-region of the protocol, derived from its syntactic structure, on which certain communication actions, such as interrupts, may act on the region as a whole. At run-time, every message identifies the scope to which it belongs as part of its meta data. From this information and by tracking the local progress in the protocol, the runtime at each endpoint in the session is able to resolve discrepancies in protocol state by discarding incoming messages that have become irrelevant due to an asynchronous interrupt. This mechanism is transparent to the user process, and although performed independently by each distributed endpoint, preserves global safety for the session.

We integrate the new interrupt construct in our framework for runtime monitoring presented in Chapter 3. Here we recall the monitoring process. As a session is conducted at run-time, the monitor at each endpoint uses a finite state machine (FSM) representation of the local communication behaviour, generated from the local protocol for its role, to track interactions in a session.

The FSM generation, presented in Chapter 3, is extended to support interruptible protocol scopes. We treat interruptible scopes by generating nested FSM structures. In the case of scopes that may be entered multiple times by recursive protocols, we use dynamic FSM nesting (conceptually, a new sub-FSM is created each time the scope is entered) corresponding to the generation of fresh scope names in the syntactic model.

Our dynamic MPST framework with interrupts ensures, from the decentralised monitoring of each local endpoint, that the progress of the session as a whole conforms to the original global protocol, and that unsafe actions by an incorrectly implemented endpoint cannot corrupt the protocol state of other compliant endpoints.

The OOI use cases motivating the work presented in this chapter include a variety of RPC-based service calls (request-reply) with timeout interrupts, and publish-subscribe applications where the consumer or other parties can interrupt to pause, resume and stop remotely driven sensor feeds; we use the latter for the main running example in this chapter. Although the existing features of Scribble (i.e. those previously established in MPST theory) are sufficiently expressive for many practical protocols, we observed that these important patterns could not be directly or naturally represented without interrupts.

Contributions and outline. We outline the structure of this chapter, summarising the contributions of each part:

Section 4.1 explains an OOI use case for the extension of Scribble with asynchronous session interrupts. This is a new feature for MPST, giving the first general mechanism for nested, multiparty interrupts. We discuss why adding this feature is a challenge in session types.

Section 4.2 discusses the extension of the Python implementation with a new construct of scopes, and demonstrates the global-to-local projection of interruptible Scribble protocols, endpoint implementations, and local FSM generation for monitoring. Section 4.2.1 demonstrates the Python API for conversation programming in Python, including event-driven conversations. Section 4.2.2 discusses the treatment of asynchronous interrupts in the monitor implementation.

Section 4.3 presents the supporting theory for asynchronous session interrupts. We show the soundness of our framework by proving session fidelity, asserting that the decentralised verification of a system always conforms to its global specification.

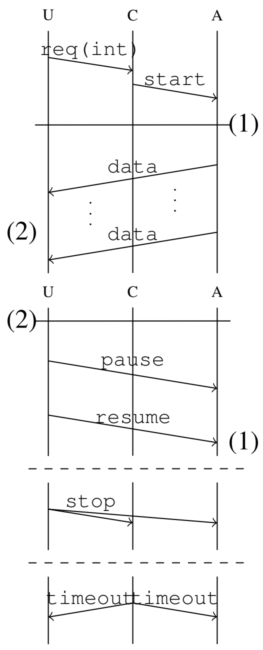
4.1. Communication Protocols with Asynchronous Interrupts

This section expands on why and how we extend Scribble to support the specification and verification of asynchronous session interrupts, henceforth referred to as just interrupts. Our running example is based on an OOI project use case, which we have distilled to focus on session interrupts. Using this example, we outline the technical challenges of extending Scribble with interrupts.

4.1.1. Use Case: Resource Access Control (RAC)

As is common practice in industry, the cyberinfrastructure team of the OOI project [OOIa] manages communication protocol specifications through a combination of informal sequence diagrams and prose descriptions. Fig. 4.1 (left) gives an abridged version of a sequence diagram given in the OOI documentation for the Resource Access Control (RAC) use case [OOIb], regarding access control of users to sensor devices in the ION cyberinfrastucture for data acquisition. In the ION setting, a User interacts with a sensor device via its Agent proxy (which interacts with the device via a separate protocol outside of this example). ION Controller agents manage concerns such as authentication of users and metering of service usage.

For brevity, we omit from the diagram some of the data types to be carried in the messages and focus on the *structure* of the protocol. The depicted interaction can be summarised as follows. The protocol starts at the top of the left-hand diagram. `User` sends `Controller` a `request` message to use a sensor for a certain amount of time (the `int` in parentheses), and Controller sends a `start` to Agent.



```
global protocol ResourceAccessControl(
  role User as U,
  role Controller as C, role Agent as A){
// U requests the device for some duration
req(duration:int) from U to C;
start() from C to A;
interruptible {// U, C and A in scope
  rec X {
    interruptible {// U and A in scope
      rec Y {
        data() from A to U;
        continue Y;}
    } with {// Interrupts A in Y
        pause() by U;}
    resume() from U to A;
    continue X;
  }
} with {// Interrupts A and C/U in X
  stop() by U;// Before duration expired

  timeout() by C;// Duration is up
 }
}
```

Figure 4.1.: Sequence diagram (left) and Scribble protocol (right) for the RAC use case

The protocol then enters a phase (denoted by the horizontal line) that we label (1), in which Agent streams `data` messages (acquired from the sensor) to User. The vertical dots signify that Agent produces the stream of data freely under its own control, i.e. without application-level control from User. User and Controller, however, have the option at any point in phase (1) to move the protocol to the phase labelled (2), below.

Phase (2) comprises three alternatives, separated by dashed lines. In the upper case, User *interrupts* the stream from Agent by sending Agent a `pause` message. At some subsequent point, User sends a `resume` and the protocol returns to phase (1). In the middle case, User interrupts the stream, sending both Agent and Controller a `stop` message. This is the case where User does not want any more sensor data, and ends the protocol for all three participants. Finally, in the lower case, Controller interrupts the stream by sending a `timeout` message to User and Agent. This is the case where, from Controller's view, the session has exceeded the requested duration, so Controller interrupts the other two participants to end the protocol. Note this diagram actually intends that `stop` (and `timeout`) can arise anytime after (1), e.g. between `pause` and `resume` (a notational ambiguity that is compensated by additional prose comments in the specification).

4.1.2. Interruptible Multiparty Session Types

Fig. 4.1 (right) shows a Scribble protocol that formally captures the structure of interaction in the Resource Access Control use case and demonstrates the uses of our new extension for asynchronous interrupts. Besides the formal foundations, we find the Scribble specification is more explicit and precise, particularly regarding the combination of compound constructs such as choice and recursion, than the sequence diagram format, and provides firmer implementation guidelines for the programmer (demonstrated in Section 4.2.1).

A Scribble protocol starts with a header declaring the protocol name (given as `ResourceAccessControl` in Fig. 4.1) and role names for the participants (three roles, aliased in the scope of this protocol definition as `U`, `C` and `A`). Lines 5 and 6 straightforwardly correspond to the first two communications in the sequence diagram. The Scribble syntax for message signatures, e.g. `req(duration:int)`, means a message with *operator* (i.e. header, or label) `req`, carrying a *payload* `int` annotated as `duration`. The `start()` message signature means operator `start` with an empty payload.

We now come to "phase" (1) of the sequence diagram. The new `interruptible` construct captures the informal usage of protocol phases in disciplined manner, making explicit the interrupt messages and the *scope* in which they apply. Although the syntax has been designed to be readable and familiar to programmers, `interruptible` is an advanced construct that encapsulates several aspects of asynchronous interaction, which we discuss at the end of this section.

The intended communication protocol in our example is clarified in Scribble as two nested `interruptible` statements. The outer statement, on lines 7–22, corresponds to the options for User and Controller to end the protocol via the `stop` and `timeout` interrupts. An `interruptible` consists of a main body of protocol actions, here lines 8–17, and a set of interrupt message signatures, lines 18–22. The statement stipulates that each participant behaves by either (a) following the protocol specified in the body until finished for their role, or (b) raising or detecting a specified interrupt at any point during (a) and exiting the statement. Thus, the outer `interruptible` states that `U` can interrupt the body (and end the protocol) by a `stop()` message, and `C` by a `timeout()`.

```
// Well-formed,
// but incorrect semantics:
par {
  rec Y {
    data() from A to U;
    continue Y; }
} and {
  // Does not stop the recursion
  pause() from U to A;
}
resume() from U to A;
```

```
// Naive mixed-choice
// is not well-formed
choice at  A {
// A should make the choice
  rec Y {
    data() from A to U;
    continue Y; }
} or {
  // ..not U
  pause() from U to A;
}
resume() from U to A;
```

Figure 4.2.: Naive, incorrect interruptible encoding attempts using parallel (left) and choice (right)

The body of the outer `interruptible` is a labelled recursion statement with label `X`. The `continue X;` inside the recursion (line 16) causes the flow of the protocol to return to the top of the recursion (line 8). This recursion corresponds to the loop implied by the sequence diagram that allows User to pause and resume repeatedly. Since the recursion body always leads to the `continue`, Scribble protocols of this form state that the loop should be driven indefinitely by one role, until one of the interrupts is raised by *another* role. This communication pattern cannot be expressed in multiparty session types without `interruptible`.

The body of the `X`-recursion is the inner `interruptible`, which corresponds to the option for User to pause the stream. The stream itself is specified by the `Y`-recursion, in which `A` continuously sends `data()` messages to `U`. The inner `interruptible` specifies that `U` may interrupt the `Y`-recursion by a `pause()` message, which is followed by the `resume()` message from `U` before the protocol returns to the top of the `X`-recursion.

4.1.3. Challenges of Asynchronous Interrupts in MPST

The following summarises our observations from the extension and usage of MPST with asynchronous interrupts. We find the operational meaning of `interruptible` as illustrated in the above example, is readily understood by architects and developers, which is a primary consideration in the design of Scribble. However, the question is how to preserve the desired safety properties for session based systems in the presence of `interruptible`. The challenges stem from the fact that `interruptible` combines several tricky, from a session typing view, aspects of communication behaviours that session type systems traditionally aim to prohibit, in order to prevent communication races and thereby ensure the desired safety properties.

A key aspect, due to asynchrony, is that an interrupt may occur in parallel to the actions of the roles being interrupted (e.g. `pause` by `U` to `A` while `A` is streaming `data` to `U`). Although standard MPST (and Scribble) support parallel sub-protocols (unordered message delivery), the interesting point here is that the nature of an interrupt is to preclude further actions in another parallel flow under the control of a different role, whereas the basic MPST parallel does not permit such interference. Fig. 4.2 (left) is a naively incorrect attempt to express this aspect without interruptible: the second parallel path is never able to intefere with the first to actually stop the recursion.

Another aspect is that of mixed choice in the protocol, in terms of both communication direction (e.g. `U` may choose to either receive the next `data` or send a `stop`), and between different roles (e.g. `U` and `C` independently, and possibly concurrently, interrupt the protocol) due to multiparty. Moreover, the implicit interrupt choice is truly optional in the sense that it may never be selected at runtime. The basic choice in standard MPST (e.g. as defined in [HYC08, DY12]) is inadequate because it is designed to safely identify a single role as the decision maker, who communicates exactly one of a set of message choices unambiguously to all relevant roles.Fig. 4.2 (right) demonstrates a naive mixed choice that is not well-formed (it breaks the unique sender condition, presented in Section 2.1.2).

Due to the asynchronous setting, it is also important that `interruptible` does not require implicit synchronisations to preserve communication safety.

4.2. Programming and Verification of Interruptible Systems

This section discusses implementation details of our monitoring framework and the accompanying Python API (Conversation API) for writing monitorable, distributed interruptible MPST programs. The design and implementation of a general, asynchronous MPST interrupt mechanism in the protocol language and API for endpoint implementation is a contribution of this thesis.

We first recall the verification methodology of our framework from Chapter 3. Developers write endpoint programs in native Python using the Conversation API

Conversation API operation	Purpose
`create(protocol_name, config.yml)`	Initiate conversation, send invitations
`join(self, role, principal_name)`	Accept an invitation
`send(role, op, payload)`	Send a message
`recv(role)`	Receive a message from a role
`recv_async(self, role, callback)`	Asynchronous receive
`scope(msg)`	Create a conversation scope
`close()`	Close the connection (implicit)

Figure 4.3.: The core Python Conversation API operations

extended with a new construct for handling *scopes*. The execution of these operations at each endpoint is performed by the local conversation library runtime. The API enables MPST verification of message exchanges by the monitor by embedding a small amount of MPST meta data (e.g. conversation identifier, message kind and operator, source and destination roles), based on the actions and current state of the endpoint, into the message payload. For each conversation initiated or joined by an endpoint, the monitor generates an FSM from the local protocol for the role of the endpoint. The monitor uses the FSM to track the progress of this conversation according to the protocol, validating each message (via the meta data) as it is sent or received.

4.2.1. Interrupt Handling via Conversation Scopes

Fig. 4.3 recalls the core API operations. The basic API operations are (1) session initiation and joining, (2) basic send/receive. We extend the API to provide functionality for *conversation scope* management for handling interrupt messages.

We demonstrate the usage of the API in a Python implementation of the local protocol projected for the User role. Fig. 4.4 gives the local protocol and its implementation.

First, the `create` method of the Conversation API (line 6, right) initiates a new conversation instance of the `ResourceAccessControl` protocol (Fig. 4.1), and returns a token that can be used to join the conversation locally. The `config.yml` file specifies which network principals will play which roles in this session and the runtime sends invitation messages to each. The `join` method confirms that the endpoint is joining the conversation as the principal `alice` playing the role `User`, and returns a conversation channel object for performing the subsequent communication operations. Once the invitations are sent and accepted (via

```
1  class UserApp(BaseApp):
2    user, controller, agent =
3        ['User', 'Controller', 'Agent']
4    def start(self):
5      self.buffer = buffer(MAX_SIZE)
6      conv = Conversation.create(
7               'RACProtocol', 'config.yml')
8      c = conv.join(user, 'alice')
9      c.send(controller, 'req', 3600)
10     with c.scope('timeout', 'stop') as c_x:
11       while not self.should_stop():
12         with c_x.scope('pause') as c_y:
13           while not self.buffer.is_full():
14             data = c_y.recv(agent)
15             self.buffer.append(data)
16           c_y.send_interrupt('pause')
17         use_data(self.buffer)
18         self.buffer.clear()
19         c_x.send(agent, 'resume')
20       c_x.send_interrupt('stop')
21     c.close()
```

```
local protocol RACProtocol
 at U (role C, role A){
 req(duration:int) to C;
 interruptible {
  rec X {
   interruptible {
    rec Y {
     data() from A;
     continue Y;
     }
    } with {
     pause() by U;
    }
    resume() to A;
    continue X;
   }
  } with {
    stop() by U;
    timeout() by C;
   }
}
```

Figure 4.4.: Python implementation (left) for the User role and Scribble local protocol (right)

`Conversation.join`), the conversation is established and the intended message exchanges can proceed.

Following its local protocol, the User program sends a request to the `controller`, stating the duration for which it requires access to `agent`. The `send` method called on the conversation channel `c` takes, in this order, the destination role, message operator and payload values as arguments. The `recv` method can take the source role as a single argument, or additionally the operator of the desired message.

Interrupt handling. The implementation of the User program demonstrates a way of handling conversation interrupts by combining conversation scopes with the Python `with` statement (an enhanced try-finally construct). We use `with` to conveniently capture interruptible conversation flows and the nesting of interruptible scopes, as well as automatic `close` of interrupted channels in the standard manner, as follows. The API provides the `c.scope()` method, as in line 10, to create and enter the scope of an `interruptible` Scribble block (here, the outer interruptible of the RAC protocol). The `timeout` and `stop` arguments associate these message signatures as interrupts to this scope. The conversation channel c_x returned by scope is a wrapper of the parent channel c that (1) records the current scope of every message sent in its meta data , (2) ensures every send and receive operation is guarded by a check on the local interrupt queue, and (3) tracks the nesting of scope contexts through nested with statements. The interruptible scope of c_x is given by the enclosing with (lines 10–20); if, e.g., a timeout is received within this scope, the control flow will exit the with to line 21. The inner with (lines 12–16), corresponding to the inner interruptible block, is associated with the pause interrupt. When an interrupt, e.g. pause in line 16, is thrown (send_interrupt) to the other conversation participants, the local and receiver runtimes each raise an internal exception that is either handled or propagated up, depending on the interrupts declared at the current scope level, to direct the interrupted control flow accordingly. The delineation of interruptible scopes by the global protocol, and its projection to each local protocol, thus allows interrupted control flows to be coordinated between distributed participants in a structured manner.

The scope wrapper channels are closed (via the with) after throwing or handling an interrupt message. For example, using c_x after a timeout is received (i.e. outside its parent scope) will be flagged as an error. By identifying the scope of every message from its meta data, the conversation runtime (and monitor) is able to compensate for the inherent discrepancies in protocol synchronisation, due to asynchronous interrupts between distributed endpoints, by safely discarding out-of-scope messages. In our example, the User runtime discards data messages that arrive after pause is thrown. To prevent the loss of such messages in the application logic when the stream is resumed, we could extend the protocol to simply carry the id of the last received data element in the payload of the resume (in line 19). The API can also make the discarded data available to the programmer through secondary (non-monitored) operations.

Message handlers with scopes. As demonstrated in Chapter 3, our Python API supports asynchronous message receive through the primitive recv_async. The construct is used to register a method that should be invoked on message receive. To support event-driven programming with interrupts, we extend the implementation presented in Chapter 3. The difference is in the signature and semantics of event handlers. More precisely, each event handler is associated not only to the conversation channel (as in Chapter 3), but is registered for a particular scope. Therefore, if an interrupt is received, but the protocol state is not in the same scope

```
1  class UserApp(BaseApp):
2    def start(self):
3      self.buffer = buffer(MAX_SIZE)
4      conv = Conversation.create(
5            'RACProtocol', config.yml)
6      c = conv.join(user, 'alice')
7      # request 1 hour access
8      c.send(controller, 'req', 3600)
9      c_x = c.scope('timeout', 'stop')
10     c_y = c_x.scope('pause')
11     c_y.recv_async(agent, recv_handler)
12
13   def recv_handler(self, c, op, payload):
14     with c:
15       if self.should_stop():
16         c.send_interrupt('stop')
17   elif self.buffer.is_full():
18     self.process_buffer(c, payload)
19   else:
20     self.buffer.append(payload)
21     c.recv_async(agent, recv_handler)
22
23   def process_buffer(self, c, payload):
24     with c:
25       c_x = c.send_interrupt('pause')
26       use_data(self.buffer, payload)
27       self.buffer.clear()
28       c_x.send(agent, 'resume')
29       c_y = c_x.scope('pause')
30       c_y.recv_async(agent, recv_handler)
```

Figure 4.5.: Event-driven conversation implementation for the User role

as the scope written in the conversation header of the interrupt message, the interrupt will be discarded.

Fig. 4.5 shows an alternative implementation of the `user` role using callbacks. We first enter the nested conversation scopes according to the potential interrupt messages (lines 9 and 10). The callback method (`recv_handler`) is then registered using the `recv_async` operation (line 11). The callback executions are linked to the flow of the protocol by taking the scoped channel as an argument (e.g. `c` on line 13). Note that if the `stop` and `pause` interrupts were not declared for these scopes, line 16 and line 25 would be considered invalid by the monitor. When the buffer is full (line 17), the user sends the pause interrupt. After raising an interrupt, the current scope becomes obsolete and the channel object for the parent scope is returned. After the data is processed and the buffer is cleared,

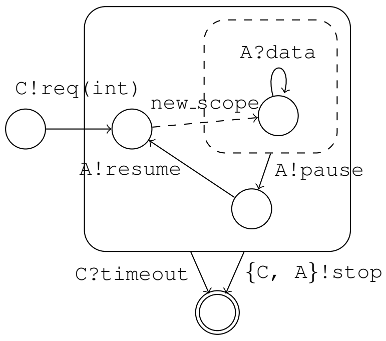


Figure 4.6.: Nested FSM generated from the User local protocol

the `resume` message is sent (line 28) and a fresh scope is created and again registered for receiving data events (line 29). Although the event-driven API promotes a notably different programming style, our framework monitors both this implementation and that in Fig. 4.4 transparently without any modifications.

4.2.2. Monitoring Interrupts

The algorithm for generating a FSM from a MPST protocol is presented in Chapter 2 and its implementation is given in Chapter 3. Here we outline only the changes needed to support interrupts.

FSM generation for interruptible local protocols makes use of nested FSMs. Each `interruptible` induces a nested FSM given by the main interruptible block, as illustrated in Fig. 4.6 for the User local protocol. The monitor internally augments the nested FSM with a scope id, derived from the signature of the interruptible block, and an interrupt table, which records the interrupt message signatures that may be thrown or received in this scope. Interrupt messages are marked via the same meta data field used to designate invitation and in-conversation messages, and are validated in a similar way except that they are checked against the interrupt table. However, if an interrupt arrives that does not have a match in the interrupt table of the immediate FSM(s), the check searches upwards through the parent FSMs; the interrupt is invalid if it cannot be matched after reaching the outermost FSM.

$$\begin{array}{rl} G_{\mathtt{ResCont}} = & \mathtt{U}{\to}\mathtt{C}:\mathrm{req}; C{\to}\mathtt{A}:\mathrm{start} \\ & \{\!|\mu X. \\ & \quad \{\!|\mu Y.\mathtt{A}{\to}\mathtt{U}:\mathrm{data}; Y|\!\}^{\mathsf{c}_2}\langle \mathrm{pause}\ \mathtt{by}\ \mathtt{U}\rangle; \\ & \quad \mathtt{U}{\to}\mathtt{A}:\mathrm{resume}; X \\ & |\!\}^{\mathsf{c}_1}\langle \mathrm{stop}\ \mathtt{by}\ \mathtt{U}, \mathrm{timeout}\ \mathtt{by}\ \mathtt{C}\rangle; \mathtt{end} \end{array}$$

Figure 4.7.: Global type for the Resource Access Control protocol Fig. 4.1

4.3. Multiparty Session Types with Asynchronous Interrupts

This section presents the underlying session type theory with interrupts and its correctness result, *session fidelity*, justifying our choice for the implementation of interrupt messages. We show that interruptible blocks can be treated through the use of *scopes*, a new formal construct that makes explicit, through an explicit identifier, the domain of interrupts.

In our theory, we manipulate global types, which correspond to global protocols, as presented in Chapter 2, and local types, which are used to express monitored behaviours of processes [BCD+13].

Global session type for RAC use case. In Fig. 4.7, to introduce the syntax of global types informally, we first show a global type which corresponds to the Scribble protocol Fig. 4.1. The formal syntax for the new constructs for handling scopes will be given in the next subsection. The first line denotes the two interactions at the start between the three participants. The outer loop is embedded inside a scope construct, explicitly by c_1. The inner loop is embedded inside another scope c_2. The information directly after the scope describes how it can be interrupted.

We insist on the fact that the formal global type G_{ResCont} is very close to its Scribble counterpart in Fig. 4.1. The main difference comes from the explicit naming of the scopes (here, c_1 and c_2). Note that:

- Our types are equi-recursive and every scope annotation has to be different, so in this representation c_2 actually stands for an infinite set of scopes $(\mathsf{c}_2^i)_{i\geq 0}$, one for every unfolding of the recursion of X.
- This example requires to enrich the syntax presented below with interruptible constructs accepting two interrupt messages, which can be performed either by slightly updating the semantics or by encoding the example into

two nested interruptible constructs.

4.3.1. Global and Local Types

We extend the definition of global and local types, presented in Chapter 2, Definition 2.2.1 and 2.3.10 respectively, with constructs for handling interrupts.

The inductive definition of G in the upper half of Fig. 4.8 recalls the syntax of *global types* and $\boxed{\text{highlights}}$ the new constructs. Notice that, for readability, in this theory section, we simplify the basic interaction structure $\mathtt{A}{\to}\mathtt{B}:\{l_i.G_i\}_{i\in I}$. More precisely, we do not specify the content of messages (or their type), as it is irrelevant in the presented semantics. Therefore, the interaction $\mathtt{A}{\to}\mathtt{B}:\{l_i.G_i\}_{i\in I}$ stands for a message from A to B containing a label chosen by A between the l_i, where each label l_i corresponds to a specific continuation G_i We remind that $G_1 \mid G_2$ denotes the parallel composition of two protocols; $\mu\mathtt{t}.G$ and $\mathtt{t}$ are, respectively, the recursion construct and the recursion variable, and end is the end of a protocol.

Scopes. We use *scopes* to delimit interruptible blocks inside protocols. In types, scopes are made explicit by the use of scope variables $\mathtt{c}$. We assume there is an infinite set of such variables and that no two variables are the same inside global types. This is crucial as our syntax contains recursion: recursive types are treated as equi-recursive terms, meaning that the lazy unfolding of the types is implicit; thus, when a scope variable appears inside of a recursion loop, it actually stands for an infinite number of fresh variables. We consider that this is an appropriate abstraction of the dynamic scope generation present in the implementation in Section 4.2.2.

Interrupts. Our types feature a new interrupt mechanism by explicit interruptible scopes: we write $\{|G|\}^{\mathtt{c}}\langle l\ \mathtt{by}\ \mathtt{A}\rangle;G'$ to denote a creation of an interruptible block identified by scope $\mathtt{c}$, containing protocol G (called *inner protocol*), that can be interrupt by a message l from A and continued after completion (either normal or exceptional) with protocol G' (called *continuation protocol*). This construct corresponds to the `interruptible` of Scribble, presented in Section 4.1. For the sake of clarity, we suppose there is only one possible interrupt message (from one particular role) for each scope, but extending it to multiple interrupt messages (possibly from different roles) is not difficult. Note that we allow interruptible scopes to be nested.

$$
\begin{array}{rcl}
G & ::= & \texttt{A}\!\rightarrow\!\texttt{B}\!:\!\{l_i.G_i\}_{i\in I} \mid G|G \mid \mid \mu\texttt{t}.G \mid \texttt{t} \mid \texttt{end} \\
& \mid & \boxed{\{\!|G|\!\}^{\texttt{c}}\langle l \text{ by } \texttt{A}\rangle;G' \mid \texttt{Eend}} \\[1em]
T & ::= & \texttt{B}!\{l_i.T_i\}_{i\in I} \mid \texttt{B}?\{l_i.T_i\}_{i\in I} \mid \boxed{T|T} \mid \mu\texttt{t}.T \mid \texttt{t} \mid \texttt{end} \mid \\
& \mid & \boxed{\{\!|T|\!\}^{\texttt{c}} \triangleleft \langle \texttt{B}!l\rangle;T' \mid \{\!|T|\!\}^{\texttt{c}} \triangleright \langle \texttt{B}?l\rangle;T' \mid \texttt{Eend}}
\end{array}
$$

Figure 4.8.: Global and local types

We use `Eend` to denote the exceptional termination of a scope. As a result, `Eend` is not present in a specification and will appear at runtime, to denote that a block has been interrupted.

Local types. The syntax is presented in the lower half of Fig. 4.8 and the new constructs for handling interrupt are $\boxed{\text{highlighted}}$. We remind that $\texttt{A}!\{l_i.T_i\}_{i\in I}$ denotes emitting a message, $\texttt{A}?\{l_i.T_i\}_{i\in I}$ stands for receiving a message, and parallel composition $T \mid T$, recursion and ending constructs serve the same purpose as their global type counterparts.

For scopes, the main difference is that the interruptible operation is divided into two sides, one $\triangleleft$ side for the role which can send an interrupt $\{\!|T|\!\}^{\texttt{c}} \triangleleft \langle \texttt{B}!l\rangle;T'$, and the $\triangleright$ side for the roles which should expect to receive an interrupt message $\{\!|T|\!\}^{\texttt{c}} \triangleright \langle \texttt{B}?l\rangle;T'$.

Well-formedness. As explained in Chapter 2 global types are subject to some well-formedness conditions [HYC08], which constrain the type syntax. We assume every global type G is well-formed according to the conditions from Section 2.1.2, and handling interruptible blocks introduces a unique condition: *uniqueness of scope names*, meaning that in a (equi-recursive) well-formed type, a scope name appears only once in an interruptible construct (note that, as explained above, scope names inside recursions are considered as name generators).

Projection. Fig. 4.9 defines the projection operation $\upharpoonright \texttt{A}$, which, for any participant playing a role `A` in a session G, specifies its local type. We write $\texttt{A} \in G$ when role `A` appears in global type G either as an endpoint in an interaction (sender or receiver) or as role allowed to send an interrupt message in a scope construct.

The projection rules themselves are identical to the ones in [HYC08] except the interrupts: an interaction is projected as a send action `B!` of the sender side, a

We assume $\mathtt{A}$, $\mathtt{B}$ and $\mathtt{A}_0$ are pairwise distinct.

$$\begin{array}{rcl}
(\mathtt{A}{\rightarrow}\mathtt{B}:\{l_i.G_i\}_{i\in I})\uparrow\mathtt{A} & = & \mathtt{B}!\{l_i.(G_i\uparrow\mathtt{A})\}_{i\in I}\\
(\mathtt{A}{\rightarrow}\mathtt{B}:\{l_i.G_i\}_{i\in I})\uparrow\mathtt{B} & = & \mathtt{A}?\{l_i.(G_i\uparrow\mathtt{B})\}_{i\in I}\\
(\mathtt{A}{\rightarrow}\mathtt{B}:\{l_i.G_i\}_{i\in I})\uparrow\mathtt{A}_0 & = & \sqcup(G_1\uparrow\mathtt{A}_0)\\
(\mu\mathtt{t}.G)\uparrow\mathtt{A}_0 & = & \mu\mathtt{t}.(G\uparrow\mathtt{A}_0)\text{ when }\mathtt{A}_0\in G\\
(\mu\mathtt{t}.G)\uparrow\mathtt{A}_0 & = & \mathtt{end}\text{ otherwise}\\
\mathtt{t}\uparrow\mathtt{A}_0 & = & \mathtt{t}\\
\mathtt{end}\uparrow\mathtt{A}_0 & = & \mathtt{end}
\end{array}$$

$$\begin{array}{rcl}
\{\!|G|\!\}^{\mathsf{c}}\langle l\ \mathtt{by}\ \mathtt{B}\rangle;G'\uparrow\mathtt{A} & = & \{\!|G\uparrow\mathtt{A}|\!\}^{\mathsf{c}}\triangleright\langle\mathtt{A}?l\rangle;G'\uparrow\mathtt{A}\\
\{\!|G|\!\}^{\mathsf{c}}\langle l\ \mathtt{by}\ \mathtt{A}\rangle;G'\uparrow\mathtt{A} & = & \{\!|G\uparrow\mathtt{A}|\!\}^{\mathsf{c}}\triangleleft\langle\mathtt{A}!l\rangle;G'\uparrow\mathtt{A}\\
 & & \text{when }\mathtt{A}\in G\\
\{\!|G|\!\}^{\mathsf{c}}\langle l\ \mathtt{by}\ \mathtt{B}\rangle;G'\uparrow\mathtt{A} & = & G'\uparrow\mathtt{A}\\
 & & \text{otherwise}
\end{array}$$

Figure 4.9.: Projection algorithm

receive $\mathtt{B}?$ action on the receiver side and is transparent to other roles (the well-formedness conditions from [HYC08] allows us to do as such by ensuring that every branch is the same to these roles). When projecting types embedded inside a recursion on a role that does not appear inside the body of the recursion, we project on the end type $\mathtt{end}$.

When it comes to interruptible constructs, the projection on role $\mathtt{A}$ works as follows: if role $\mathtt{A}$ is the role responsible for the interrupt, the projection is a $\triangleright$ local type; and if the role $\mathtt{A}$ is not responsible for the interrupt, but appears inside the inner scope, the projection is a $\triangleleft$ local type. If $\mathtt{A}$ does not appear in the inside protocol, the projection ignores the construct and amounts to the projection on the continuation.

As an example, we give projections of our global type. As stated above, we restrict ourselves in the formal section to interruptible scopes accepting only one interrupt message (this can be encoded by two nested scopes), so we omit the timeout interruption. On role $\mathtt{U}$, projection $G_{\mathtt{ResCont}}\uparrow\mathtt{U}$ gives:

$$\boxed{\mathtt{C}!\mathrm{req};\{\!|\mu X.\{\!|\mu Y.\mathtt{A}?\mathrm{data}.Y|\!\}^{\mathsf{c}_2}\triangleright\langle\mathtt{U}?\mathrm{pause}\rangle;\mathtt{A}!\mathrm{resume}.X|\!\}^{\mathsf{c}_1}\triangleright\langle\mathtt{U}?\mathrm{stop}\rangle;}$$

The two nested scopes can be interrupted by $\mathtt{U}$ (hence the $\triangleright$ symbol). Projection $G_{\mathtt{ResCont}}\uparrow\mathtt{A}$ of the same global type on $\mathtt{A}$ would yield:

$$\boxed{\texttt{C?start}; \{|\mu X.\{|\mu Y.\texttt{U!data}.Y|\}^{\texttt{c}_2} \triangleright \langle \texttt{U?pause}\rangle; \texttt{U?resume}.X|\}^{\texttt{c}_1} \triangleleft \langle \texttt{U!stop}\rangle;}$$

As $\texttt{C}$ does not appear inside the loops (and we omit the `timeout` interrupt), the projection $G_{\texttt{ResCont}} \upharpoonright \texttt{C}$ is:

$$\boxed{\texttt{U?req}; \texttt{A!start}; \texttt{end}}$$

4.3.2. Configurations and Semantics

In order to justify our framework, we introduce a semantics for local types through the use of configurations, which are meant to represent the situation of an on-going network of monitored principals.

Environments. We identify multiple sessions – possibly instances of the same global type – taking place simultaneously by a unique *session channel* $(s, k, \ldots)$, mimicking the conversation id in our implementation.

Definition 4.3.1 (Session Environments) *Session Environments are mappings from session channels to local types, i.e.*

$$s_1[\texttt{A}_1] : T_1, \ldots, s_n[\texttt{A}_n] : T_n.$$

The session environments abstract monitored principals. More precisely $s_1[\texttt{A}_1] : T_1$ is the status of participant $\texttt{A}_1$ in session s_1 which is expected to behave as T_1. We use Δ to denote session environments.

Messages and queues. *Standard messages* are explained as follows: $\texttt{c}[\texttt{A},\texttt{B}]\langle l\rangle$ meaning it appears inside scope $\texttt{c}$, is sent from $\texttt{A}$ to $\texttt{B}$ and contains label l. We also annotate messages for interrupts as in $\texttt{c}^{\texttt{I}}[\texttt{A},\texttt{B}]\langle l\rangle$. A *queue* $s[\texttt{A}] : h$ is a sequence of messages waiting to be consumed by a particular role $\texttt{A}$ in session s. Queues are ordered, but we allow permutations of two messages in the same queue if they have different receivers (as in [HYC08, Che13]). For the sake of clarity, we do not describe here the relaxing of conditions on permutability induced by the use of scope (we could allow two messages to the same receiver to be permuted if they are not tagged with the same scope).

$$
\begin{array}{rllll}
E^{\varepsilon} = & [\,] \mid (E^{\varepsilon}|T) \mid (T|E^{\varepsilon}) & & & \\
E^{\mathtt{c}} = & \{\!|E^{\mathtt{c}}|\!\}^{\mathtt{c}'\neq\mathtt{c}} \rhd \langle \mathtt{A}?l\rangle;T' & \mid \{\!|E^{\mathtt{c}}|\!\}^{\mathtt{c}'\neq\mathtt{c}} \lhd \langle \mathtt{A}!l\rangle;T' & \mid \{\!|E^{\varepsilon}|\!\}^{\mathtt{c}} \rhd \langle \mathtt{A}?l\rangle;T' & \\
 & \mid \{\!|E^{\varepsilon}|\!\}^{\mathtt{c}} \lhd \langle \mathtt{A}!l\rangle;T' & \mid \{\!|\mathtt{Eend}|\!\}^{\mathtt{c}'\neq\mathtt{c}} \rhd \langle \mathtt{A}?l\rangle;E^{\mathtt{c}} & \mid \{\!|\mathtt{Eend}|\!\}^{\mathtt{c}'\neq\mathtt{c}} \lhd \langle \mathtt{A}!l\rangle;E^{\mathtt{c}} & \\
 & \mid \{\!|\mathtt{end}|\!\}^{\mathtt{c}'\neq\mathtt{c}} \rhd \langle \mathtt{A}?l\rangle;E^{\mathtt{c}} & \mid \{\!|\mathtt{end}|\!\}^{\mathtt{c}'\neq\mathtt{c}} \lhd \langle \mathtt{A}!l\rangle;E^{\mathtt{c}} & \mid E^{\mathtt{c}}|T \quad \mid \quad T|E^{\mathtt{c}} &
\end{array}
$$

Figure 4.10.: Evaluation contexts

Configurations. $\Delta;\Sigma$ are pairs composed of a session environment and a *transport* Σ which is a collection of queues. Configurations model the behaviour of a network of monitored agents.

We define a reduction semantics for configurations in Fig. 4.11. In order to treat a message with its corresponding scope, we need to remember from which scope the message was sent. To this purpose, we enrich the definition of scopes with ε the empty scope and add a scope annotation on contexts. Evaluation contexts are defined in Fig. 4.10.

Evaluation contexts. The contexts are indexed by scope $\mathtt{c}$; our definition ensures that the evaluation actually happens inside $\mathtt{c}$ (i.e. $\mathtt{c}$ is the innermost scope in which the hole appears). Evaluation can proceed from inside the inner scope of an interruptible (either $\rhd$ or $\lhd$) construct, or from inside the continuation scope of a interruptible, but only when the inner scope has ended (normally or exceptionally).

Semantics. The reduction semantics is defined w.r.t. a *scope environment.*

Definition 4.3.2 (Scope Environment) *Scope environment* $\Gamma = \mathscr{T}, \sqsubset$ *is composed of a* scope table

$$\mathscr{T} ::= \varepsilon \mid \mathtt{c} : \{\mathtt{A}_1, \dots, \mathtt{A}_n\}, \mathscr{T}$$

and a scope order *which is the reflexive and transitive closure of the relation given by:* $\mathtt{c}_1 \sqsubset \mathtt{c}_2$ *whenever a global type contains* $E^{\mathtt{c}_1}[\{\!|G|\!\}^{\mathtt{c}_2}\langle l \;\mathtt{by}\; \mathtt{A}\rangle;G']$. *The scope table keeps track of every participant in a scope and the scope order keeps track of scope nesting (when* $\mathtt{c}_1 \sqsubset \mathtt{c}_2$ *it means that scope* $\mathtt{c}_2$ *is inside scope* $\mathtt{c}_1$*). We note* $\Gamma(\mathtt{c}) = \{\mathtt{A}_1, \dots, \mathtt{A}_n\}$ *whenever* $\Gamma = \mathscr{T}, \sqsubset$ *and* $\mathscr{T}$ *contains* $\mathtt{c} : \{\mathtt{A}_1, \dots, \mathtt{A}_n\}$. *The environment is omitted when not necessary.*

Semantics rules in Fig. 4.11 are as follows: in $\lfloor\text{OUT}\rfloor$, an output from $\mathtt{A}$ to $\mathtt{B}$ appearing inside the scope $\mathtt{c}$ of the type of role $\mathtt{A}$ in session s is played and a message is placed in the queue $s[\mathtt{B}]$, tagged with $\mathtt{c}$. Conversely in $\lfloor\text{IN}\rfloor$, a message

$$\begin{array}{ll}
\lfloor \textsc{Out} \rfloor & s[\mathtt{A}] : E^{\mathtt{c}}[\mathtt{B}!\{l_i.T_i\}]; s[\mathtt{B}] : h \rightarrow \\
& s[\mathtt{A}] : E^{\mathtt{c}}[T_i]; s[\mathtt{B}] : h.\mathtt{c}[\mathtt{A},\mathtt{B}]\langle l_i \rangle \\[1ex]
\lfloor \textsc{In} \rfloor & s[\mathtt{A}] : E^{\mathtt{c}}[\mathtt{B}?\{l_i.T_i\}]; s[\mathtt{A}] : \mathtt{c}[\mathtt{B},\mathtt{A}]\langle l_i \rangle.h \rightarrow \\
& s[\mathtt{A}] : E^{\mathtt{c}}[T_i]; s[\mathtt{A}] : h \\[1ex]
\lfloor \textsc{EOut} \rfloor & s[\mathtt{A}] : E^{\mathtt{c}_0}[\{|T|\}^{\mathtt{c}} \triangleright \langle \mathtt{A}?l \rangle; T']; s[\mathtt{A}_1] : h, \ldots, s[\mathtt{A}_n] : h \rightarrow \\
& s[\mathtt{A}] : E^{\mathtt{c}_0}[\{|\mathtt{Eend}|\}^{\mathtt{c}} \triangleright \langle \mathtt{A}?l \rangle; T']; \\
& s[\mathtt{A}_1] : \mathtt{c}^{\mathtt{I}}[\mathtt{A},\mathtt{A}_1]\langle l \rangle.h, \ldots, s[\mathtt{A}_n] : \mathtt{c}^{\mathtt{I}}[\mathtt{A},\mathtt{A}_n]\langle l \rangle.h \\[1ex]
\lfloor \textsc{EIn} \rfloor & s[\mathtt{A}] : E^{\mathtt{c}_0}[\{|T|\}^{\mathtt{c}} \triangleright \langle \mathtt{B}?l \rangle; T']; s[\mathtt{A}] : \mathtt{c}^{\mathtt{I}}[\mathtt{B},\mathtt{A}]\langle l \rangle.h \rightarrow \\
& s[\mathtt{A}] : E^{\mathtt{c}_0}[\{|\mathtt{Eend}|\}^{\mathtt{c}} \triangleright \langle \mathtt{B}?l \rangle; T']; s[\mathtt{A}] : h \\[1ex]
\lfloor \textsc{Disc} \rfloor & s[\mathtt{A}] : E^{\mathtt{c}_0}[\{|\mathtt{Eend}|\}^{\mathtt{c}} \triangleright \langle \mathtt{B}?l \rangle; T']; s[\mathtt{A}] : \mathtt{c}_1[\mathtt{B},\mathtt{A}]\langle l \rangle.h \rightarrow \\
& s[\mathtt{A}] : E^{\mathtt{c}_0}[\{|\mathtt{Eend}|\}^{\mathtt{c}} \triangleright \langle \mathtt{B}?l \rangle; T']; s[\mathtt{A}] : h \\[1ex]
\lfloor \textsc{EDisc} \rfloor & s[\mathtt{A}] : E^{\mathtt{c}_0}[\{|\mathtt{Eend}|\}^{\mathtt{c}} \triangleright \langle \mathtt{B}?l \rangle; T']; s[\mathtt{A}] : \mathtt{c}_1^{\mathtt{I}}[\mathtt{B},\mathtt{A}]\langle l \rangle.h \rightarrow \\
& s[\mathtt{A}] : E^{\mathtt{c}_0}[\{|\mathtt{Eend}|\}^{\mathtt{c}} \triangleright \langle \mathtt{B}?l \rangle; T']; s[\mathtt{A}] : h \\[1ex]
\lfloor \textsc{Par} \rfloor & \Delta, \Delta_0; \Sigma, \Sigma_0 \rightarrow \Delta', \Delta_0; \Sigma', \Sigma_0 \quad \text{if } \Delta; \Sigma \rightarrow \Delta'; \Sigma'
\end{array}$$

In $\lfloor \textsc{EOut} \rfloor$, we assume $\Gamma(\mathtt{c}) = \{\mathtt{A}, \mathtt{A}_1, \ldots, \mathtt{A}_n\}$; and in $\lfloor \textsc{Disc,EDisc} \rfloor$, we assume $\Gamma \vdash \mathtt{c} \sqsubset \mathtt{c}_1$.

Figure 4.11.: Reduction semantics for a specification

in queue $s[\mathtt{B}]$ can be consumed by $\mathtt{B}$ inside a matching scope. In rule $\lfloor \textsc{EOut} \rfloor$, a type T inside scope $\mathtt{c}$ is interrupted by $\mathtt{A}$, which replaces T by $\mathtt{Eend}$ and places an interrupt message in the queues of each participant of scope $\mathtt{c}$ (we need the table from Γ). Conversely in rule $\lfloor \textsc{EIn} \rfloor$, an interrupt message for scope $\mathtt{c}$ is consumed to exceptionally terminate the type T inside scope $\mathtt{c}$. Rule $\lfloor \textsc{Disc} \rfloor$ discards an incoming message to scope $\mathtt{c}_1$ nested inside scope $\mathtt{c}$ if the latter has already been exceptionally terminated (we need the scope order from Γ). Rule $\lfloor \textsc{EDisc} \rfloor$ performs the same thing for exceptional messages. The three points highlighted in the implementation 4.2.1 are treated in the theory as follows: (1) scopes are explicitly present in messages (2) interrupt messages can be fired at any time from the global queue (see rule $\lfloor \textsc{EIn} \rfloor$); however, using a single-queue system prevents us from giving them priority and (3) scope nesting is handled by the definition of evaluation contexts (which depends on scopes and includes scope nesting in their structures).

Remarks on semantics. Regarding the semantics, we have two remarks. Most of existing theoretical works such as [HYC08] consider session creations, through the use of auxiliary actions. Also the garbage collection can be handled by adding completion annotation to types and additional rules to control broadcasts of special messages: when a participant receives a completion message it can assume its sender is finished, and when every other participants of a scope are finished the whole interrupt construct can be garbage collected. Both these facilities can be integrated into the current semantics.

Next we define configuration correspondence.

Definition 4.3.3 (Configuration Correspondence) *Configuration* Δ, Σ corresponds to *a collection of global types* $G_1, \ldots, G_l$ *whenever* Σ *is empty and* $\Delta = \{G_i \upharpoonright \mathtt{A} \mid \mathtt{A} \in G_l,\ 1 \leq i \leq l\}$. *That is, the environment is a projection of existing well-formed global types.*

We use $\rightarrow^*$ to denote the reflexive-transitive closure of $\rightarrow$.

Definition 4.3.4 (Derivative) *We say that a global type* G' *is a* derivative *of* G *whenever* G' *can be obtained from* G *by progressing in the types. The formal definition is given by taking the reflexive and transitive closure of the* $\rightsquigarrow$*-relation:*

$$\mathtt{A}\rightarrow\mathtt{B}:\{l_i.G_i\}_{i\in I} \rightsquigarrow G_i \qquad \{|G|\}^{\mathsf{c}}\langle l \ \mathtt{by}\ \mathtt{A}\rangle; G_0 \rightsquigarrow \{|\mathtt{Eend}|\}^{\mathsf{c}}\langle l \ \mathtt{by}\ \mathtt{A}\rangle; G_0$$

$$\{|G|\}^{\mathsf{c}}\langle l \ \mathtt{by}\ \mathtt{A}\rangle; G_0 \rightsquigarrow \{|G'|\}^{\mathsf{c}}\langle l \ \mathtt{by}\ \mathtt{A}\rangle; G_0 \ \textit{if}\ G \rightsquigarrow G' \qquad G \mid G_0 \rightsquigarrow G' \mid G_0 \ \textit{if}\ G \rightsquigarrow G'$$

The correctness of our theory is ensured by Theorem 4.3.3. This theorem states the following property:

Definition 4.3.5 (Session Fidelity) *A local enforcement implies global correctness: if a network of monitored agents (modelled as a configuration) corresponds to a collection of well-formed specifications and makes some steps by firing messages, then the network can perform reductions (consuming these messages) and reaches a state that corresponds to a collection of well-formed specifications, obtained from the previous one.*

This property guarantees that the network is always linked to the specification, and proves, that the introduction of interruptible blocks to the syntax and semantics yields a sound theory.

4.3.3. Type Memory

In order to reconstruct a global type from a configuration, we need to remember what was the type inside a scope at the moment it was interrupted; this is done by using type memories. Type memories are a syntactical construct which works as a simple way to remember interrupted types. It is needed because we want, in order to prove session fidelity, to build a well-formed global type from a configuration. If we use the syntax proposed above, some problems would arise: consider a type whose scope $\mathtt{c}$ contains a sequence interaction between $\mathtt{A}$ and $\mathtt{B}$, and suppose the scope is interrupted by $\mathtt{B}$, but the interrupt message is not yet received by $\mathtt{A}$: we cannot obtain a session fidelity result, i.e. we cannot build a well-formed session types from the configuration, the reason being there is no counterpart to the type currently included in scope $\mathtt{c}$ in $\mathtt{A}$, as it as been discarded when $\mathtt{B}$ exceptionally terminated. As a way to remember what $\mathtt{B}$ was supposed to do before the interruption, we use a type memory, that is, a syntactic annotation that remembers the type in $\mathtt{c}$ in $\mathtt{B}$ when the interrupt was raised.

Memories. We use a special annotation, called *memory* to remember what has been discarded by exceptions. The syntax of memory types is the same as the one for standard local types except we add $E^{\mathtt{c}}[\| T \|]$. We define the erase operator $Erase(\cdot)$ which removes memory annotations from types:

$$Erase(s[\mathtt{A}] : E^{\mathtt{c}}[\| T \|]) = s[\mathtt{A}] : E^{\mathtt{c}}[\mathtt{Eend}].$$

We say that a queue *has an ongoing exception on* $\mathtt{c}$, written $\varphi(\Sigma, \mathtt{c})$ whenever Σ contains at least one message $\mathtt{c}_1^{\mathtt{I}}[\mathtt{B}, \mathtt{A}]\langle l \rangle$ and $\mathtt{c} \sqsubset \mathtt{c}_1$.

Intermediate correspondence. From the definition of the correspondence relation between global types and Δ we build the *intermediate correspondence* between global types and configurations Δ, Σ containing types with memories using the following updates:

- For normal (not interruptible) messages in the queue, then

$$\Delta, s[\mathtt{A}] : E^{\mathtt{c}}[T]; \Sigma, s[\mathtt{B}] : h.\mathtt{c}[\mathtt{A}, \mathtt{B}]\langle l_j \rangle$$

  becomes

$$\Delta, s[\mathtt{A}] : E^{\mathtt{c}}[\mathtt{A}!\{l_i.T_i\}]; \Sigma$$

  for some $(T_i)_{i \neq j}$

- For interruptible messages, given participants of $\mathtt{c}$ are $\mathtt{A}, \mathtt{A}_1, \dots, \mathtt{A}_n$, then

$$\begin{aligned}\Delta,\ & s[\mathtt{A}] : E^{\mathtt{c}}[\|T\|] \textstyle\prod_{1\leq i\leq k} s[\mathtt{A}_i] : E_i{}^{\mathtt{c}}[\|T_i\|],\\ & \textstyle\prod_{k+1\leq j\leq n} s[\mathtt{A}_j] : E_i{}^{\mathtt{c}}[T_j];\Sigma,\\ & \textstyle\prod_{k+1\leq j\leq n} s[\mathtt{A}_j] : \mathtt{c}^{\mathtt{I}}[\mathtt{A},\mathtt{A}_j]\langle l\rangle.h_j\end{aligned}$$

  is treated as

$$\Delta, s[\mathtt{A}] : E^{\mathtt{c}}[T], \prod_{1\leq i\leq n} s[\mathtt{A}_i] : E_i{}^{\mathtt{c}}[T_i];\Sigma.$$

This definition ensures first that ongoing outputs are treated as if they were not yet emitted, and that ongoing exceptions are treated as if the exceptions were not yet triggered.

Special semantics for types with memory annotations is obtained by giving memories the same semantics as $\mathtt{Eend}$ w.r.t. contexts and using the following rules for annotated types (replacing $\lfloor$DISC$\rfloor$, $\lfloor$EDISC$\rfloor$ and $\lfloor$EIN$\rfloor$) and presented in Fig. 4.12.

For the rule corresponding to $\lfloor$DISC$\rfloor$, we reduce the memory instead of discarding the message. For rules corresponding to $\lfloor$EIN$\rfloor$ and $\lfloor$EDISC$\rfloor$, in both cases, the semantics distinguish whether the exception corresponding to the message is "ongoing" or not. If it is the case, it means other exception messages for the same scope still exist in queue, thus we annotate the type in the scope (which would have been discarded) as a memory type, in order to remember it. If the exception message was the last one from its scope, then we remove the whole memory for this exception by replacing every corresponding memory (in every type) with $\mathtt{Eend}$.

It is easy to see that Δ,Σ simulates $\mathit{Erase}(\Delta),\Sigma$ and that $\mathit{Erase}()$ preserves the intermediate correspondence w.r.t. $G_1,\dots,G_n$. Thus in the following we will work with memory annotated configurations, which are useful because they remember what local type has been discarded by an exception as long as the type has not been yet discarded for every participant of the scope.

Results. Theorem 4.3.3 states that if a configuration corresponds to $G_1,\dots,G_n$ and makes some reduction steps, we can let it make other steps to reach a configuration that corresponds to some derivatives of $G_1,\dots,G_n$. The intermediate configurations correspond to the situation where messages are exchanged through queues.

$\lfloor$DISC'$\rfloor$ assuming $s[\mathtt{A}] : E^{\mathtt{c}_0}[T]; s[\mathtt{A}] : \mathtt{c}_1[\mathtt{B},\mathtt{A}]\langle l\rangle.h \rightarrow s[\mathtt{A}] : E^{\mathtt{c}_0}[T']; s[\mathtt{A}] : h$

$$s[\mathtt{A}] : E^{\mathtt{c}_0}[\{\!|\,\|T\|\,|\!\}^{\mathtt{c}} \triangleright \langle \mathtt{B}?l\rangle; \_]; s[\mathtt{A}] : \mathtt{c}_1[\mathtt{B},\mathtt{A}]\langle l\rangle.h \rightarrow$$
$$E^{\mathtt{c}_0}[\{\!|\,\|T'\|\,|\!\}^{\mathtt{c}} \triangleright \langle \mathtt{B}?l\rangle; \_]; s[\mathtt{A}] : h$$

$\lfloor$EIN 1$\rfloor$ assuming $\varphi(\Sigma, \mathtt{c})$

$$s[\mathtt{A}] : E^{\mathtt{c}_0}[\{\!|T|\!\}^{\mathtt{c}} \triangleright \langle \mathtt{B}?l\rangle; T']; s[\mathtt{A}] : \mathtt{c}^{\mathtt{I}}[\mathtt{B},\mathtt{A}]\langle l\rangle.h \rightarrow$$
$$E^{\mathtt{c}_0}[\{\!|\,\|T\|\,|\!\}^{\mathtt{c}} \triangleright \langle \mathtt{B}?l\rangle; T']; s[\mathtt{A}] : h$$

$\lfloor$EIN 2$\rfloor$ assuming $\neg\varphi(\Sigma, \mathtt{c})$

$$\prod_{1\le i\le n} s[\mathtt{A}_i] : E_i{}^{\mathtt{c}_i}[\{\!|\,\|T_i\|\,|\!\}^{-} \triangleright \langle \mathtt{B}?l\rangle; \_],$$
$$s[\mathtt{A}] : E^{\mathtt{c}_0}[\{\!|T|\!\}^{\mathtt{c}} \triangleright \langle \mathtt{B}?l\rangle; T']; \Sigma, s[\mathtt{A}] : \mathtt{c}^{\mathtt{I}}[\mathtt{B},\mathtt{A}]\langle l\rangle.h \rightarrow$$
$$\prod_{1\le i\le n} s[\mathtt{A}_i] : E_i{}^{\mathtt{c}_i}[\{\!|\mathrm{Eend}|\!\}^{-} \triangleright \langle \mathtt{B}?l\rangle; \_],$$
$$s[\mathtt{A}] : E^{\mathtt{c}_0}[\{\!|\mathrm{Eend}|\!\}^{\mathtt{c}} \triangleright \langle \mathtt{B}?l\rangle; T']; \Sigma, s[\mathtt{A}] : h$$

$\lfloor$EDISC 1$\rfloor$ assuming $\varphi(\Sigma, k_1)$

$$s[\mathtt{A}] : E^{\mathtt{c}_0}[\{\!|\,\|E^{-}[\{\!|T|\!\}^{k_1} \triangleright \langle \mathtt{B}?l\rangle; T']\|\,|\!\}^{\mathtt{c}} \triangleright \langle \mathtt{B}'?l''\rangle; \_];$$
$$\Sigma, s[\mathtt{A}] : k_1^{\mathtt{I}}[\mathtt{B},\mathtt{A}]\langle l\rangle.h \rightarrow$$
$$s[\mathtt{A}] : E^{\mathtt{c}_0}[\{\!|\,\|E^{-}[\{\!|\,\|T\|\,|\!\}^{k_1} \triangleright \langle \mathtt{B}?l\rangle; T']\|\,|\!\}^{\mathtt{c}} \triangleright \langle \mathtt{B}'?l''\rangle; \_]; \Sigma, s[\mathtt{A}] : h$$

$\lfloor$EDISC 2$\rfloor$ assuming $\neg\varphi(\Sigma, k_1)$

$$s[\mathtt{A}] : E^{\mathtt{c}_0}[\{\!|\,\|E^{\mathtt{c}}[\{\!|T|\!\}^{k_1} \triangleright \langle \mathtt{B}?l\rangle; T']\|\,|\!\}^{\mathtt{c}} \triangleright \langle \mathtt{B}'?l''\rangle; \_],$$
$$\prod_{1\le i\le n} s[\mathtt{A}_i] : E_1{}^{\mathtt{c}}[\{\!|\,\|T_i\|\,|\!\}^{k_1} \triangleright \langle \mathtt{B}?l\rangle; T_i']; \Sigma, s[\mathtt{A}] : \mathtt{c}_i^{\mathtt{I}}[\mathtt{B},\mathtt{A}]\langle l\rangle.h \rightarrow$$
$$\prod_{1\le i\le n} s[\mathtt{A}_i] : E_i{}^{-}[\{\!|\mathrm{Eend}|\!\}^{\mathtt{c}_i} \triangleright \langle \mathtt{B}?l\rangle; T_i],$$
$$s[\mathtt{A}] : E^{\mathtt{c}_0}[\{\!|\,\|E^{\mathtt{c}}[\{\!|\mathrm{Eend}|\!\}^{k_1} \triangleright \langle \mathtt{B}?l\rangle; T']\|\,|\!\}^{\mathtt{c}} \triangleright \langle \mathtt{B}'?l''\rangle; \_]; \Sigma, s[\mathtt{A}] : h$$

where $\varphi(\Sigma, \mathtt{c})$ denotes that there is *an ongoing exception on* $\mathtt{c}_1(\mathtt{c} \sqsubset \mathtt{c}_1)$ in one of the queues in Σ.

Figure 4.12.: Semantics for types with memories

Theorem 4.3.3 (Session fidelity) *If Δ corresponds to $G_1,\ldots,G_n$ and $\Delta,\varepsilon \to^* \Delta',\Sigma'$, there exists $\Delta',\Sigma' \to^* \Delta'',\varepsilon$ such that Δ'' corresponds to $G''_1,\ldots,G''_n$ which is a derivative of $G_1,\ldots,G_n$.*

Proof. We prove that if there is an intermediate correspondence between Δ,Σ and $G_1,\ldots,G_n$ and if $\Delta,\varepsilon \to \Delta',\Sigma'$, then there is an intermediate correspondence Δ'' and $G''_1,\ldots,G''_n$ which is a derivative of $G_1,\ldots,G_n$ by induction on $\to$. A full proof can be found in Appendix A.

4.4. Related Work

Session Types with Exception Handling Our theoretical work is new, as existing works for distributed system do not support at the same time nested interrupt, multiparty protocols and continuations to interruptible blocks. [CHY08, Car09] contained *interactional* exceptions for *binary only* sessions and for web service choreographies. This approach is different as try-catch blocks are built upon session-connections—for a single session, exactly two different behaviours are described—which constrains the shape of the protocols. [CGY10] implements nested exceptions with queues and a central synchronisation through the use levels. Yet, the interruptible construct is blocking w.r.t. continuations and thus not truly asynchronous. [CVB$^+$16] proposes exception blocks with handlers for every message interaction. The set of participants that have to be informed on failure is generated automatically. This leads to the introduction of synchronisation points for every exception block and independently of whether an error has been raised or not. Most importantly, synchronisation is required at every unfolding of a recursion, and thus their model cannot efficiently realise our running example.

Exception handling is common in distributed object-oriented programming [RW95, XRR98]. Composition Actions have been adapted in [TIRL03] to model fault tolerant Web services but these works do not address the same models of protocols. [JP14] presents a copyless message passing model with exceptions and delegation with a common heap. They do not address distributed systems and use transaction to restored to a consistent configuration. Service-oriented calculi (e.g. [BLZ03, LPT07b, VCS08]) include compensation or termination handling, but do not coordinate participant behaviour after an exception is raised. CaSPiS [BBNL08] models binary sessions (and handle nesting with *pipes*); a session can be explicitly terminated by one participant and a termination handler is subsequently activated at the other endpoint.

4.5. Concluding Remarks

We have augmented the dynamic verification framework, presented in Chapter 3 with a new feature for interruptible conversations. In contrast to other exception handling approaches, our design has the advantage of allowing decentralised monitoring infrastructure, which does not require any synchronisation at runtime. We have formalised asynchronous interruptions with conversation scopes, and proved the correctness of our design through the session fidelity theorem. Future work includes the incorporation of more elaborate handling of error cases, allowing insertions of recovery actions.

5. Monitoring Real-time Systems

5.1. Timed Monitored Framework

5.1.1. Background

Recent work [BYY14a] extends Multiparty Session Types (MPSTs) with time, to enable the verification of real-time distributed systems. This extension with time allows specifications (i.e., timed-MPSTs) to express properties on the causalities of interactions, on the carried data types, and on the *times* in which interactions occur. The work in [BYY14a] enables modular static type checking of distributed implementations (i.e., processes in a session π-calculus) against timed-MPSTs.

In this chapter, we apply the theories in [BYY14a] to implement a toolchain for the design of timed specification and dynamic *enforcement* of real-time distributed applications. Enforcement capabilities allow time violations to be verified and *corrected* at runtime.

We build upon the runtime verification framework, presented in Chapter 3 and Chapter 4, by extending it with support for time constraints. There is a substantial difference between the timed and the untimed monitoring framework. We show that ensuring time-consistency over timed global protocols is non trivial. It requires additional verification checks to be performed to ensure the satisfiability of time constraints. To enable the verification of real-time distributed systems we have designed a timed API for Python and enriched the monitor capabilities, providing recovery mechanism for violated time constraints, which were not explored in previous works.

Preserving time-transparency for monitored systems is a new challenge with time. We demonstrate preliminary results on the implications of the dynamic verification overhead over program correctness.

This work is also partially motivated by our collaboration with the Ocean Observatories Initiative (OOI) [OOIa], directed at developing a large-scale cyberinfrastructure for ocean observation. As we have demonstrated in previous chapters (Chapter 3 and Chapter 4), several types of protocols used in the governance of the OOI infrastructure (e.g., users remotely accessing instruments via service agents) can be suitably expressed using Scribble. OOI protocols can be naturally represented as global protocols as they are distributed, typically multiparty, and centered on asynchronous communications via FIFO channels. Time constraints are specified in many OOI use cases, for instance to associate timeouts to requests when resources can be used for fixed amounts of time, or to schedule the execution of services at certain time intervals to reduce the busy wait and minimise energy consumption.

5.1.2. Use Case: Time-bound Log Crawling

To illustrate timed-protocols used in the OOI infrastructure, we present a simple distributed computation of a word count over a set of logs. The timed global protocol, depicted Fig. 5.1 using a message sequence chart (MSC)-like notation, involves three roles: a master `M`, a worker `W` and an aggregator `A`. Each participant has a clock, $x_{\mathtt{M}}$, $x_{\mathtt{W}}$, and $x_{\mathtt{A}}$, respectively, initially set to 0.[1]

1. At the beginning of the session `M` sends `W` a message of type `task` together with a variable of type `log` (i.e., the list of log names to crawl) and a variable of type `string` (i.e., the word to search). The message must be sent by `M` within one second ($x_{\mathtt{M}} < 1$) and received by `W` at time $x_{\mathtt{W}} = 1$. Both `M` and `W` reset their clocks upon sending/receiving the message.
2. The protocol then enters a loop. At each iteration, `W` replies to `M` in exactly 20 seconds with a message of type `result` along with a variable of type `log` (i.e., the logs that have been crawled in the given amount of time) and a variable of type `data` (i.e., the result of the word search). This message is received by `M` at any time satisfying $21.5 < x_{\mathtt{M}} < 22$.
3. A choice is then made locally to `M` at time 22: depending on whether the results are satisfactory or not, the worker chooses to either terminate the session (message of type `end`), or to continue the crawling (messages of type `more`). If `W` chooses `more` all clocks are reset. In both cases the results of the last iteration are forwarded to `A`.
4. This timed protocol allows `M` to wake up at regular intervals (e.g., every 20 seconds) to evaluate the results and decide when to continue or terminate the loop. Otherwise `M` can remain idle (e.g., sleep).

[1]As customary in MPSTs, protocols start synchronously for all roles, hence all clocks start counting at the same time.

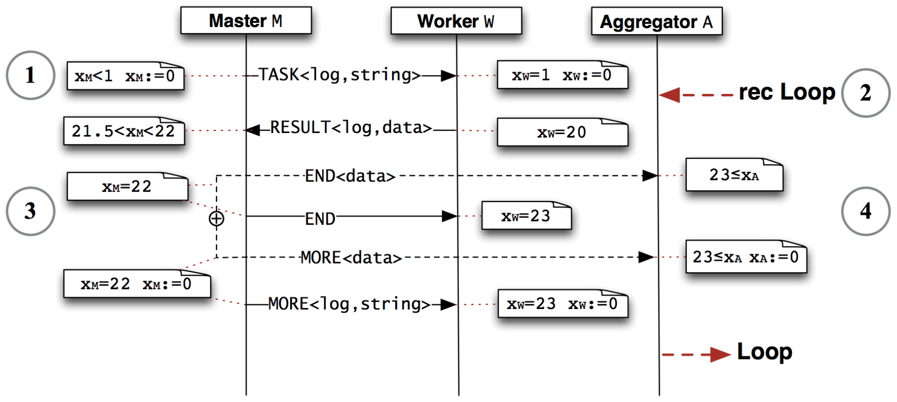


Figure 5.1.: Global protocol for log crawling in Scribble

5.1.3. A Timed Monitor Framework

Building on the theory in [BYY14a] we have extended the toolchain and framework for verification of choreographic sessions in Python, presented in Chapter 3 with time. As it will be illustrated in Section 5.2, our timed extension of the specification language Scribble allows a natural representation of global protocols as the one in Fig. 5.1. Our toolchain supports the top-down development methodology illustrated in Fig. 5.2 and explained below.

- In **step 1**, a global communication is *specified* as a Scribble *timed global protocol*. A timed global protocol defines: (a) the causality among interactions in a session involving two or more roles, (b) the datatypes carried by the messages, and (c) the timing constraints of each interaction. We extend Scribble with the notion of time from [KY06, BYY14a]: each participant owns a clock on which timing constraints can be defined. The clock can be reset many times in a session, and we assume that time flows at the same pace for all clocks and parts of the system.
- In **step 2**, the Scribble toolchain performs a sanity, or *consistency* check on the timed global protocol produced in step 1. In addition to the well-formedness checks, explained in Section 2.1.2, a timed global protocol is also checked against two consistency conditions called *feasibility* and *wait-freedom*. These conditions rule out protocols with unsatisfiable constraints which would intrinsically force well-intentioned principals to either stop

performing actions or continue by violating the protocol's constraints for the role they implement.

- In **step 3**, the Scribble toolchain is used to algorithmically *project* the timed global protocol to *timed local protocols*. Each timed local protocol specifies the actions in a session (and their timing) from the perspective of a single role.
- In **step 4**, principals over a network *implement* one or more, possibly interleaved, timed local protocols. We will call these implementations *timed endpoint programs*. In our prototype implementation, timed local protocols are written in native Python using our in-house developed conversation API, presented in Chapter 3. The API is extended with primitives for handling timeouts and program delays.
- Finally, in **step 5**, the timed endpoint programs are executed. Each endpoint is associated to a dedicated and trusted monitor. A monitor checks that the interactions of the monitored timed endpoint program conform to the implemented timed local protocols. In case of violation, the monitor either throws a time error (error detection mode), or triggers recovery actions to amend the conversation (error prevention/recovery mode).

Note that there is a substantial difference between step 2 and step 5: step 2 checks that the protocol is *satisfiable*, step 5 checks that the protocol is actually *satisfied* by a specific implementation.

Contributions and outline In this chapter we present a toolchain for timed interactions, allowing to

- define timed protocols with Scribble – step 1 Fig. 5.2;
- automatically verify the consistency of these timed protocols (w.r.t. the consistency principle envisaged in [BYY14a]) – step 2 Fig. 5.2;
- automatically project timed protocols onto local timed protocols – step 3 Fig. 5.2; and
- automatically derive runtime monitors from each local timed protocol to check the incoming/outgoing interactions of the corresponding timed endpoint program – step 4 Fig. 5.2.

The implementation presented in this chapter is based on the theory that is given in [BYY14a, BYY14b]. Our contribution with respect to [BYY14a, BYY14b] consists of embedding algorithms and theoretical results into the Scribble toolchain, applying them to run-time monitoring and assessing the practicality of the overall

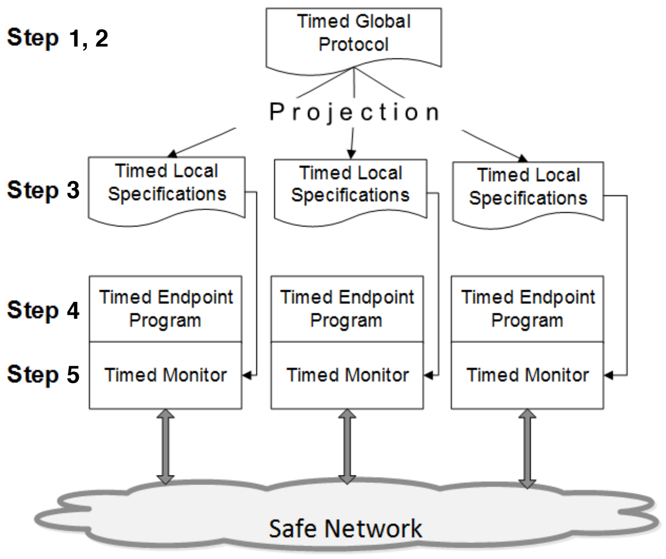


Figure 5.2.: Scribble toolchain framework

approach for timed protocol design and verification.

Concretely, our contributions are: (i) embedding the primitives for *timed protocol specifications* from [BYY14a] into the Scribble toolchain, and giving a concrete implementation into the Scribble toolchain of the algorithms from [BYY14a] for consistency checking and projection; (ii) embedding the calculus for *timed protocol implementations* into a Python API; (iii) exploiting the encoding from timed-MPST into Communicating Timed Automata given in [BYY14a] (and detailed in the corresponding technical report [BYY14b]) to produce run-time monitors for programs implemented using the API from (ii). In (iii), several monitoring *modes* have been provided, with different degrees of 'intervention' of the monitor on the ongoing interactions: from just observing interactions to attempts to fix time mismatches. Moreover, we have assessed the practicality of our implementation in two ways. First, to assess the usability of timed Scribble in more general scenarios than those we developed in OOI, we have gathered a wider (albeit not exhaustive) portfolio of properties of timed distributed protocols from literature, and provided a number of *timed patterns* which demonstrate how these properties can be expressed in Scribble. The implementability of the time-properties expressed by timed Scribble in Python is then demonstrated via examples.

Second, we investigate the concrete effect of time in our monitoring framework via benchmarking. We focus on a property, transparency, which is customary in untimed monitoring frameworks. Transparency (roughly, 'monitors should not affect interactions that are correct') is particularly delicate in timed monitoring frameworks because monitors may introduce time overhead, hence violations, in otherwise correct implementations. We introduce a weaker property called *timed transparency* (roughly, 'monitors should have negligible effect on interactions that are correct') and provide some experimental observation, which interestingly involve the form of the specific protocols being monitored, to estimate whether the monitor overhead will be negligible or not.

In the following sections we will discuss in detail each of the steps of the methodology illustrated in Fig. 5.2.

Section 5.2 (**steps 1 and 2**) presents Scribble timed global and local protocols, which are extensions of Scribble global and local protocols, presented in Chapter 2, and which correspond to timed global and local types in [BYY14a].

Section 5.3 gives a walk-through of the implementation of an algorithm (from [BYY14b]) for checking consistency over timed Scribble protocols, namely the feasibility and wait-freedom properties.

Section 5.4 presents our timed API (**step 3**) in Python based on the calculus with delays in [BYY14a] (a simple timed extension of the π-calculus used to implement timed local types). More precisely, we add two primitives, *timeout* and *delay*. The former interrupts an ongoing computation to meet an approaching deadine, while the latter suspends the execution of a program for a given amount of time.

Section 5.5 discusses runtime enforcement of timed properties (**step 4**). Timed local protocols are automatically encoded into timed automata (using the encoding from timed local types to timed automata from [BYY14a, BYY14b]), which are in turn used by our runtime monitors for error detection. Additional mechanisms for error prevention and recovery are implemented and explained.

Section 5.6 shows benchmark results.

Section 5.7 evaluates the practicality of our approach, and in particular of our timed extension of Scribble. We present a number of temporal patterns drawn from literature together with their Scribble representation.

Section 5.8 discusses related work.

Our prototype implementation is available from [pyt].

5.2. Specifying Timed Protocols with Scribble

5.2.1. Timed Global Protocols

This subsection introduces some preliminary notations needed for timed Scribble protocols. First, we recall from Chapter 2 that in Scribble the interactions between pairs of roles are asynchronous, and can be thought of as being broken down into two actions: the sending action, which adds a message to an unbounded FIFO queue, and the receiving action, which collects a message from the queue. We fix a finite set $\mathscr{R} \subset \mathbb{N}$ of roles ranged over by $\mathtt{A}, \mathtt{B}, \ldots$ which exchange messages in a protocol. Each interaction in a protocol transmits a label (we assume the set of labels is finite and is ranged over by $\mathtt{a}, \mathtt{b}, \mathtt{c}$) and a payload of some sort (e.g., `int`, `bool`, etc.). We assume that each pair of roles, say A and B, can communicate along two dedicated channel: one for messages from A to B and one for messages from B and A.

To model time constraints we fix a set X of real valued clocks (ranged over by $x, x', \ldots$) and let each role in a protocol *own* a finite number of clocks in X, assuming that the sets of clocks owned by each role in $\mathscr{R}$ is a partition of X. We let each sending (resp. receiving) action to be annotated with a time-constraint δ (or simply *constraint*) and a *reset* $\lambda \subseteq X$. An action can be executed only if the associated constraint is satisfied, and the clocks in λ must be reset upon execution of that action. Each role can only reset clocks he/she owns. Note that clocks may have different values at some point in time, since the roles can reset their clocks at different times. However, we assume that time flows at the same pace for all of them (this is a standard assumption e.g. [KY06]). If the time does not flow at the same pace, no global guarantees can be given.

We extend the syntax of Scribble global protocols, given in Definition 2.2.1 in Chapter 2, by adding time constraints on message interactions. The grammar for time constraints and the extended syntax of message interactions is given below.

$$\begin{array}{ll} \delta \ ::= \ \mathtt{true} \mid x < c \mid x = c \mid \neg\delta \mid \delta_1 \wedge \delta_2 & \textit{constraint} \\ [\mathtt{@A} : \delta, \mathtt{reset}(\lambda)][\mathtt{@B} : \delta', \mathtt{reset}(\lambda')]\ \mathtt{a}(\mathtt{T})\ \mathtt{from}\ \mathtt{A}\ \mathtt{to}\ \mathtt{B}; \mathtt{G} & \textit{interaction} \end{array}$$

In the above definition of δ, c denotes a constant value in $\mathbb{Q}^{\geq 0}$. We remind that messages are of the form $\mathtt{a}(\mathtt{T})$ with $\mathtt{a}$ being a label and $\mathtt{T}$ being the constant type of the message exchanged (such as `real`, `bool` and `int`).

```
global protocol WordCount(
  role M, role A, role W){
  [@M: xm<1,reset(xm)]
  [@W: xw=1,reset(xw)]
  task(log,string) from M to W;
    rec Loop{
      [@W: xw=20]
      [@M: 21.5<xm<22]
      result(data) from W to M;
      choice at M{
        [@M: xm=22]
        [@A: 23<=xa,reset(xa)]
        more(data) from M to A;
        [@M: xm=22,reset(xm)]
        [@W: xw=23,reset(xw)]
        more(string) from M to W;
        continue Loop;}
      or {
        [@M: xm=22][@A: 23<=xa]
          end(data) from M to A;
          [@M: xm=22][@W: xw=23]
          end() from M to W; } }
```

```
local protocol WordCount at M(
  role A, role W){

  [@M: xm<1,reset(xm)]
  task(log,string) to W;
  rec Loop{

    [@M: 21.5<xm<22]
    result(data) from W;
    choice at M{

      [@M: xm=22]
      more(data) to A;

      [@M: xm=22, reset(xm)]
      more(log,string) to W;
      continue Loop;}
    or {
      [@M: xm=22]
      end(data) to A;
      [@M: xm=22, reset(xm)]
      end() to W; } }
```

Figure 5.3.: Scribble timed global protocol for 'WordCount' (left) and Projection onto M (right)

Interactions are annotated with constraints and resets, enclosed by square brackets, and explicitly bound to a role. More precisely, the interaction above has two time annotations, $[\texttt{@A}:\delta,\texttt{reset}(\lambda)]$ and $[\texttt{@B}:\delta',\texttt{reset}(\lambda')]$, one for the sender A and one for the receiver B. By $[\texttt{@A}:\delta,\texttt{reset}(\lambda)]$ the message must be sent at a time satisfying the constraint δ. Furthermore, all clocks in λ are reset upon sending the message. We assume that only clocks owned by A occur in δ and λ, and omit $\texttt{reset}(\lambda)$ when $\lambda = \emptyset$. Similarly, $[\texttt{@B}:\delta',\texttt{reset}(\lambda')]$ requires that role B retrieves the message from the queue at any time satisfying δ', and that resets all clocks in λ'. We assume that δ' and λ' are defined only on the clocks owned by B.

Fig. 5.3 (left) shows the global protocol of the example 'WordCount' illustrated in Fig. 5.1 where M is the master, A is the aggregator and W is the worker. For readability, the actual code uses different tabs and newlines (e.g., the interactions are shown in two lines, with the annotations above), and constraints may use formulae (e.g, $21.5 < \texttt{xm} < 22$, and $23 <= \texttt{xa}$) which can be easily derived from the ones given in the grammar for δ.

5.2.2. Formal Semantics of Scribble Timed Global Protocols

The formal semantics of global protocols characterises the desired/correct behaviour of the roles in a multiparty protocol. Semantics is based on the Labelled Transition System (LTS) given in Section 2.2 and adapted from [BYY14a] to support time. The transition labels over which the LTS is defined are extended with a label for time actions, as highlighted below:

$$\ell ::= \mathtt{AB!a(T)} \mid \mathtt{AB?a(T)} \mid \boxed{t}$$

We remind that label $\mathtt{AB!a(T)}$ is for a send action where role $\mathtt{A}$ sends to role $\mathtt{B}$ a message $\mathtt{a(T)}$, and label $\mathtt{AB?a(T)}$ is for a receive action where $\mathtt{B}$ receives (i.e., collects from the queue associated to the appropriate channel) message $\mathtt{a(T)}$ that was previously sent by $\mathtt{A}$. Action $t \in \mathbb{R}^{\geq 0}$ is a time action modelling the elapsing of t time units. We allow $t = 0$ as to account for clock resets. The definition of a subject of an action, modelling the role that has the responsibility of performing that action, is also modified to account for time :

$$subj(\mathtt{AB!a(T)}) = \mathtt{A} \qquad subj(\mathtt{AB?a(T)}) = \mathtt{B} \qquad \boxed{subj(t) = \emptyset}$$

The LTS is defined over states of the form (ν, G) where $\nu : X \mapsto \mathbb{R}^{\geq 0}$ is a clock assignment mapping clocks to values in $\mathbb{R}^{\geq 0}$. We write:

- $\nu + t$ for the assignment obtained replacing $\nu(x)$ with $\nu(x) + t$ in ν for all $x \in X$, namely shifting the time forward of t time units.
- $[\lambda \mapsto 0]\nu$ for the clock assignment obtained by setting the value of all $x \in \lambda$ to 0, namely resetting all clocks in λ.
- $\nu \models_t \delta$ if the constraint obtained by substituting each clock x occurring in δ with $\nu(x)$ is satisfied.

We also extend the syntax for intermediary states, which model a message that has been sent but has not been yet received. The syntax for intermediate states keeps only the time constraints associated with the receiver. For example,

$$[\mathtt{@B} : \delta', \mathtt{reset}(\lambda')]\ \mathtt{a(T)}\ \mathtt{from}\ \mathtt{A}\ \mathtt{to}\ \mathtt{B}; \mathtt{G}$$

describes the state in which message $\mathtt{a(T)}$ has been sent by $\mathtt{A}$ but not yet received by $\mathtt{B}$. The state describes only the time constraints for $\mathtt{B}$. We call *runtime* global protocol a protocol obtained by extending the syntax of timed Scribble with these

$$\frac{\boxed{\nu \models_t \delta \quad \nu' = [\lambda \mapsto 0]\nu}}{(\nu, [@\mathtt{A} : \delta, \lambda][@\mathtt{B} : \delta', \lambda']\ \mathtt{a(T)}\ \mathtt{from}\ \mathtt{A}\ \mathtt{to}\ \mathtt{B}; \mathtt{G}) \xrightarrow{\mathtt{AB!a(T)}} (\nu', [@\mathtt{B} : \delta', \lambda']\ \mathtt{a(T)}\ \mathtt{from}\ \mathtt{A}\ \mathtt{to}\ \mathtt{B}; \mathtt{G})} \lfloor \text{SEND} \rfloor$$

$$\frac{\boxed{\nu \models_t \delta' \quad \nu' = [\lambda' \mapsto 0]\nu}}{(\nu, [@\mathtt{B} : \delta', \lambda']\ \mathtt{a(T)}\ \mathtt{from}\ \mathtt{A}\ \mathtt{to}\ \mathtt{B}; \mathtt{G}) \xrightarrow{\mathtt{AB?a(T)}} (\nu', \mathtt{G})} \lfloor \text{RECV} \rfloor$$

$$\frac{i \in \{1,..,n\} \quad (\nu, \mathtt{G}_i) \xrightarrow{\ell} (\nu', \mathtt{G}'_i) \quad \ell \neq t}{(\nu, \mathtt{choice\ at}\ \mathtt{A}\ \{\mathtt{G}_1\}\ \mathtt{or}\ \ldots\ \mathtt{or}\ \{\mathtt{G}_n\}) \xrightarrow{\ell} (\nu', \mathtt{G}'_i)} \lfloor \text{CHOICE} \rfloor$$

$$\frac{(\nu, \mathtt{G}) \xrightarrow{\ell} (\nu', \mathtt{G}') \qquad \mathtt{A}, \mathtt{B} \notin subj(\ell) \quad \ell \neq t}{(\nu, [@\mathtt{A} : \delta, \lambda][@\mathtt{B} : \delta', \lambda']\ \mathtt{a(T)}\ \mathtt{from}\ \mathtt{A}\ \mathtt{to}\ \mathtt{B}; \mathtt{G}) \xrightarrow{\ell} (\nu', [@\mathtt{A} : \delta, \lambda][@\mathtt{B} : \delta', \lambda']\ \mathtt{a(T)}\ \mathtt{from}\ \mathtt{A}\ \mathtt{to}\ \mathtt{B}; \mathtt{G}')} \lfloor \text{ASYNC1} \rfloor$$

$$\frac{(\nu, \mathtt{G}) \xrightarrow{\ell} (\nu', \mathtt{G}') \qquad \mathtt{B} \notin subj(\ell) \quad \ell \neq t}{(\nu, [@\mathtt{B} : \delta', \lambda']\ \mathtt{a(T)}\ \mathtt{from}\ \mathtt{A}\ \mathtt{to}\ \mathtt{B}; \mathtt{G}) \xrightarrow{\ell} (\nu', [@\mathtt{B} : \delta', \lambda']\ \mathtt{a(T)}\ \mathtt{from}\ \mathtt{A}\ \mathtt{to}\ \mathtt{B}; \mathtt{G}')} \lfloor \text{ASYNC2} \rfloor$$

$$\frac{(\nu, \mathtt{G}[\mathtt{rec}\ \mathtt{L}\ \mathtt{G}/\mathtt{continue}\ \mathtt{L}]) \xrightarrow{\ell} (\nu', \mathtt{G}')}{(\nu, \mathtt{rec}\ \mathtt{L}\ \mathtt{G}) \xrightarrow{\ell} (\nu', \mathtt{G}')} \lfloor \text{REC} \rfloor$$

$$\boxed{\frac{\nu' = \nu + t \quad \nu' \models^* \mathtt{rdy}(\mathtt{G})}{(\nu, \mathtt{G}) \xrightarrow{t} (\nu', \mathtt{G})} \lfloor \text{TIME} \rfloor}$$

Figure 5.4.: Labelled transitions for timed global protocols.

intermediary states.

The transition rules are given in Fig. 5.4. We have highlighted the major differences w.r.t. the transition rules for global protocols (without time), presented in Fig. 2.4 in Chapter 2. Rule $\lfloor \text{SEND} \rfloor$ models a sending action: given that the constraint δ associated with the send action of A is satisfied by the current clock assignment ν (i.e., $\nu \models_t \delta$), $\lfloor \text{SEND} \rfloor$ produces a label AB!a(T). The sending action yields a state in which: the clock assignment ν' is obtained from ν by resetting all the clocks in λ, and the global protocol is in an intermediate state where a(T) has been sent but not received by B.

Rule $\lfloor \text{RECEIVE} \rfloor$ models the dual receive action, from the intermediate state to its continuation G. Rule $\lfloor \text{CHOICE} \rfloor$ continues the execution of the protocol as the continuation of one of the branches, given that the action is not a time action (all time actions are all handled by one rule, $\lfloor$TIME$\rfloor$, which will be discussed later). Rules $\lfloor$ASYNC1$\rfloor$ and $\lfloor$ASYNC2$\rfloor$ caters for asynchrony and distribution, as explained in the Scribble semantics in Chapter 2. Rule $\lfloor$ASYNC1$\rfloor$ allows any order of execution for two send actions that *are not causally related* (e.g have different subjects), Rule $\lfloor$ASYNC2$\rfloor$ is similar but caters for intermediate states, Rule $\lfloor$REC$\rfloor$ is standard and unfolds recursive protocols.

Before explaining the semantics of time passing, modelled by $\lfloor$TIME$\rfloor$, it is necessary to recall a few notions from [BYY14a]: the *ready actions* of a protocol and the *satisfiability of ready actions*. The ready actions of a runtime Scribble global protocol $\mathtt{G}$ are the actions that have no causal relationship with other actions that occur earlier, syntactically, in $\mathtt{G}$ hence could be immediately executed. For example, consider the partial protocols given below:

$$\begin{array}{l}[\mathtt{@A}:\delta_{\mathtt{A}},\mathtt{reset}(\lambda_{\mathtt{A}})][\mathtt{@B}:\delta_{\mathtt{B}},\mathtt{reset}(\lambda_{\mathtt{B}})]\ \mathtt{a(T)}\ \mathtt{from}\ \mathtt{A}\ \mathtt{to}\ \mathtt{B};\\ [\mathtt{@C}:\delta_{\mathtt{C}},\mathtt{reset}(\lambda_{\mathtt{C}})][\mathtt{@D}:\delta_{\mathtt{D}},\mathtt{reset}(\lambda_{\mathtt{D}})]\ \mathtt{a(T)}\ \mathtt{from}\ \mathtt{C}\ \mathtt{to}\ \mathtt{D}\end{array} \tag{5.1}$$

$$\begin{array}{l}[\mathtt{@B}:\delta_{\mathtt{B}},\mathtt{reset}(\lambda_{\mathtt{B}})]\ \mathtt{a(T)}\ \mathtt{from}\ \mathtt{A}\ \mathtt{to}\ \mathtt{B};\\ [\mathtt{@C}:\delta_{\mathtt{C}},\mathtt{reset}(\lambda_{\mathtt{C}})][\mathtt{@D}:\delta_{\mathtt{D}},\mathtt{reset}(\lambda_{\mathtt{D}})]\ \mathtt{a(T)}\ \mathtt{from}\ \mathtt{C}\ \mathtt{to}\ \mathtt{D}\end{array} \tag{5.2}$$

The ready actions of (5.1) are $\mathtt{AB!a(T)}$ and $\mathtt{CD!a(T)}$ and the ready interactions of (5.2) are $\mathtt{AB?a(T)}$ and $\mathtt{CD!a(T)}$. We write $\mathtt{rdy}(\mathtt{G})$ for the ready constraints of $\mathtt{G}$, namely the set of constraints associated to the ready actions of $\mathtt{G}$. For example, if $\mathtt{G}$ is the protocol in (5.1) then $\mathtt{rdy}(\mathtt{G}) = \{\{\delta_{\mathtt{A}}\},\{\delta_{\mathtt{C}}\}\}$, and if $\mathtt{G}$ is the protocol in (5.2) then $\mathtt{rdy}(\mathtt{G}) = \{\{\delta_{\mathtt{B}}\},\{\delta_{\mathtt{C}}\}\}$. The ready constraint set is a *set of sets of constraints*, to cater for the choice construct. For example, in the protocol in (5.3), $\mathtt{rdy}(\mathtt{G}) = \{\{\delta\},\{\delta_{C1},\delta_{C2}\}\}$, meaning that there are two ready actions: one (i.e., the receive action of $\mathtt{B}$) that can be executed given that δ is satisfied, and one (i.e., one of the two possible send action from $\mathtt{C}$) that can be executed given that either δ_{C1} or δ_{C2} is satisfied.

$$\begin{array}{rl}\mathtt{G} = & [\mathtt{@B}:\delta,\mathtt{reset}(\lambda)]\ \mathtt{a(T)}\ \mathtt{from}\ \mathtt{A}\ \mathtt{to}\ \mathtt{B};\\ & \mathtt{choice\ at}\ \mathtt{C}\\ & \quad\{[\mathtt{@C}:\delta_{\mathtt{C1}},\mathtt{reset}(\lambda_{\mathtt{C1}})][\mathtt{@D}:\delta_{\mathtt{D1}},\mathtt{reset}(\lambda_{\mathtt{D1}})]\ \mathtt{a_1(T)}\ \mathtt{from}\ \mathtt{C}\ \mathtt{to}\ \mathtt{D}\}\ \mathtt{or}\\ & \quad\{[\mathtt{@C}:\delta_{\mathtt{C2}},\mathtt{reset}(\lambda_{\mathtt{C2}})][\mathtt{@D}:\delta_{\mathtt{D2}},\mathtt{reset}(\lambda_{\mathtt{D2}})]\ \mathtt{a_2(T)}\ \mathtt{from}\ \mathtt{C}\ \mathtt{to}\ \mathtt{D}\};\end{array} \tag{5.3}$$

The aim of the semantics of global protocols is to determine the desirable executions allowed by a protocol. To this aim, we want its semantics of time passing to allow each participant to be able to execute one of the possible next ready action of which this participant is subject/responsible. In other words, we want to prevent the elapsing of time intervals that would invalidate the constraint of the action prescribed by a ready interaction. Consider again the protocol in (5.1) and assume that $\delta_A = x \leq 20$, $\delta_C = y < 30$ with $\nu(x) = \nu(y) = 0$: we want to prevent the LTS to perform time actions that invalidate any of the two ready actions. Therefore, we will only allow time steps than do not let time elapse of more than $\mathtt{min}(20, 30)$ time units as they would 'not give time' to A and B to perform their ready action. In the case of the protocol in (5.3) we want that: (1) the only possible action of $\mathtt{B}$ (i.e., δ), **and** (2) at least one of the choice options of $\mathtt{C}$ (i.e., δ_{C1} **or** δ_{C2}) remain satisfiable at present or some times in the future. In Definition 5.2.1 we recall satisfiability of ready interactions, written $\nu \models^* \mathtt{rdy}(\mathtt{G})$, from [BYY14a].

Definition 5.2.1 (Satisfiability of ready interactions) *We write $\nu \models^* \mathtt{rdy}(\mathtt{G})$ when the constraints of all ready actions of $\mathtt{G}$ are satisfiable under ν or sometimes in the future. Formally, $\nu \models^* \mathtt{rdy}(\mathtt{G})$ iff $\forall \{\delta_i\}_{i \in I} \in \mathtt{rdy}(\mathtt{G})\, \exists t \geq 0,\ j \in I.\ \nu + t \models_t \delta_j$.*

Finally, we explain rule $\lfloor\textsc{time}\rfloor$. The rule allows time to elapse of t time units yielding a new clock assignment $\nu' = \nu + t$ (recall that $\nu + t$ shifts all clocks in ν of t) *as long as* this does not invalidate any of the ready actions of $\mathtt{G}$ (i.e., $\nu' \models^* \mathtt{rdy}(\mathtt{G})$). Note that the idle process always allows time to elapse, namely $(\nu, \mathtt{end}) \xrightarrow{t} (\nu + t, \mathtt{end})$ for any t by rule $\lfloor\textsc{time}\rfloor$, as $\mathtt{end}$ has no ready actions.

5.2.3. Time Properties of Global Protocols

The theoretical framework in [BYY14a] sets two consistency conditions on timed global types: *feasibility* and *wait-freedom*. Feasibility (first introduced in [AFK87]) requires that for each partial execution allowed by a specification there is a correct complete one, namely that the protocol will not get stuck due to some unsatisfiable constraint. Wait-freedom requires that if senders respect their time constraints then receivers never have to wait for their messages. These conditions rule out protocols which may intrinsically lead to undesirable scenarios, as shown by the examples in Fig. 5.5.

The protocol $\mathtt{pro1}$ in Fig. 5.5 (left) violates feasibility since it allows $\mathtt{A}$ to send $\mathtt{msg}$ at any time satisfying $\mathtt{xa} < 10$, for instance at time 8, for which then $\mathtt{B}$ has no means to satisfy constraint $\mathtt{xb} < 5$ for the corresponding receive action. The

```
global protocol pro1(
  role A, role B){
  [@A: xa<10][@B: xb<5]
  M1(string) from A to B;
    ...}
```

```
global protocol pro2(
  role A, role B){
  [@A: x<10][@B: x<20]
  M1(string) from A to B;
  [@B: x<20][@A: true]
  M2(string) from B to A;
    ...}
```

Figure 5.5.: Protocol which violates feasibility (left) and wait-freedom (right)

protocol `pro2` in Fig. 5.5 (right) violates wait-freedom as it allows `A` to send a message when `B` is already waiting for it. Assume `B` to be implemented by a timed endpoint program that receives `M1` at time 5, and then engages in a time-consuming activity for 14 seconds before sending `M2`. The plan of `B` conforms to the timed behaviour prescribed by `pro2` for `B`. If, however, we compose the timed endpoint program described before with an implementation of `A` that sends `M1` at time 8, we have that `B` will not find the message in the queue at the expected time 5, will 'be delayed' with respect to his planned timing, and may end up violating the contract at a later action.

In [BYY14a] these conditions (feasibility and wait-freedom) yield progress for *statically* validated timed programs, which ensures that the next available action will be executed in the specified time-range. In the case of *dynamically* verified programs against Scribble specifications (as in our timed framework) progress is difficult to attain. In fact, monitors cannot force timed endpoint programs to send the remaining messages in a protocol when these programs are deliberately refusing or are not able to do so (e.g., their machine is down). Ensuring that conversations are established on feasible and wait-free protocols is, however, a good practice as it prevents violations of time-progress that are induced by the protocol itself.

We implemented a syntactic checker for global protocols of feasibility and wait-freedom, which we will explain in detail in Section 5.3.

5.2.4. Timed Local Protocols

The syntax for Scribble timed local protocols extends the syntax of Scribble local protocols by time constraints on the message send and receive actions. The new constructs are given below:

$$\begin{array}{ll}[@\mathtt{A}:\delta,\mathtt{reset}(\lambda)]\ \mathtt{a}(\mathtt{T})\ \mathtt{to}\ \mathtt{B};\mathtt{T} & \textit{send}\\ [@\mathtt{A}:\delta,\mathtt{reset}(\lambda)]\ \mathtt{a}(\mathtt{T})\ \mathtt{from}\ \mathtt{B};\mathtt{T} & \textit{receive}\end{array}$$

Local protocol $[@\mathtt{A}:\delta,\mathtt{reset}(\lambda)]\ \mathtt{a}(\mathtt{T})\ \mathtt{to}\ \mathtt{B};\mathtt{T}$ models a send action from A to B; the dual local protocol is $[@\mathtt{A}:\delta,\mathtt{reset}(\lambda)]\ \mathtt{a}(\mathtt{T})\ \mathtt{from}\ \mathtt{B};\mathtt{T}$ that models a receive action of A from B.

The definition of projection of timed global protocols to timed local protocols differs from the definition 2.3.12 from Chapter 2 in the following way. Each $[@\mathtt{A}:\delta,\mathtt{reset}(\lambda)][@\mathtt{B}:\delta',\mathtt{reset}(\lambda')]$ in a global protocol interaction is projected onto the sender (resp. receiver) by keeping only the time annotations (contraints and resets) associated to the send action $[@\mathtt{A}:\delta,\mathtt{reset}(\lambda)]$ (resp. the receive action $[@\mathtt{A}:\delta',\mathtt{reset}(\lambda')]$). More precisely, the projection of $\mathtt{G}$ onto $\mathtt{A}\in\mathscr{P}(\mathtt{G})$, written $\mathtt{G}\downarrow_{\mathtt{A}}$, is defined as:

$$\begin{array}{lll}[@\mathtt{A}:\delta,\mathtt{reset}(\lambda)]\ \mathtt{a}(\mathtt{T})\ \mathtt{to}\ \mathtt{B};(\mathtt{G}'\downarrow_{\mathtt{A}}) & \text{if } \mathtt{G}= & [@\mathtt{A}:\delta,\mathtt{reset}(\lambda)][@\mathtt{B}:\delta',\mathtt{reset}(\lambda')]\\ & & \mathtt{a}(\mathtt{T})\ \mathtt{from}\ \mathtt{A}\ \mathtt{to}\ \mathtt{B};\mathtt{G}'\\ [@\mathtt{A}:\delta',\mathtt{reset}(\lambda')]\ \mathtt{a}(\mathtt{T})\ \mathtt{from}\ \mathtt{B};(\mathtt{G}'\downarrow_{\mathtt{A}}) & \text{if } \mathtt{G}= & [@\mathtt{B}:\delta,\mathtt{reset}(\lambda)][@\mathtt{A}:\delta',\mathtt{reset}(\lambda')]\\ & & \mathtt{a}(\mathtt{T})\ \mathtt{from}\ \mathtt{B}\ \mathtt{to}\ \mathtt{A};\mathtt{G}'\end{array}$$

The projection for the rest of the Scribble constructs stays unchanged.

For example, the projection onto M of interaction

$$\begin{array}{l}[@\mathtt{M}:\mathtt{xm}<1,\mathtt{reset}(\mathtt{xm})][@\mathtt{W}:\mathtt{xw}=1,\mathtt{reset}(\mathtt{xw})]\\ \mathtt{task}(\mathtt{log},\mathtt{string})\ \mathtt{from}\ \mathtt{M}\ \mathtt{to}\ \mathtt{W};\mathtt{G}\end{array}$$

yields

$$[@\mathtt{M}:\mathtt{xm}<1,\mathtt{reset}(\mathtt{xm})]\ \mathtt{task}(\mathtt{log},\mathtt{string})\ \mathtt{to}\ \mathtt{W};(\mathtt{G}\downarrow_{\mathtt{M}})$$

Fig. 5.3 (right) presents the local protocol resulting from projecting on M the global protocol 'WordCount' Fig. 5.3 (left).

Formal semantics of local protocols. The LTS for local protocols is defined by the rules in Fig. 5.6, which use the same labels of the global semantics in Fig. 5.4. We $\boxed{\text{highlight}}$ the main differences w.r.t the local semantic for Scribble protocols (without time). given in Fig. 2.5 in Chapter 2. Note that, for readability, in Fig. 5.6 we omit the explicit reset in the syntax of time constraints and we write only the clock variable (e.g we write λ instead of $\mathtt{reset}(\lambda)$). The rules

$$\frac{\boxed{\nu \models_t \delta \quad \nu' = [\lambda \mapsto 0]\nu}}{(\nu, [@\mathtt{A} : \delta, \mathtt{reset}(\lambda)]\ \mathtt{a(T)}\ \mathtt{to}\ \mathtt{B;T}) \xrightarrow{\mathtt{AB!a(T)}} (\nu', \mathtt{T})} \quad \lfloor\text{SEND}\rfloor$$

$$\frac{\boxed{\nu \models_t \delta \quad \nu' = [\lambda \mapsto 0]\nu}}{(\nu, [@A : \delta, \mathtt{reset}(\lambda)]\ \mathtt{a(T)}\ \mathtt{from}\ \mathtt{B;T}) \xrightarrow{\mathtt{AB?a(T)}} (\nu', \mathtt{T})} \quad \lfloor\text{RECV}\rfloor$$

$$\frac{i \in \{1,..,n\} \quad (\nu, \mathtt{T}_i) \xrightarrow{\ell} (\nu', \mathtt{T}'_i) \quad l \neq t}{(\nu, \mathtt{choice\ at\ A}\ \{\mathtt{T}_1\}\ \mathtt{or} \ldots \{\mathtt{T}_n\}) \xrightarrow{\ell} (\nu', \mathtt{T}'_i)} \quad \lfloor\text{CHOICE}\rfloor$$

$$\frac{(\nu, \mathtt{T}[\mathtt{rec\ L\ T}/\mathtt{continue\ L}]) \xrightarrow{l} (\nu', \mathtt{T}')}{(\nu, \mathtt{rec\ L\ T}) \xrightarrow{l} (\nu', \mathtt{T})} \quad \lfloor\text{REC}\rfloor$$

$$\boxed{\frac{\nu' = \nu + t \quad \nu' \models^* \mathtt{rdy(T)}}{(\nu, \mathtt{T}) \xrightarrow{t} (\nu', \mathtt{T})} \quad \lfloor\text{TIME}\rfloor}$$

Figure 5.6.: Labelled transitions for local protocols.

⌊SEND⌋, ⌊RECV⌋, ⌊CHOICE⌋, ⌊REC⌋ are similar to the respective rules for global protocols. We do not need rules for modelling asynchrony as each participant is assumed to be single threaded. For the time passing rule ⌊TIME⌋ the constraints of the ready action of T must be still satisfiable after t in ν except T has only one ready action.

Formal semantics of configurations. The LTS in Fig. 5.6 describes the behaviour of each single role in isolation. In the rest of this section we give the semantics of systems resulting from the composition of Scribble local protocols and the communication channels. Given a set of roles $\{1,\ldots,n\}$ we define configurations $(\mathtt{T_1},\ldots,\mathtt{T_n},\vec{w})$ where $\vec{w} ::= \{w_{ij}\}_{i\neq j\in\{1,\ldots,n\}}$ are unidirectional, possibly empty (denoted by ε), unbounded FIFO queues with elements of the form $\mathtt{a(T)}$.

Definition 5.2.2 (Semantics of configurations) *The LTS of* $(\mathtt{T_1},\ldots,\mathtt{T_n},\vec{w})$ *is defined as follows, with* ν *being the overriding union (i.e.,* $\oplus_{i\in\{1,\ldots,n\}}\nu_i$*) of the clock assignments* ν_i *of the roles:*

$(\nu,(\mathtt{T_1},\ldots,\mathtt{T_n},\vec{w})) \xrightarrow{\ell} (\nu',(\mathtt{T'_1},\ldots,\mathtt{T'_n},\vec{w}'))$ *iff:*

$$(1)\ (\nu_{\mathtt{A}},\mathtt{T_B}) \xrightarrow{\mathtt{AB!a(T)}} (\nu'_{\mathtt{A}},\mathtt{T'_B}) \wedge w'_{\mathtt{AB}} = w_{\mathtt{AB}}\cdot \mathtt{a(T)} \wedge (ij \neq \mathtt{AB} \Rightarrow w_{ij} = w'_{ij} \wedge \mathtt{T_i} = \mathtt{T'_i})$$

$$(2)\ (\nu_{\mathtt{B}},\mathtt{T_B}) \xrightarrow{\mathtt{AB?a(T)}} (\nu'_{\mathtt{B}},\mathtt{T'_B}) \wedge \mathtt{a(T)}\cdot w'_{\mathtt{AB}} = w_{\mathtt{AB}} \wedge (ij \neq \mathtt{AB} \Rightarrow w_{ij} = w'_{ij} \wedge \mathtt{T_j} = \mathtt{T'_j})$$

$$(3)\ \forall \mathtt{A} \neq \mathtt{B}.\ (\nu_{\mathtt{A}},\mathtt{T_A}) \xrightarrow{t} (\nu_{\mathtt{A}}+t,\mathtt{T_A}) \quad\wedge\quad w_{\mathtt{AB}} = w'_{\mathtt{AB}} \quad\wedge$$

$$(\ \mathtt{T_A} = \mathtt{choice\ at\ A}\ \{[\mathtt{@B} : \delta_i, \mathtt{reset}(\lambda_i)]\ \mathtt{a}_i(\mathtt{T}_i)\ \mathtt{from\ A}; \mathtt{T}_i\}_{i\in\{1,..,m\}}$$

$$\wedge\ w_{\mathtt{AB}} = w''_{\mathtt{AB}}\cdot \mathtt{a}_s(\mathtt{T}_s)\ \ \textit{for some } w''_{\mathtt{AB}} \textit{ and } s \in \{1,..,m\} \quad\Rightarrow\quad \nu + t \models^* \delta_s$$

with $\mathtt{A},\mathtt{B},i,j \in \{1,\dots,n\}$.

We explain only (3) since (1) and (2) are the same as the definition of configurations of local protocols (without time), given in Definition 2.2.3 from Chapter 2, except the addition of clock updates. In (3) the configuration can make a time action t given that all participants can let time elapse for t time units (i.e., according to their ready actions). The additional condition in (3) is needed to cater for scenarios where the protocol being executed is a choice and the message has been sent but not yet received i.e., it is stored in a queue. In this case, the sender has already decided the choice to be taken and it is necessary that time actions preserve the receiver's ability of receiving that specific message (i.e., the constraints of the choice selected by the sender must not to be made unsatisfiable by time actions). The condition in (3) is needed to guarantee this, as the receiver (and the definition of receiver's ready actions) do not take into account the state of the queues. Consider for example the protocol below:

```
choice at B {
  [@A : x < 10][@B : x < 10] a₁(T₁) from B to A
  [@A : x < 20][@B : x < 20] a₂(T₂) from B to A}
```

If the message $\mathtt{a_1(T_1)}$ from $\mathtt{B}$ is already in the queue, namely the first branch has already been chosen. By default, the $\lfloor\text{TIME}\rfloor$ rule dictates that the time can elapse as far as there are satisfiable ready actions (e.g., either receiving $\mathtt{a_1(T_1)}$ or $\mathtt{a_2(T_2)}$). However, if we allow time to elapse of $t = 19$ even when the sender has already chosen the first branch, the receiver will not be able to act as prescribed by the protocol (i.e., will not be able to receive the sent message $\mathtt{a_1(T_1)}$ at the time prescribed by the corresponding constraint). In fact, we must not allow $\lfloor\text{TIME}\rfloor$ to let elapse for more than 10 unit, otherwise the constraint of the first branch will become unsatisfiable. The constraints in (3) ensure that if there is a message in the queue, the time constraints associated with this messages should be satisfiable after $\nu + t$.

5.2.5. Correspondence of Global and Local Protocols

First, we recall some preliminary notations from Chapter 2. We write $TR(\mathtt{G})$ for the set of visible traces obtained by reducing $\mathtt{G}$ under the initial assignment v_0. Similarly for $TR(\mathtt{T}_1, \ldots, \mathtt{T}_n, \vec{\varepsilon})$ where $\vec{\varepsilon}$ is the vector of empty queues. We denote trace equivalence by $\approx$. Theorem 5.2.3 gives the correspondence between the traces produced by a global protocol $\mathtt{G}$ and those produced by the configuration that consists of the composition of the projections of $\mathtt{G}$ onto $\mathscr{P}(\mathtt{G})$.

Theorem 5.2.3 (Soundness of projection) *Let* $\mathtt{G}$ *be a Scribble timed global protocol and* $\{\mathtt{T}_1, \ldots, \mathtt{T}_n\} = \{\mathtt{G}\downarrow_{\mathtt{A}}\}_{\mathtt{A} \in \mathscr{P}(\mathtt{G})}$ *be the set of its projections, then*

$$\mathtt{G} \approx (\mathtt{T}_1, \ldots, \mathtt{T}_n, \vec{\varepsilon}).$$

Theorem 5.2.3 directly follows by: (i) the correspondence between timed Scribble global protocols and timed-MPSTs global types, the correspondence is trivial given the correspondence between Scribble global protocols and MPSTs global types, proven in Chapter 2, since the rules for handling time in the semantics of timed global protocols are precisely adapted from the rules for handling time in the semantics of timed-MPSTs, given in [BYY14a]; (ii) trace equivalence between global types and configuration of projected global types (Theorem 3.3 in [BYY14a]); (iii) the correspondence between configurations of timed-MPSTs local types and configurations of timed Scribble local protocols, which similarly follows by the proven correspondence in Chapter 2 and the precise translation of semantic rules, w.r.t time, from [BYY14a]. Correspondence is important as it ensures that the composition of processes, each implementing some timed local protocol, will behave as prescribed by the original timed global specification.

5.3. Checking Feasibility and Wait-Freedom

In this section we present a syntactic checker for two time properties on Scribble global protocols: feasibility and wait-freedom. These properties have been first introduced in [BYY14a] to guarantee time-progress for statically validated programs. Time-progress ensures that a validated program is guaranteed to proceed until the completion of all activities of the protocols it implements. A protocol is *feasible* if every partial execution can be extended to a terminated session (i.e., the execution never gets stuck because of an unsatisfiable clock constraint). A proto-

Algorithm 1 Algorithm for checking feasibility and wait-freedom in global protocols

```
Require: G = build_time_dependency_graph(AST)                          ▷ Step 1
1: for node in G do
2:     for (constraints, resets) in dfs(root, node) do                 ▷ Step 2
3:         constraints, resets = convert_absolute(constraints, resets)  ▷ Step 3
4:         formula = build_z3_formula(constraints, resets)             ▷ Step 4
5:         result = formula.is_satisfiable()
6:         if not result then
7:             return False
8: return True
```

col is *wait-free* when, in all its distributed implementations, a receiver checking the queue never has to wait for the message. We have discussed in § 5.2.3 that in our framework, which differently from [BYY14a] is based on dynamic verification, these properties do not guarantee time-progress. Feasibility and wait-freedom are nevertheless important to rule out protocols that may intrinsically lead to violations. Feasibility provides a sanity check on whether the constraints specified by a protocol are satisfiable by some implementation. Wait-freedom rules out timed global protocols whose distributed implementation, built *modularly* and in conformance with each projected timed local protocol, may actually 'get late' with respect to the timing prescribed by the original timed global protocol. In addition, wait-freedom rules out busy waiting, which is critical in areas such as sensor networks, where one of the main sources of energy inefficiency is listening on idle channels [YHE02].

Algorithm 1 presents the high level steps for checking the above mentioned properties on global Scribble protocols. In short, the algorithm is based on the following steps:

- **Step 1:** Building a *time dependency graph*, a directed acyclic graph that models the actual causal dependencies between the actions in a protocol. The time dependency graph is built by traversing the Abstract Syntax Tree (AST) of a global Scribble protocol. The nodes of the graph model the actions of the timed protocol, annotated with constraints and resets, and the edges model the causal dependencies between actions.
- **Step 2:** For each node `n`, do a depth-first-search traversal (dfs) of the graph and find the set of paths from the initial node to `n` .
- **Step 3:** To model the range of 'absolute' times (i.e., *virtual time*) in which

the state represented by n can be reached, we convert (shift) each time constraint with respect to the clock resets of the preceding nodes, yielding a *virtual time constraint* $\overline{\delta}_{\mathtt{n}}$ for each node n. Then the constraint for each node n is modified to account for the restrictions imposed by the virtual time constraints of all preceding nodes (constructed in Step 2) of n.

- **Step 4:** To check the feasibility property we build a logic formula to check that the modified constraint of a node n is satisfiable given the restrictions from previous nodes. For wait-freedom, we check that all solutions for a *receiving node* occur at the same or at a later time with respect to any solutions allowed by constraints of preceding nodes. We feed these formulas to an SMT solver to check if the formulas are satisfiable.

The algorithm terminates either when the maximum number of states are reached, that is all nodes are checked (Line 8 of Algorithm 1), or when a node is found for which the feasibility/wait-freedom formulae is unsatisfiable (Line 6-7 of Algorithm 1). The logic used for evaluating node satisfiability (Step 4) forms a subset of Presburger arithmetic, which is decidable (see e.g., [BYY14b]) therefore the termination of the algorithm depends on the termination of traversing all nodes. To ensure the graph traversing always terminates we impose a condition on time constraints used in recursion bodies (see [BYY14b]). More precisely, we consider only protocols that are infinitely satisfiable:

Definition 5.3.1 (Infinitely satisfiable) *A global protocol is infinitely satisfiable if: (1) constraints in recursion bodies have no resets, no equalities, no upper bounds or (2) all participants reset at each iteration.*

The above condition ensures that checking the one-time unfolding of each recursion is sufficient to ensure satisfiability of all successive unfoldings. This is the reason the graph traversing in Step 2 is done on acyclic graph, that is built from a global protocol after one-time unfolding of all recursions and subsequently replacing `continue` t with `end` in the body of the protocol. Therefore, the algorithm always terminates as it considers finite paths for a finite number of nodes.

The complexity of the algorithm 1 is mostly affected by creating the dependency graph (linear on the size of the protocol), on enumerating all paths from the root(s) to a node in the graph (polynomial on the size of the graph) and on the satisfiability of Presburger formulae. In general, the asymptotic running-time complexity of satisfiability of Presburger formulae is doubly exponential. SMT solvers use various techniques to reduce the running time and space, thus the precise complexity

```
rec Loop {                                   n1: msg1(string) from A to B;
    n1: msg1(string) from A to B;            n2: msg2(string) from A to C;
    n2: msg2(string) from A to C;            n3: msg1(string) from A to B;
continue Loop;                               n4: msg2(string) from A to C;
  }
```

Figure 5.7.: An example of a recursion protocol (left) and its one-time unfolding (right).

is solver specific.

In the rest of this section we explain each step of Algorithm 1 in detail.

5.3.1. Step 1: Build the time dependency graph.

As shown in Algorithm 1 to make dependencies between constraints explicit when checking feasibility and wait-freedom, we create a representation of global timed protocols as time dependency graphs (hereafter dependency graphs). A dependency graph of a global protocol G is a pair (N,E) where N denotes the set of nodes and E denotes the set of edges.

To construct the time dependency graph we annotate in G each syntactic occurrence of subterms of the form

$$[\texttt{@A}:\delta,\texttt{reset}(\lambda)][\texttt{@B}:\delta',\texttt{reset}(\lambda')]\texttt{a}(\texttt{T})\ \texttt{from A to B};\texttt{G}'$$

with a node name (denoted by $\texttt{n}_1,\texttt{n}_2,\ldots$).

The nodes of the time graph have the form of $(\texttt{n},!,\texttt{A},\delta,\lambda)$ which represents a sending action from participant A and $(\texttt{n},?,\texttt{B},\delta',\lambda')$ which represents a receiving action at participant B.

The time dependency graph represents all causal dependencies in a protocol. These dependencies are not explicitly captured as the syntactic order of interactions in a Scribble protocol does not necessarily imply a causal dependency between actions. We illustrate message causalities on an example. Consider the protocol in Figure 5.7 (left), where we have annotated the interaction nodes (for simplicity, we have omitted time constraints):

As customary in multiparty session types, the receive action of msg1 by B and of msg2 by C could happen in any order due to asynchrony of communications, despite appearing in a specific syntactic order in pro. However, *some* causal dependencies are indeed enforced by the syntax of protocol pro above:

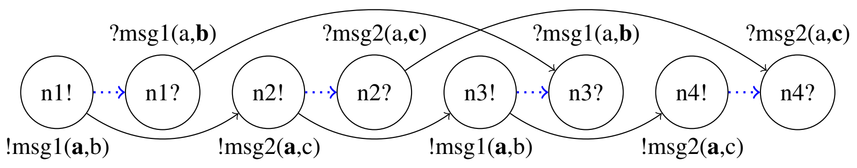


Figure 5.8.: Dependency graph for the protocol in Figure 5.7

1. (I/O dependencies) The receive action of `msg1` by `B` causally depends from the send action of `msg1` by `A`. Thus, two nodes that are generated from the same interaction type should be connected by an edge, e.g $((\mathtt{n}_1, !, \mathtt{A}),$ $(\mathtt{n}_1, ?, \mathtt{B})) \in E$ and $((\mathtt{n}_2, !, \mathtt{A}),$ $(\mathtt{n}_2, ?, \mathtt{C})) \in E$;
2. (Single participant dependencies) The initial sending of `msg1` must occur before the sending of `msg2`, by `A` (namely the syntactic order of actions in a protocol corresponds to an actual causal dependency when actions are performed by the same role). Thus, two consecutive interactions for the same participant should be connected by an edge, e.g. $((\mathtt{n}_1, !, \mathtt{A}), (\mathtt{n}_2, !, \mathtt{A})) \in E$;
3. (Recursion dependencies) Sending of `msg2` by `A` happens both before and after sending of `msg1` by `A` due to recursion. Following [BYY14a, BYY14b], we consider the class of global protocol that are infinitely satisfiable (Definition 5.3.1). Infinite satisfiability allows us to check for feasibility and wait-freedom by checking the one-time unfolding (unfolding all recursions in the protocol *only once*), as it guarantees that the same properties will then hold also in successive unfoldings. Figure 5.7 (right) shows the one-time unfolding for the protocol in Figure 5.7 (left). Recursion dependencies, as the ones between `msg1` and `msg2`, are captured in the dependency graph right away, by considering the one-time unfolding of the global protocol.

All the above dependencies are represented in the dependency graph. The dependency graph for the protocol in Figure 5.7 is shown in Figure 5.8 and the dependency graph for our running example is shown in Figure 5.9. The causal dependencies between sending and their corresponding receive actions are represented by dotted edges, the ones reflecting the syntactic order of actions performed by the same role by **solid** edges, and recursion dependencies are directly captured (as single participant dependencies) by considering the one-time unfolding. As can be seen in the figures, an interaction in Scribble is represented as two nodes in the time dependency graph: a sending node (!(sender, receiver)) and a receiving node (?(sender, receiver)) connected by a directed edge from the sending to the receiving node. For readability, we keep both sender and receiver roles in the node representation but we **bold** the role that corresponds to $\texttt{subject}(\texttt{n})$, we also omit time constraints.

Formally, dependency graphs are defined inductively (Definition 5.3.2) using two functions, $\texttt{nodes}(\texttt{G})$ and $\texttt{edges}(\texttt{G})$, which given $\texttt{G}$ return the set of nodes and edges, respectively, of its dependency graph by checking all the sub terms of $\texttt{G}$. The function $\texttt{edges}(\texttt{G})$ uses an auxiliary mappings $\texttt{D}$, initially empty. The mapping $\texttt{D} : \mathscr{P}(\texttt{G}) \to \texttt{nodes}(\texttt{G})$ from participants to nodes is for constructing edges of type (2). We use the information in $\texttt{D}$ to return the node for the last action of each participant in $\mathscr{P}(\texttt{G})$.

Definition 5.3.2 (Dependency graph) *Let* $\texttt{G}_0$ *be a global protocol and* $\texttt{G}$ *be its one-time unfolding. The dependency graph of* $\texttt{G}_0$ *is a pair* (N, E) *where* $N = \texttt{nodes}(\texttt{G})$ *and* $E = \texttt{edges}(\texttt{G})$*. The functions* $\texttt{nodes}(\texttt{G})$ *and* $\texttt{edges}(\texttt{G})$ *are defined as follows:*

1. *if* $\texttt{G}'$ *is* $\texttt{n} : [@\texttt{A} : \delta, \texttt{reset}(\lambda)][@\texttt{B} : \delta', \texttt{reset}(\lambda')]\ \texttt{a}(\texttt{T})\ \texttt{from A to B};\texttt{G}''$ *then:*
   - $\texttt{nodes}(\texttt{G}') = (\texttt{n}_1) \cup (\texttt{n}_2) \cup \texttt{nodes}(\texttt{G}'')$ *where* $\texttt{n}_1 = (\texttt{n}, !, \texttt{A}, \texttt{B}, \delta, \lambda)$ *and* $\texttt{n}_2 = (\texttt{n}, ?, \texttt{A}, \texttt{B}, \delta', \lambda')$
   - $\texttt{edges}(\texttt{G}', \texttt{D}) = (\texttt{n}_1, \texttt{n}_2) \cup (\texttt{D}(\texttt{A}), \texttt{n}_1) \cup (\texttt{D}(\texttt{B}), \texttt{n}_2) \cup \texttt{edges}(\texttt{G}'', \texttt{D}[\texttt{A} \mapsto \texttt{n}_1, \texttt{B} \mapsto \texttt{n}_2])$
2. *if* $\texttt{G}'$ *is* $\texttt{choice at A}\ \texttt{G}_1\ \texttt{or}\ \ldots \texttt{or}\ \texttt{G}_\texttt{n}$ *then*
   - $\texttt{nodes}(\texttt{G}') = \bigcup_{i \in [1\ldots n]} \texttt{nodes}(\texttt{G}_\texttt{i})$
   - $\texttt{edges}(\texttt{G}', \texttt{D}) = \bigcup_{i \in [1\ldots n]} \texttt{edges}(\texttt{G}_\texttt{i}, \texttt{D})$

Every time a node $\texttt{n}$ is added to N, the mapping is updated $\texttt{D}[\texttt{A} \mapsto \texttt{n}_1, \texttt{B} \mapsto \texttt{n}_2]$ and an edge between $\texttt{n}$ and the $\texttt{D}(\texttt{subject}(\texttt{n}))$ is created, where $\texttt{subject}(\texttt{n})$ is the role element (the third or the forth element respectively) in $\texttt{n}$. Note that the choice itself does not introduce any new edges or nodes. The causality between the actions is preserved in the mapping $\texttt{D}$. All choice branches are traversed with the same mapping $\texttt{D}$.

5.3.2. Step 2: Collecting all paths to a node

After the time dependency graph is built, for each node $\texttt{n}$ we collect all paths from the initial node to $\texttt{n}$. Each path to $\texttt{n}$ represents a possible execution to the protocol

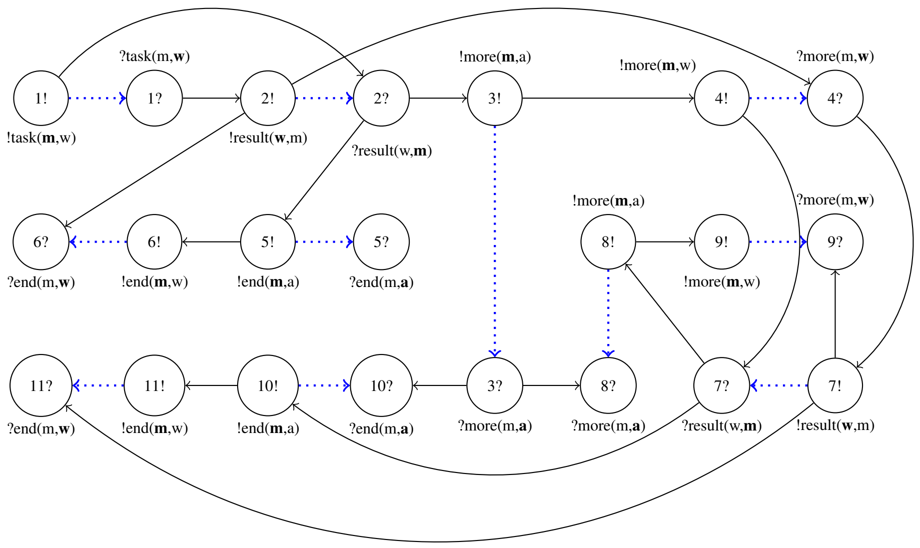


Figure 5.9.: Time dependency graph for the WordCount protocol

leading to state n, and captures time constraints and resets that should be taken into account when building the formula to check if the constraint annotating n satisfies feasibility and wait-freedom. Noticeably, the number of paths to n is finite since the dependency graph is acyclic. For collecting all paths, we traverse the graph using a standard modification of depth-first-search with backtracking.

5.3.3. Step 3: Virtual time constraints

The value of a clock variable in a Scribble constraint represents the time elapsed since the previous resets of that clock variable. To reason on properties of a constraint, such as its satisfiability, we need to consider all possible time scenarios (e.g., all possible 'pasts') that lead to the execution point in which that constraint must be evaluated. We will do this relying on the notion of *virtual time* (that is the time elapsed since the beginning of the session) as opposed to relative time (time elapses since the previous clock reset) and, more precisely, of *virtual time constraint* of a node. The virtual time constraint $\overline{\delta}_{\mathtt{n}}$ of a node n of a time dependency graph models the possible virtual times in which the execution flow may reach a node n. The virtual time constraint is calculated by taking into account constraints of past actions and by 'shifting' time to account for previous resets for the clock.

We illustrate Step 3 under the simplifying assumption that each role owns exactly one clock, hence each constrain in a Scribble global protocol has exactly one free clock variable. As remarked in [BYY14b], the extension to multiple clocks is considerably more verbose but does not pose qualitative challenges.

We denote by $fc(\delta)$ the free clock variable of δ, and by $\vec{x}$ a finite vector of clock variables. Assume $\mathtt{p}$ owns a clock, we denote with $x_{\mathtt{n}}$ the state of the clock owned by $\mathtt{p}$ in node $\mathtt{n}$, where $\mathtt{subject}(\mathtt{n}) = \mathtt{p}$. We call $x_{\mathtt{n}}$ a *clock state*. A clock state uniquely identifies (e.g., in a virtual time constraint) the value of the clock of $\mathtt{p}$ in a node $\mathtt{n}$. Below we give the extended definition of δ, which we will use to reason on *clock states*.

Definition 5.3.3 (Extended clock constraints (adapted from [BYY14b])) *The set of extended clock constraints δ is:*

$$\begin{array}{lcl} \delta & ::= & \mathtt{true} \mid x > e \mid x = e \mid \neg\delta \mid \delta_1 \wedge \delta_2 \\ e & ::= & x \mid c \mid e + e \end{array}$$

To calculate the virtual time in a node, we need to know how many times the clock variable, say x, of that node, say $\mathtt{n}$, has been reset since the beginning of the session; to this aim, we define a function R (Definition 5.3.4) that takes $\mathtt{n}$ and returns the sum of clock states in which x has been reset since the beginning of the session. On the basis of Definition 5.3.4, we obtain time constraints as to represent constraints on virtual (absolute) time as opposed to constraints on relative time. More precisely, we shift a time constraint for a node $\mathtt{n}$ with a clock $x_{\mathtt{n}}$ by adding all previous states of the clock x where x has been reset. The definition of a sum of a time constraint and a clock variable is given in Definition 5.3.5. For example, consider the partial protocol given below.

```
[@A: xa>2; reset(xa)]
n1: msg1() from A to B;
[@A: xa<1]
n2: msg2() from A to C;
```

Assume $\mathtt{n}_{1!}$ and $\mathtt{n}_{2!}$ are the sending nodes for the first and the second interaction in the dependency graph for the protocol. Then $R(\mathtt{n}_{2!}) = \{xa_1\}$ and the clock xa_2, in constraint $xa_2 < 1$, represents a relative time for the clock at $\mathtt{A}$, e.g xa_2 measures the elapsed time between the previous reset of clock xa, which happens at time xa_1 and the current time. If we modify the constraint $xa_2 < 1$ to $xa_2 < 1 + R(\mathtt{n}_{2!})$ (e.g

$xa_2 < 1 + xa_1$) then xa_2 represents a value on the absolute timeline of xa.

Next we give the definitions of a reset function R and of $\delta + x$, adapted from [BYY14b].

Definition 5.3.4 (Reset function (adapted from [BYY14b])) *Let* $\mathtt{n}$ *be a node in the time dependency graph* N *of* $\mathtt{G}$*. We denote by* $\prec_{\mathtt{G}}$ *the transitive closure of* $<_{\mathtt{G}}$*.*

The reset set $\overline{R}$ *for a node* $\mathtt{n}$ *contains all clock states where the clock has been reset.*

$$\overline{R}(\mathtt{n}) = \{x_{\mathtt{n}'} \mid \mathtt{n}' \in N \wedge \mathtt{n}' \prec_{\mathtt{G}} \mathtt{n} \ \wedge \ \mathtt{subject}(\mathtt{n}) = \mathtt{subject}(\mathtt{n}') \ \wedge \ \mathtt{resInfo}(\mathtt{n}')\}$$

where $x'_{\mathtt{n}}$ *is the state of the clock at node* $\mathtt{n}'$ *and* $\mathtt{resInfo}(\mathtt{n}')$ *is* $\mathtt{true}$ *if the clock has been reset at node* $\mathtt{n}'$*.*

The reset function of $\mathtt{n}$ *is defined as a sum of all clock states in* $\overline{R}$*.*

$$R(\mathtt{n}) = \textstyle\sum_{\mathtt{n}' \in \overline{R}(\mathtt{n})} x_{\mathtt{n}'}$$

Definition 5.3.5 (Sum of a constraint with a clock (adapted from [BYY14b])) *The sum of* δ *with a clock* x*, written* $\delta + x$ *is defined as follows:*

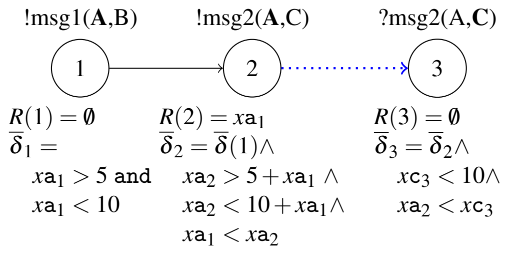
$$\begin{aligned}
&\mathtt{true} + x = \mathtt{true} \\
&(x' \ \mathtt{rop}\ e) + x = \begin{cases} x' \ \mathtt{rop}\ (e + x) & (f(x, x') = \mathtt{true}) \\ x' \ \mathtt{rop}\ e & (f(x, x') = \mathtt{false}) \end{cases} \quad \textit{with } (\mathtt{rop} \in \{>, =\}) \\
&(\neg\delta) + x = \neg(\delta + x) \\
&(\delta \wedge \delta') + x = (\delta + x) \wedge (\delta' + x) \\
&\textit{with} \quad f(x_{\mathtt{n}}, x_{\mathtt{n}'}) = \begin{cases} \mathtt{true} & (\mathtt{subject}(\mathtt{n}) = \mathtt{subject}(\mathtt{n}')) \\ \mathtt{false} & (\textit{otherwise}) \end{cases}
\end{aligned}$$

The sum of δ *with a vector of clocks is* $\delta + \vec{x} = (\delta + x_0) + \vec{x}'$ *with* $\vec{x} = x_0 + \vec{x}'$*.*

Next we give the formal definition of a virtual time constraint of a node. We calculate the time scenario of a node recursively as to account for all previously occurred constraints.

Definition 5.3.6 (Virtual time constraint (adapted from [BYY14b])) *Let* $\mathtt{n}$ *be a node in the time dependency graph* N *of* $\mathtt{G}$ *with* $\mathtt{const}(\mathtt{n}) = \delta$*, and* $M = \{\mathtt{n}' \mid \mathtt{n}' \in$

```
[@A: xa>5 and xa<10;
reset(xa)]
msg1() from A to B;
[@A: xa>5 and xa<10]
[@C: xc<10]
msg2() from A to C;
```



!msg1(**A**,B)
!msg2(**A**,C)
?msg2(A,**C**)
1
2
3
$R(1)=\emptyset$
$\overline{\delta}_1 =$
$x\mathtt{a}_1 > 5$ and
$x\mathtt{a}_1 < 10$
$R(2)=x\mathtt{a}_1$
$\overline{\delta}_2 = \overline{\delta}(1)\wedge$
$x\mathtt{a}_2 > 5 + x\mathtt{a}_1 \wedge$
$x\mathtt{a}_2 < 10 + x\mathtt{a}_1 \wedge$
$x\mathtt{a}_1 < x\mathtt{a}_2$
$R(3)=\emptyset$
$\overline{\delta}_3 = \overline{\delta}_2\wedge$
$x\mathtt{c}_3 < 10\wedge$
$x\mathtt{a}_2 < x\mathtt{c}_3$

Figure 5.10.: An example of clock resets. A Scribble protocol with resets (a) and its dependency time graph with dependency resets (b)

N and $\mathtt{n}' <_{\mathtt{G}} \mathtt{n}\}$ be the set of all nodes directly preceding $\mathtt{n}$, The virtual time constraint $\overline{\delta}_{\mathtt{n}}$ for a node $\mathtt{n}$ is

$$\overline{\delta}_{\mathtt{n}} = \begin{cases} (\delta[x_{\mathtt{n}}/x] + R(\mathtt{n})) \bigwedge\limits_{\mathtt{n}'\in M} (\overline{\delta}_{\mathtt{n}'} \wedge x_{\mathtt{n}'} \leq x_{\mathtt{n}}) & \textit{if } fc(\delta) = x \\ \bigwedge\limits_{\mathtt{n}'\in M} \overline{\delta}_{\mathtt{n}'} & \textit{if } fc(\delta) = \emptyset \end{cases}$$

Note that if a node does not have incoming edges $\bigwedge\limits_{\mathtt{n}'\in M} \overline{\delta}_{\mathtt{n}'}$ and $\bigwedge\limits_{\mathtt{n}'\in M} (\overline{\delta}_{\mathtt{n}'} \wedge x_{\mathtt{n}'} \leq x_{\mathtt{n}})$ are trivially true.

In the above definition we assume that a node has an unique name $\mathtt{n}$. We first rename the clock in the constraint of $\mathtt{n}$ (e.g., $\delta[x_{\mathtt{n}}/x]$) by using this unique node name, e.g. $x_{\mathtt{n}}$. In this way, $x_{\mathtt{n}}$ uniquely identifies the state of the clock in that node. We shift the constraint to represent a constraint on the virtual timeline, e.g $\delta[x_{\mathtt{n}}/x] + R(\mathtt{n})$. Then we take into account all past constraints $\overline{\delta}_{\mathtt{n}'}$. We model the linear flow of time in causally related actions by requiring that the virtual time in node $\mathtt{n}$ is after the virtual time in preceding nodes, e.g $x_{\mathtt{n}'} \leq x_{\mathtt{n}}$.

We illustrate Step 3 by using the example in Figure 5.10 (a). Figure 5.10 (b) illustrates both the reset function and the virtual time constraints for nodes 1, 2 and 3. In the fragment of the time dependency graph for this example, shown in Figure 5.10 (b), the receive action of `C` causally depends from the send action of `msg2` by `A` (node 2) which, in turn, depends from the send action of `msg1` from `A` (node `1`). The constraints of node `1` and node `2` alone do not give sufficient information about the virtual time in node `3`. For instance, due to the reset of $x\mathtt{a}$ in the first action, the action in node `3` will be ready to be executed at an absolute time which is shifted by 5-to-10 time units. Namely, the virtual time for `3` is greater than 10 (i.e., the sum of the lower bounds of the constraints in node `1` and 2, that is $5+5$) and smaller than 20 (i.e., the sum of the upper bounds of the constraints in node 1 and 2, that is $10+10$). Since $x\mathtt{c}$ is never reset it will have, in node 3, some value greater than 10 and smaller than 20. This means that node 3 will be reached when the constraint $x\mathtt{c} < 10$ cannot be satisfied. This example shows that it is necessary to take resets into account when evaluating the satisfiability of a constraint in a node. In this specific example, $x\mathtt{a}_1 \leq x\mathtt{a}_2$ is redundant as it is implicitly expressed by the 'shift' in $\overline{\delta}_2$. In general, however, as no resets may have occurred, this inequality is needed.

5.3.4. Step 4: Construct and check feasibility and wait-freedom formula

Up to this step, wait-freedom and feasibility share the same approach for (a) collecting all nodes occurring before a given node $\mathtt{n}$ (i.e., the nodes that are occurring before $\mathtt{n}$ in some one-unfolding path), and (b) modifying the constraints with renaming and, and (c) calculating the virtual time constraint $\overline{\delta}_{\mathtt{n}}$, a constraint giving a range of possible absolute times in which the execution of the protocol could reach the state modelled by $\mathtt{n}$. The difference in checking the two properties lies in the formulas for the satisfiability property for that node. These formulas are then solved by using an off-the-shelf SMT solver, Z3 [Z3C], which is a tool for checking satisfiability of logical formulas over one or more theories.

In this subsection we first remind the two properties, feasibility and wait-freedom, and give their representation as logical formulas, as shown in [BYY14b]. Then we present the formulas in a Z3 input format.

Feasibility of a global protocol requires the satisfiability of each constraint in the protocol, in every possible scenario that satisfies the previously occurred constraints. We define protocol feasibility in terms of node satisfiability. i.e a protocol is feasible if all nodes in its time dependency graph are satisfiable. For a node to be satisfiable its constraint must be satisfiable given all restrictions posed by constraints of preceding nodes. Next we give the formal definition of a satisfiable node, adapted from [BYY14b]. [2]

Definition 5.3.7 (Satisfiable node (adapted from [BYY14b])) *Let* $\mathtt{n}$ *be a node of the (unfolding) time graph of a global protocol* $\mathtt{G}$*,* $\mathtt{const}(\mathtt{n}) = \delta$*,* $\mathtt{fn}(\delta) = x$*,*

[2] In [BYY14b] the definition is given in the general case for multiple clocks per role, in this article we assume one clock per role.

and $M = \{\mathtt{n}' \mid \mathtt{n}' \in N$ *and* $\mathtt{n}' <_{\mathtt{G}} \mathtt{n}\}$. *Node* $\mathtt{n}$ *is* satisfiable *if*

$$\bigwedge_{\mathtt{n}'\in M} \overline{\delta}_{\mathtt{n}'} \supset \exists x_{\mathtt{n}}.((\delta[x_{\mathtt{n}}/x] + R(\mathtt{n})) \bigwedge_{\mathtt{n}'\in M} x_{\mathtt{n}'} \leq x_{\mathtt{n}})$$

The formulae expresses that given all previously occurred virtual time constraints $\bigwedge_{\mathtt{n}'\in M} \overline{\delta}_{\mathtt{n}'}$ there is a solution $x_{\mathtt{n}}$ of the shifted time constraint for the given node $(\delta[x_{\mathtt{n}}/x] + R(\mathtt{n}))$ and this solution is not in the past (i.e., before the current virtual time), thus it is strictly bigger than any solutions $x_{\mathtt{n}'}$ of previously occurred constraints.

The translation of the logical formulae to a format, accepted by the Z3 checker is straightforward. The formulae is given below, where $\bigcup_{\mathtt{n}'\in M} \mathtt{fn}(\overline{\delta}_{\mathtt{n}'}) = \{x_1, \ldots, x_m\}$, $\delta_{\mathtt{n}} = \bigwedge_{\mathtt{n}'\in M} \overline{\delta}_{\mathtt{n}'}$ and $\delta_{x_{\mathtt{n}}} = \delta[x_{\mathtt{n}}/x]$

$$\text{ForAll}(x_1, \ldots, x_m, \text{Implies}(\delta_{\mathtt{n}}, \text{Exists}(x_{\mathtt{n}}, (\delta_{x_{\mathtt{n}}} + R(\mathtt{n}))\mathtt{and}\ x_1 \leq x_{\mathtt{n}}\ \mathtt{and}\ \ldots\ \mathtt{and}\ x_m \leq x_{\mathtt{n}})))$$

Wait-freedom requires that all solutions of time constraints of receivers in a global protocol do not precede solutions posed by previously occurred constraints. Next we give the formal definition of a wait-free node.

Definition 5.3.8 (Wait-free node (adapted from [BYY14b])) *Let* $\mathtt{n}_?$ *be a receiving node of the (unfolding) time graph of* $\mathtt{G}$ *with* $\mathtt{const}(\mathtt{n}_?) = \delta$, *and* $\mathtt{fn}(\delta) = x$, *and* $M = \{\mathtt{n}' \mid \mathtt{n}' \in N$ *and* $\mathtt{n}' <_{\mathtt{G}} \mathtt{n}\}$. *We say that* $\mathtt{n}_?$ *is* wait-free *if*

$$(\bigwedge_{\mathtt{n}'\in M} \overline{\delta}_{\mathtt{n}'}) \wedge (\delta[x_{\mathtt{n}}/x] + R(\mathtt{n}) \supset \bigwedge_{\mathtt{n}'\in M} x_{\mathtt{n}'} \leq x_{\mathtt{n}}$$

The formula states that all solutions $x_{\mathtt{n}'}$ of virtual time constraints of nodes preceding node $\mathtt{n}_?$, e.g solutions of $\bigwedge_{\mathtt{n}'\in M} \overline{\delta}_{\mathtt{n}'}$, and all solutions $x_{\mathtt{n}}$ of the time constraint of $\mathtt{n}_?$, e.g solutions of $\delta[x_{\mathtt{n}}/x] + R(\mathtt{n})$, are such that the virtual time at node $\mathtt{n}_?$ is after the time posed by the previously occurred constraints, e.g $\bigwedge_{\mathtt{n}'\in M} x_{\mathtt{n}'} \leq x_{\mathtt{n}}$.

The formula, as accepted in Z3, is given below.

$$\text{ForAll}(x_1, \ldots, x_m, x_{\mathtt{n}}, \text{Implies}(\delta_{\mathtt{n}_?}\ \mathtt{and}\ \delta_{x_{\mathtt{n}}}, x_1 \leq x_{\mathtt{n}}\ \mathtt{and}\ \ldots\ \mathtt{and}\ x_m \leq x_{\mathtt{n}}))$$

where $x_1, \ldots, x_m$ are the free variables of $\delta_{\mathtt{n}_?}$, and $x_{\mathtt{n}}$ is the (renamed) clock of $\mathtt{n}_?$.

A node $\mathtt{n}$ such that $\mathtt{subject}(\mathtt{n}) = \emptyset$ or $\mathtt{fn}(\mathtt{const}(\mathtt{n})) = \emptyset$ is always satisfiable and wait-free.

We feed the constructed formulas to the Z3 SMT solver and check if they are satisfiable. When these formulas are satisfiable for each node of a time dependency graph, then the original Scribble global protocol is feasible and wait-free. This follows from the fact that the checking method presented and implemented in this section closely follows the theoretical constructions used to prove decidability of feasibility and wait-freedom for infinitely satisfiable timed global protocol in [BYY14a, BYY14b] (Proposition 5.1).

5.4. Implementing Timed Protocols with Python

To implement and verify timed local protocols we extend the monitoring framework, previously presented in Chapter 3.

More precisely, we extend the Python conversation API with two standard time primitives, *sleep* and *timeout*, and the monitor tool with capabilities for checking time violations.

Specifically, the API is extended with:

- a *delay* primitive which suspends the execution of a Python program for a prescribed time. The primitive is implemented using a lower level function provided by the 'gevent' library: `gevent.sleep` which lets time elapse for a specified amount of time.
- a *timeout* primitive which interrupts an ongoing computation to meet an approaching deadline. Timeouts are useful for *computation-intensive functions*, operations that take an amount of time which is not negligible such as, for instance, the log crawling performed by the worker in Section 5.1. It may be difficult to foresee the exact duration of a computation-intensive function; in order to ensure that its execution does not exceed the time prescribed by the local protocol, we associate each computation-intensive function to a parameter `timeout` that is an upper bound to the duration of its execution; an exception is raised if the function is not completed in the given time frame. In the implementation of the worker, the function which interrupts the crawling after 20 seconds is `self.crawl(log, word, timeout=20)`; the resulting exception can be handled by simply proceeding with the computation.

In [BYY14a] processes are modelled using a simple extension of the π-calculus with a delay operator `delay(t)`.P that executes as process P after waiting exactly `t` units of time. All of the other actions are assumed to take no time. In practice, however, operations do take time and the delay primitive models standard operations happening between communication actions, as well as time primitives such as delays and timeouts.

The monitor tool is augmented with a local clock and performs several checks summarised below:

- when an action (send or receive) is performed on a conversation channel, the monitor checks if the action is performed within the time constraints specified in a protocol;
- when a delay is issued, the monitor checks that the specified delay time does not exceed time constrains specified in the protocol; and
- when a timeout is issued, the monitor checks that the timeout does not exceed the prescribed time constraints.

We illustrate more concretely the primitives introduced earlier in this section through a Python implementation of the running example. Listing 5.1 shows the Python program for all roles of our running example.

Listing 5.1: Participants implementation in Python

```
#==Master Program=======#
def master_proc():
  c = Conversation.create(...)
  c.send('W', 'task',
    'log', 'string')
  c.delay(22)
  c.receive('W')

  while more_tasks():
    c.send('A','more',
           'data')
    c.send('W', 'more',
     'log', 'string')
    c.delay(22)
    c.receive('W')

  c.send('A', 'end','data')
  c.send('W', 'end')
```

```
#==Worker Program=======
def worker_proc():
  c = Conversation.join(...)
  c.delay(1)
  log = c.receive('M')
  while conv_msg.label != 'end':
    data = self.crawl(log,
    timeout=20)
    c.send('M','result', data)
    c.delay(23)
    conv_msg = c.receive('M')
#==Aggregator Program=======#
def aggregator_proc()
    c = Conversation.join(...)
    op = None
    while op !='end':
      c.delay(23)
      conv_msg = c.receive('M')
      op = conv_msg.label
```

We explain the implementation for the master process (`master_proc`), given in

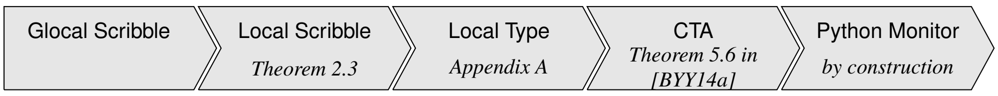


Figure 5.11.: The work-flow for deriving monitors from Global Scribble protocols.

Listing 5.1. The programs starts by creating a *conversation channel* c. Then, following the local protocol, the master sends a request to the worker passing the log name and the word to be counted. The send method, called on conversation channel c, takes as arguments a destination role, a message operator and payload values. This information is encapsulated in the message payload as part of a conversation header and is later used for checking by the runtime verification module. The receive method can take the sender as a single argument, or additionally the operator of the desired message. The code continues with the *delay* operator. Meanwhile, other green threads run, preventing the worker from busy waiting. After the delay expires, the master wakes up and continues.

5.5. Runtime Verification and Enforcement of Time Properties

In this section we recall our monitoring framework, introduced in Chapter 3 and discuss the challenges of verifying and enforcing time constraints. A monitor acts as a membrane between one endpoint and the rest of the network, checking that the send and receive actions performed by timed endpoint programs conform to timed Scribble local protocols. The main property enforced by our framework is that in a network where all endpoints are monitored then either all actions will occur at the prescribed timing, or an error will be detected.

Fig. 5.11 summarises the work-flow of constructing a configuration of local monitors from a global Scribble protocol and justifies the correctness of a monitor implementation: (1) a global Scribble protocol is projected into a set of local Scribble protocols; (2) each local Scribble protocol corresponds to a local type in [BYY14a]; (3) each local type corresponds to a CTA (Communicating Timed Automaton); and (4) a CTA is used to monitor an application. Thus, the monitor is correct by construction.

The monitor has two purposes (or *modes*) with respect to time: error detection and error prevention/recovery.

5.5.1. Error Detection

The monitor verifies communication actions of monitored endpoint against Scribble timed local protocols, expressed as timed automata. First, the monitor verifies that the type (operation and payload) of each message matches its specification and that a message occurs in the right causal order w.r.t. a Scribble protocol (as in the untimed Scribble toolchain). Second, the monitor checks the correct timing of actions. For each ongoing protocol, its respected monitor is augmented with a *local clock*. A synchronisation has been introduced in the prototype to ensure that all processes and monitors will start a protocol at the same time, with clocks set to 0. When a timed endpoint program executes an action the monitor checks the clock constraint of that action (in the timed automaton) against the value of the local clock. The value of the local clock is determined simply as a difference between two timestamps (the initial and the current time). The time is measured using the built-in python function *timeit*.[3] If an action complies with the prescribed timing it is made visible (i.e., forwarded) into the network, otherwise the monitor raises a *TimeException*.

For example, if we change the delay for `master_proc` program in Listing 5.1 to `delay(30)` (instead of `delay(22)`) this will result in a *TimeException*. Thus the monitor stops the time error from propagating into the other endpoints.

5.5.2. Error Prevention and Recovery

This mode relies on the error detection mechanism: when a violation occurs the monitor enforces the clock constraints by generating recoverable actions. We have two types of scenarios: an action is launched by the local endpoint too early or too late (or not at all) w.r.t. the prescribed timing. In the first case, the monitor generates a *delay* equal to the time that is left until an appropriate time is reached, and then it forwards the action to the rest of the network. For example, if we delete any of the `delay` constructs in Listing 5.1 or modify them with a smaller value then the monitor will introduce the missing delay so that the monitored application will appear correct to the network. When a deadline is reached but its associated action is still not executed, the monitor raises a *TimeoutException*. The application can try and recover itself using exception handler, e.g., by interrupting an ongo-

[3]The function timeit returns the time in seconds since the epoch, i.e., the point where the time starts. As recommended in `http://pythoncentral.io/measure-time-in-python-time-time-vs-time-clock/`, this is the preferable method for measuring time in Python.

prescribed action	clock constraint	pre-action	post-action
`s.send`	$x \geq c$		`s.sleep`$(c - x_{cur})$
`s.send`	$x \leq c$	`s.timeout`$(c - x_{cur})$	
`s.recv`	$x \geq c$	`s.sleep`$(c - x_{cur})$	
`s.recv`	$x \leq c$		`s.timeout`$(c - x_{cur})$

Table 5.1.: Recovery actions generated by the monitor

ing computation and continuing the conversation, or restarting the protocol with different settings.

The monitor looks at the next action prescribed by the timed automaton (or *prescribed action*) and acts according to the pre- and post-actions in the table. Pre-actions (resp. post-actions) denote actions performed by the monitor before (resp. after) that the timed endpoint program executes the action that corresponds to the prescribed action. Table 5.1 summarises the actions generated by the monitor in error prevention/recovery mode. In the table x_{cur} is the local clock of the monitor. If the clock constraint of the prescribed action specifies a lower bound $x \geq c$ then the monitor introduces a delay of exactly c (mapped to the low level Python *gevent.sleep* primitive). In case of send we have a post-action: the monitor sleeps *after* observing the action of the endpoint and forwards it to the network at the right time. In case of receive we have a pre-action: the monitor sleeps *before* observing the receive action so that the incoming message will be read at the appropriate time. Similarly, when the clock constraint specifies an upper bound $x \leq c$ the monitor inserts a *timeout* (a timer triggering a *TimeoutException*).

5.6. Benchmarks on Transparency of Timed Monitors

Listing 5.2: A recursive protocol with resets

```
global protocol WordCount2(
  role M, role R, role W){
  [@M: xm<0.01,reset(xm)]
  [@W: xw=0.01,reset(xw)]
  task(log,tring) from M to W;
    rec Loop{
      [@W: xw=0.20]
      [@M: 0.21<xm<0.22]
      result(data) from W to M;
      choice at M{
      [@M: xm=0.22]
  [@A: 0.23<=xa,reset(xa)]
  more(data) from M to A;
  [@M: xm=0.22,reset(xm)]
  [@W: xw=0.23,reset(xw)]
  more(log,string) from M to W;
```

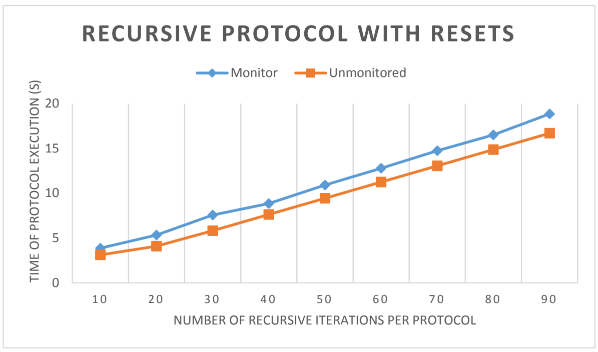


Figure 5.12.: The execution time per number of recursions for the protocol in Listing 5.3

```
  continue Loop;
  } or {
  [@M: xm=0.22][@A: 0.23<=xa]
  end(data) from M to A;
  [@M: xm=0.22][@W: xw=0.23]
  end() from M to W; } }
```

Listing 5.3: Protocol with accumulating delays

```
global protocol ClientServer(
  role C, role S){
  [@C: xc<c][@S: xs=c]
  ping(data) from C to S;
  [@C: xc<2*c][@S: xs=2*c]
  ping(data) from C to S;
  [@C: xc<3*c][@S: xs=3*c]
  ping(data) from C to S;
  ...
  [@C: xc<200*c][@S: xs=200*c]
  ping(data) from C to S;}
```

Listing 5.4: Python implementation for Client and Server role from Listing 5.3

```
def client_proc(t):
  c = Conversation.
  create(...)
  c.receive('S')
  while true:
    c.delay(t)
    c.receive('W')
```

```
def server_proc(t, n):
  c = Conversation.
  create(...)
  c.send('C')
  for i in range(0, n)
    c.delay(t)
    c.send('C')
```

The practicality of our timed monitoring framework depends on the transparency

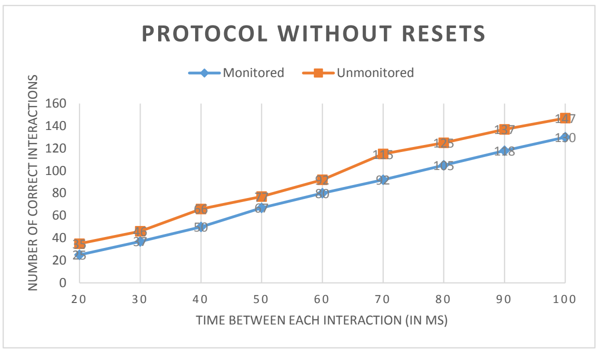


Figure 5.13.: The maximum number of correct interactions for the protocol in Listing 5.4

of the execution in a monitored environment. By transparency we mean: *a program that executes all actions at the right times when running unmonitored will do so when running monitored*. Transparency and overhead are closely related in the timed scenario, since the overhead introduced by the monitor may interfere with the time in which the interactions are executed. We have tested the transparency by providing two different protocols - a protocol with resets and a protocol without resets. The former proves the usability of the monitor in a typical scenario, while the latter demonstrates its limitations.

To set up the benchmark, we have fixed a Scribble timed protocol and manually created a correct implementation of the participants for that protocol using our timed Python API. We run the implementation in two scenarios using our monitoring framework, and with the monitors 'turned off'. The graphs in Fig. 5.12 and 5.13 present the average execution time with a confidence interval of 95 %. The result is obtained by repeating each example 30 times. This experimental design is influences by [GBE07], where a typical large repetition size is $\mathtt{n} \geq 30$.

Participants were run as separate Python applications on the same machine (Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz, Ubuntu 14.04.3 LTS), to minimise the latency between the endpoints. If latency is bigger, the protocol might become unsatisfiable and therefore an error will be triggered by the monitor before the completion of the protocol. For example, in a scenario where a sender's constraint is $x \leq 1$, a receiver's constraint is $x \leq 1$, and a latency is 1.5s (such latency is bigger than the monitor error margin), even if the sender sends the message on time, the monitor at the receiver endpoint cannot receive it on time due to network delay. We are not interested in testing such scenarios, because the errors are not due to monitor overhead. In cases of big latency the monitor correctly detects constraint violations. We want to test the impact that the monitor overhead has on the correctness of the protocol execution in an environment where correctly implemented unmonitored programs would be executed without time violations, such behaviour can be easily exhibited when the latency is negligible w.r.t the granularity of the time units used in time constraints. In the presented results, on average the latency between two endpoints is 0.04. The full benchmark protocols, the applications and the raw data are available from the project page [pyt]. To measure the execution time we use the built-in Python function *timeit* [4].

Scenario 1. We have initially considered a protocol with the same structure as the protocol Fig. 5.3. Running the example with the initial time constraints, however, requires significantly long time runs where most of the execution time is spent performing *time.sleep* at each endpoint (see the Python implementations). Since we are interested in measuring the monitor overhead, which is not affected by how big the actual value constraints are, we decrease the constants in clock constraints and in *time.sleep* by a scale of 100. We use the implementation in Listing 5.1 with delays updated to match the protocol, as shown in Listing 5.2. The outcome is presented in Fig. 5.12. The graph illustrates the time for completing a protocol for increasing number of recursive executions.

This experiment shows that for the given protocol and implementation all executions are without constraint violations. Transparency is guaranteed (i.e., the overhead induced by the monitor does not affect the correctness of the program). Since resets prevent the monitor overhead to *accumulate up to a non negligible overall delay*, delays are bigger than the error margin. The monitor clock is reset at each iteration and therefore the monitor overhead does not accumulate.

Scenario 2. Our second experiment was specifically targeted at checking how many interactions can generate a non-negligible accumulation of delays. We do this by removing resets. In case of no resets both the unmonitored and monitored

[4]The function *timeit* returns the time in seconds since the epoch, i.e., the point where the time starts. As recommended in http://pythoncentral.io/measure-time-in-python-time-time-vs-time-clock/, this is the preferable method for measuring time in Python.

programs are expected to start violating the constraints after certain number of executions. In Scenario 1 recursion allowed us to express repeated interactions by using resets. In order to observe a large number of repeated interactions *without resets* we have created the sequential protocol in Listing 5.3 and implementation in Listing 5.4. We have generated a protocol with 200 consecutive point to point interactions happening at increasing times (at each interaction the time is increased by c). We run the experiment for different values of c (horizontal axis on the figure) and measure the maximum number of interactions (vertical axis on the figure) that can be executed before the program violates the time constraint.

Results, given on the graph in Fig. 5.13, demonstrate that a monitored application performs 90% of the maximum number of possible interactions. This example comes to show the limitations of the timed monitoring framework. The practical scenarios we have encountered so far did not include long sequences of interactions, and repetitive operations are handled via recursions with resets at each cycle.

Further discussion on benchmarks. Overall, the time transparency of the monitoring framework depends on the following:

- *Monitor overhead.* The complexity of monitor checking is linear on the number of protocol states and therefore the upper bound of the overhead can be approximated. Note that errors can be caused by accumulated monitor overhead. If clocks are not reset, the overhead of the monitor is propagated at each run. The accumulated delay can be calculated as a sum of the delays in the longest path between two actions without resets and executed on the same participant in the dependency graph.
- *Error margin.* The monitor accepts an application as correct if its timing t conforms to the time constraints δ in the specification within a certain margin of error ε. Thus, the monitor checks if $t \in [\delta - \varepsilon, \delta + \varepsilon]$. The error margin is a parameter set to the monitor system during its initial configuration. The error margin is application and environment specific, but it should be at least as big as the time accuracy guaranteed by the execution environment. For example, in the Python documentation, the preciseness of the *time* function is specified as follows: *"It returns the time in seconds since the epoch as a floating point number. Note that even though the time is always returned as a floating point number, not all systems provide time with a better precision than 1 second.*"[5]. Therefore, for our benchmark experiments

[5]`https://docs.python.org/2/library/time.html`

Pattern	Use Case	Source
Request-Response Timeout	Travel Agency, SMTP	[Lar, smt]
Messages in a Time Frame	Denial of service attacks, Travel Agency	[Lar]
Action Duration	Progress properties, User inactivity	[UPP]
Repeated Constraint	Pull notification services	[OOIa]

Table 5.2.: Timed patterns

we have chosen an error margin within a second of the specified constraints ($\varepsilon = 0.5s$).

Time errors can also be caused by the execution environment when the machine (or the network) is overloaded. Other system activities can make the kernel unable to schedule the endpoint process as soon as needed. The purpose of the monitor is to detect time errors at their first occurrence and to stop time drifts propagating to other endpoints. Therefore, although in the above case time violations are not a result of a programming error, the executed program should be flagged as wrong, because the time constraints in the protocol are not satisfied.

Note that when the system is under load the time requirements will be violated due to the high load of the system, not because of the overhead induced by the monitor. However, the system load have an impact on the protocol execution, because in a system with inconsistent time drifts the protocol constraints can become unsatisfiable. In an operating system without real-time guarantees, our framework reasons only about soft time deadlines, such as the deadline patterns expressed in Section 5.7.

5.7. Temporal Patterns in Global Protocols

In this section we present a number of timed patterns that we have collected from literature, and which include industrial cases studies [BFM98, KCD+09], verification tools [UPP, Lar] and web service specifications such as the Twitter API and Simple Mail Transfer Protocol (SMTP). Each pattern will be first introduced with a motivation and reference to literature, then modelled using Scribble and applied to a real-case scenario. The given set of patterns does not aim to be exhaustive, but to allow us to assess the usability of Scribble in known timed scenarios. Table 5.2 summarises the timed patterns, with reference to the corresponding use cases and references to literature.

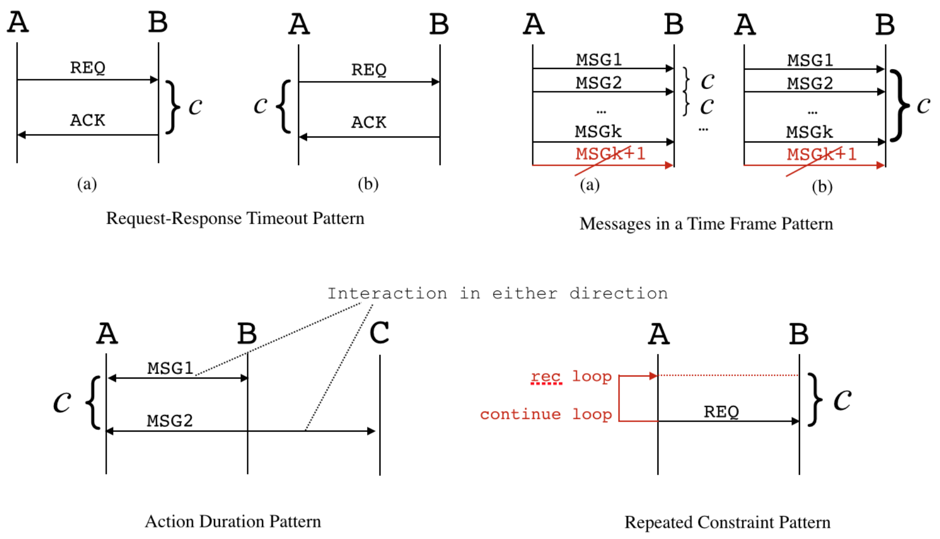

A
B
REQ
ACK
c
(a)
(b)
Request-Response Timeout Pattern
MSG1
MSG2
...
MSGk
MSGk+1
Messages in a Time Frame Pattern
Interaction in either direction
C
Action Duration Pattern
rec loop
continue loop
Repeated Constraint Pattern

Figure 5.14.: An illustrated summary of the timed patterns

Request-Response Timeout Pattern. This pattern allows to enforce quality of service requirements on the timing of a response. The requirement can be set either at the server side or at the client side, as we have identified in the two use cases that follow. In the first use case, drawn from [Lar], a service is requested to reply in a timely manner: *"An acknowledgement message* `ACK` *should be sent (by the server) no later than one second after receiving the request message* `REQ`*"*. Namely, the *sending* of a reply message should be executed within a fixed time after the corresponding request message has been *received*. In the second use case, the Travel Agency web service specification, also drawn from [Lar], the timeout is specified at the client side : *"A user should be given a response* `RES` *within one minute of a given request* `REQ`*"*. In this case, it is the *receiving* of the response from the server that must be possible after the request, within the specified timeout.

The above requirements can be generalised by the following pattern, which is also illustrated in Fig. 5.14:

(a) After receiving a message `REQ` from role `A`, role `B` must send the acknowledgment `ACK` to `A` within c time units.

(b) After sending a message `REQ` to role `B`, role `A` must be able to receive the acknowledgment `ACK` from `B` within c time units.

The corresponding skeletal Scribble specification is given in Fig. 5.15, where in Fig. 5.15 (left) the sending of a reply message should be executed within a fixed

```
[@A:...][@B: reset(xb)]          [@A: reset(xa)][@B:...]
REQ  from A to B;                REQ  from A to B;
...;                             ...
[@B: xb<=c][@A:...]              [@B:...][@A: xa<=c]
ACK from B to A;                 ACK from B to A;
```

Figure 5.15.: Request-Response Timeout Pattern in Scribble: server side (left) and client side (right)

```
[@U: reset(xu1)][...] MAIL from U to S;
[@U: reset(xu2)][...] DATABLOCK from U to S;
[...][@U: xu2<=180000] DATABLOCK_REPLY from S to U;
[...][@U: xu1<=300000] MAIL_REPLY from S to U;
```

Figure 5.16.: SMTP timeouts in Scribble (Request-Response Timeout Pattern)

time after the corresponding request message has been received, and in Fig. 5.15 (right) the receiving of a message after sending a request should be possible within a timeout. A concrete specification can be obtained by instantiating `REQ`, `ACK`, `A`, and `B` with actual messages, roles, and c with a value in $\mathbb{Q}^{\geq 0}$. In Fig. 5.15 (left), x`b` is a clock of `B` and "..." stands for any clock constraints, clock resets, or for any interaction that does not include resets to x`b` or `ACK`. First, the time of receiving the request `REQ` is recorded by resetting the clock x`b`, and then the clock constraint $x\texttt{b} \leq c$ at the reply interaction `ACK` sets the maximum delay to be of c time units. The specification in Fig. 5.15 (right) is similar, but with a reset and a constraint for the user clock x`a`.

In Fig. 5.16 we present an example from the SMTP protocol, featuring a composition of instances of the Request-Response Timeout Pattern (b). The specification prescribes that *"A user should have a 5 minutes timeout for the* `MAIL` *command and 3 minutes timeout for the* `DATABLOCK` *command"*. The combination of two instances of the pattern, at the client side, requires the use of two clocks, one for the `MAIL` command (clock x`u1`), and one for the `DATABLOCK` command (clock x`u2`). In the constraints, note that 3 (reps. 5) minutes correspond to 180000 (resp. 300000) milliseconds.

Messages in a Time Frame Pattern. This pattern allows to set a limit to the number of messages that can be sent in a given time frame. Examples of this requirement can be found in specifications for denial of service attacks [Lar]: *"A user is allowed to send only three redirect messages to a server with interval between* *the messages no less than two time units"*, or in other scenarios such as the Travel Agency Service in [Lar]: *"A customer can change the date of his travel only two times and this must happen between one and five days of the initial reservation"*. These specifications express the repetition of a specified number of messages, occurring either (a) at a specified pace or (b) within an overall specified time-frame.

The above requirements can be generalised by the following pattern, which is also illustrated in Fig. 5.14:

(a) Role `A` is allowed to send role `B` at most k messages, and at time intervals of at least c and at most d time units.

(b) Role `A` is allowed to send role `B` at most k messages in the overall time frame of at least c and at most d time units.

Fig. 5.17 shows the abstract Scribble protocol for this pattern, where `Ga` accounts for case (a) and `Gb` for case (b). The difference is that in (a) the clock x`a` is reset at each interaction. Note that in this pattern we have a-priori knowledge of k, hence any instantiation of the pattern can be resolved as an enumeration of interactions (i.e., without using loops). The definition uses a parametric notation `Ga(k)` for the Scribble 'body' defined in terms of `Ga(k−1)` for the sake of a general and simpler definition of the abstract Scribble protocol. Concrete specification can be obtained by instantiating `A`, and `B` with actual roles, c and d with values in $\mathbb{Q}^{\geq 0}$, and k with non-negative integer values. Note that scenarios in which `A` is supposed to send *exactly* (and not at most) k messages can be modelled by omitting the 'choice' construct and only keeping the first branch (i.e., removing the branch that jumps straight to `Ga(0)` or `Gb(0)`).

Fig. 5.18 shows the protocols for the denial of service attack and the Travel Agency Service in [Lar]. Here we comment on the latter. First, we specify the resetting of the user clock x`u` on the interaction `reservePack`. Then the clock constraint $x\mathtt{u} > 1$ and $x\mathtt{u} < 5$ are set on messages of type `changePack` which are set to be at most two. In this case, we did not convert days into milliseconds for readability.

Action Duration Pattern. This pattern sets a constraint on the delay which is allowed for a participant before executing an action. For example, a common requirement in web services is that: "*A user should not be inactive more than* 30 *minutes*". This pattern can also express progress properties verified by the UPPAAL model checker [UPP] such as: "*A user is allowed to stay in the state for no more than three time units*".

```
Ga    = [@A:reset(xa)][@B:...]
       MSG from A to B; Ga(k-1)
Ga(k) = choice at A {
          [@A:xa>=c and xa<=d, reset(xa)][@B:...]
          MSG from A to B; Ga(k-1)
        } or { Ga(0) }
Ga(0) = [@A:xa>=c and xa<=d][@B:...]
        END from A to B;

Gb    = [@A:reset(xa)][@B:...]
       MSG from A to B; Gb(k-1)
Gb(k) = choice at A {
          [@A:xa>=c and xa<=d][@B:...]
          MSG from A to B; Gb(k-1)
        } or { Gb(0) }
Gb(0) = [@A:xa>=c and xa<=d][@B:...] END from A to B;
```

Figure 5.17.: Messages in a Time Frame Pattern in Scribble: single message time frames (Ga) and overall timeframe (Gb).

```
Ga   = [@U:reset(xu)][...] REDIRECT from U to S; Ga(2)
Ga(k) = choice at U {
      [@U:xu>=2, reset(xu)][...] REDIRECT from U to S;
      Ga(k-1)
    } or {
      [@U:xu>=2, reset(xu)][...] END from U to S;}

Gb   = [@U:reset(xu)][...] reservePack from U to S; Gb(2)
Gb(k) = choice at U {
      [@U:1<=xu<=5][...] changePack from U to S; Gb(k-1)
    } or {
      [@U: true][...] END from U to S;
    }
```

Figure 5.18.: Denial of service attack (Ga) and Travel Agency Service (Gb) use cases in Scribble

```
[...][@B: reset(xb)]                [@A: reset(xa)][...]
MSG1 from A to B;                   MSG1 from A to C;
[...][@B: xb<=30, reset(xb)]        [@A: xa<=3][...]
MSG2  from A to B;                  MSG2  from A to B;
```

Figure 5.19.: Protocols for web service user inactivity (left) and for the UPPAAL progress property (right).

```
rec Pull{
  [...][@A: xa==c; reset(xa)] REQ from A to B;
continue Pull;}
```

Figure 5.20.: Repeated Constraints Pattern in Scribble

The constraints above can be generalised as follows: the time elapsed between two actions performed by the same role A, where each of these two actions can be either a send or a receive action, must not exceed c time units. Notice that, unlike in the Request-Response Timeout Pattern, the constraint is specified only in terms of A and not on the type of the previous actions performed by A or the other roles interacting with A.

Fig. 5.19 (left) and Fig. 5.19 (right) present two instances of the Action Duration Pattern in Scribble: the protocols for web service user inactivity and the one for the UPPAAL progress property, respectively. In Fig. 5.19 (left) the clock xa is reset on the first message MSG1 from A to B. Then the clock check $x\mathtt{a} \leq 30$ guarantees that the user cannot send another message MSG2 if the time constraint is violated. The verification of the progress property in Fig. 5.19 (right) is similar, but more roles are involved (A, B, C). The clock is reset during the interaction between A and C, while the clock constraint is checked on the interaction between A and B.

Repeated Constraints Pattern. The pattern captures requirements occurring in pull notification services, where the intervals at which a certain interaction should be repeated is fixed. For example: "*The email client should request the emails from the service every 5 seconds*".

In general: role A must send messages to B every c time units. The main differences with the Messages in a Time Frame Pattern is that the number of messages is not bounded, and the time is required to elapse between messages of exactly c time units. For readability (i.e., to avoid nested 'choice' constructs) we will show the abstract Scribble specification in the case of non-terminating interaction (Fig. 5.20).

The repetition of the pull interactions is expressed via recursion (recPull). At every recursive iteration, the clock xa for A is checked against the constraint and then reset.

5.8. Related Work

Specification languages. The need for specifying and verifying the temporal requirements in a distributed systems is recognised. To this aim, different specification methods and verification tools have been developed, especially in the area of business process modelling (see [CKGJ13] for a survey on verification of temporal properties). The work in [WIH11] describes a framework for analysing choreographies between BPEL processes with time annotations. [GDZ12] extends BPML with time constraints. Via a mapping from BPML to timed automata, it allows verification with the UPPAAL model checker. As a language for timed protocol specifications, the main advantage of Scribble over alternatives such as BPEL, BPML and timed automata, is that it enables enforcement of global properties, e.g., conformance of the interactions to global protocols, providing an in-built mechanism (projection) for decentralisation of the verification. Another expressive formal language for specifying time-properties is XTUS-Automata, introduced in [KCD$^+$09]. Specifications written in XTUS-Automata are automatically translated into BPEL aspects, which ensure temporal constraints at runtime. Both absolute and relative time can be specified. Their approach detects contradictions between the temporal constraints implemented in the composition, but does not verify inconsistency at the specification level, as the Scribble checker can offer. As a future work we plan to identify formal transparency requirements to calculate time drifts that make a protocol unsafisfiable. We also plan to test the preciseness of time violations on real-time kernels, such as the Ubuntu RT_PREEMPT patch set. This will allow us to specify and enforce hard time deadlines.

Verification tools and frameworks. Among the state-of-the-art runtime verification tools, a few support specification of time properties [dBdGJ$^+$14, CR07, Lar]. The work in [CR07] presents a generic monitor that can be parameterised on the logic. [dBdGJ$^+$14] combines temporal properties and control flow specifications in a single formalism specified per object class. Our recovery mechanism resembles the aspect-oriented approaches used in those verifiers, but the combination of control flow checking and temporal properties in the same global specification is an unique characteristic of our work. Our tool statically checks the correctness of the specification itself in addition to the runtime checks for the program. Furthermore, via its formal basis, our framework allows to combine static verification [BYY14a] and dynamic enforcement. Other works from the multi-agent community have studied distributed enforcement of temporal properties through monitoring. [MU00] presents a distributed architecture for local enforcements, where monitors detect if agents fulfill their specification within a given deadline. The specifications are expressed in the form of event-condition-action and all agents belonging to a group obey the same roles. Our work presents a different perspective where agents follow personalised laws based on the role they play in each protocol. In runtime verification for Web Services, the work [CDVM11] transforms a subset of Web Services Choreography Description Language into timed-automata and proves their transformation is correct with respect to timed traces. Their approach is model-based, static and centralised, and does not treat either dynamic checking, or the properties of feasibility and wait-freedom.

Time properties. Several works study properties of timed formalisms, especially in the contexts of automata [KY06] and Message Sequence Charts (MSCs) [GMNK09, AGMK10]. In the timed scenario, problems such as reachability are known to be undecidable in the general case. This has been shown, for example, for timed extensions of Message Sequence Charts (e.g., Time-Constrained MSC [GMNK09]) and CTA [KY06]. Timed MPSTs [BYY14a] tackle tractability issues by relying on the syntactic constraints also inherited by Scribble and mentioned in Section 5.2.1, and a few extra constraints on time annotations of global protocols. In particular:

- The correspondence between timed MPSTs and CTA [BYY14a], ensuring the progress and liveness properties studied in [DY13], requires an additional constraint on the shape of global protocols, that is feasibility (recall that it ensures that at any point of the protocol the current time constraint should be satisfiable for any possible past).
- Progress of a well-typed program requires two additional properties: feasibility (as above) and wait-freedom.

The work in [BYY14a, BYY14b] proposes a decidable method to check if a global type is feasible and wait-free. This method, however, relies on a further restriction on global types, called *infinite satisfiability* and explained in Section 5.2. Infinite satisfiability has recently been relaxed [BLY15] to allow some clocks not to be reset, as long as there is a viable 'escape' from the loop, but has not yet been integrated in the Scribble toolchain. As the focus of the work presented in this chapter is to illustrate the implementation, practicality and usability aspects of the timed Scribble toolchain, we will not recall the technical details of the theory here, they are available from [BYY14a, BYY14b, BLY15].

Other approaches focus on the reachability problem, and are related to our algorithms for checking feasibility and wait-freedom of timed global specifications. [Tri99] gives an algorithm to check deadlock freedom for timed automata. The algorithm is based on syntactic conditions on the states relying on invariant annotations. Their approach is not directly applicable to check feasibility on timed global specifications. Other methods based on CTA [KY06] or MSC [GMNK09] address complexity by restricting topology and/or channel size. Remarkably, none of the conditions required by timed MPSTs (hence Scribble) set limitations on the network topology nor buffer size. For example the simple Scribble example below

```
rec pro [@A : x < 10,reset(x)][@B : true] a(T) from A to B; continuepro
```

allows the channel from A to B to have an arbitrary number of messages (as B can arbitrarily delay the receive actions). Our restrictions are, rather, induced by the conversation structure of timed global protocols. Tractability of MPSTs derives from the way interactions are structured and the way resets are organised.

The work in [AGMK10], focused on an extension of MSCs with timed events, is particularly related to [BYY14a] (on which theory the work presented in this chapter is based): the former focuses on the problem of language inclusion while the latter on statically checking conformance of abstract session types with concrete programs. The work in [AGMK10] gives a verification method that is decidable when the communication graph describing the behaviour of each loop can be modelled as a single strongly connected component, which implies that channels have an upper bound. In this chapter we hinge at the theory for static checking from [BYY14a] but practically apply it to dynamic checking.

Calculi with time. Our timed API is based on the session π-calculus with delays, introduced in [BYY14a]. The delay primitive from [BYY14a] has been used as a model for the timeout and sleep extensions of the Python API presented in Section 5.4. Other recent works that extend process calculi with time are, for example, [BY07, LP12, LZ02] where timeouts are added to the π-calculus syntax. Time elapses as discrete ticks and delays propagate asynchronously through the system. Timed COWS [LPT07a] extends COWS with a 'wait' primitive similar to our delays. The work in [GDNP97], which is targeted to testing, introduces a 'wait' construct along with delay annotations for actions. [SG13] extends the

π-calculus with absolute and continuous times, and includes timed constraints inspired by timed automata. Web-π [LZ05] and C^3 [LP12] extend the π-calculus and the Conversation Calculus (CC), respectively, to enable the reasoning on the interplay between time and exceptions.

The main focus of our work is different from the above works: we use timed specifications *protocols*, rather than enriching timed primitives in the π-calculus (Python) syntax level. However, these works inspire several future directions for our framework. Extensions allowing dynamic time-passing as in [SG13] and [LZ02] at the π-calculus level are possible, for instance, by extending with time the approaches in [BHTY10] and [BDY12], in which the global specifications directly model properties of the message contents.

5.9. Concluding Remarks

The chapter presents the design and implementation of a real-time verification framework. At the specification level our time extension controls and checks, via timed Scribble, the feasibility and wait-freedom of processes at the early design phase. At the programming level, we have introduced primitives for a fine grained management of time which enable to schedule interactions at exact timings w.r.t. a given protocol. At runtime, early error detection is guaranteed by raising time exceptions when time deadlines are not met. Furthermore, the monitor is augmented with enforcement capabilities for recovering from runtime violations of the time constraints. Benchmarking has revealed an interesting relationship between transparency and overhead introduced by time: monitoring overhead may break transparency by introducing delays, hence violations. The practicality of the proposed approach for specification and dynamic verification of distributed interactions has been demonstrated via: (1) the representation of a number of scenarios (i.e., a OOI use case as well as time patterns distilled from literature) with Scribble, and (2) benchmarking, showing that the overhead introduced by our monitor is, in the scenarios we encountered, negligible.

6. Monitoring Actor Programs

The actor model [Agh86, AMST97] is (re)gaining attention in the research community and in the mainstream programming languages as a promising concurrency paradigm. Unfortunately, the programming model itself does not ensure correct sequencing of interactions between different computational processes. A study in [TDJ13] points out that "the property of no shared space and asynchronous communication can make implementing coordination protocols harder and providing a language for coordinating protocols is needed". This reveals that the coordination of actors is a challenging problem when implementing, and especially when scaling up an actor system.

To overcome this problem, we need to solve several shortcomings existing in the actor programming models. First, although actors often have multiple states and complex policies for changing states, no general-purpose specification language is currently in use for describing actor protocols. Second, a clear guidance on actor discovery and coordination of distributed actors is missing. This leads to ad-hoc implementations and mixing the model with other paradigms which weaken its benefits [TDJ13]. Third, no verification mechanism (neither static nor dynamic) is proposed to ensure correct sequencing of interactions in a system of actors. Most actor implementations provide static typing within a single actor (even with expressive pattern matching capabilities as surveyed in Section 6.6), but the communication between actors – the complex communication patterns that are most likely to deadlock – are not checked.

In this chapter we tackle the aforementioned challenges by studying applicability of multiparty session types (MPST), and the specification language Scribble in particular, [HYC08] to actor systems.

In previous chapters (Chapter 3, 4, 5) we prove the suitability of the protocol description language Scribble and its tools for the dynamic verification of real world complex protocols [OOIa]. We presented a runtime verification framework that guarantees safety and session fidelity of the underlying communications. The MPST dynamic verification framework, studied in the previous chapters, is built on a runtime layer for protocol management and developers use MPST primitives for communication, which limits the verification methodology to a specific infrastructure. In this chapter, we adapt the model to present a Scribble-based runtime verification of actor systems.

Contributions and outline. In this chapter, we prove the generality of Scribble-based runtime verification by showing Scribble protocols offer a wider usage, in particular, for actor programming. A main departure from our works, presented in Chapter 3 and further extended in Chapter 4 and Chapter 5, is that an actor can implement multiple roles (in the previous chapters a process was bound to a single role). This extension allows data to be shared between roles (if the roles reside in the same actor), and hence mitigates the problem that sharing data only via message-passing introduces undesirable performance implications.

Our programming model is grounded on three new design ideas: (1) use Scribble protocols and their relation to finite state machines for specification and runtime verification of actor interactions; (2) augment actor messages and their mailboxes dynamically with protocol (role) information; and (3) propose an algorithm based on virtual routers (protocol mailboxes) for the dynamic discovery of actor mailboxes within a protocol. We implement a session actor library in Python to demonstrate the applicability of the approach. To the best of our knowledge, this is the first design and implementation of session types and their dynamic verification toolchain in an actor library.

The chapter is organised as follows:

Section 6.1 gives a brief overview of the the key features of our model through an example.

Section 6.2 describes the constructs of Session Actors and highlights the main design decisions.

Section 6.3 presents the implementation of the model on concrete middleware.

Section 6.4 evaluates the framework overheads, compares it with a popular Scala actor library [akk] and shows applications.

Section 6.5 presents and implements representative actor use cases in our framework.

Section 6.6 discusses related work and concludes.

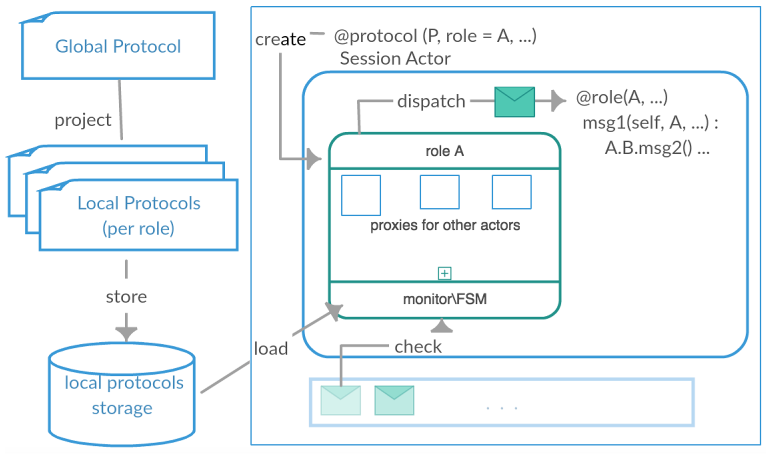


Figure 6.1.: MPST-development methodology (left) and session actors (right)

6.1. Session Actors Programming Model

This section explains how we incorporate the Scribble-based runtime verification model with the programming model of actors. The top-down development methodology for Scribble-based monitoring, presented in Chapter 3, is shown on the left part of Fig. 6.1. The figure on the right illustrates the Session Actor programming model. It consists of annotated Python classes.

6.1.1. Background

Actor model. We assume the following actor features to determine our design choices. Actors are concurrent autonomous entities that exchange messages asynchronously. An actor is an entity (a class in our framework) that has a mailbox and a behaviour. Actors communicate between each other only by exchanging messages. Upon receiving a message, the behaviour of the actor is executed, upon which the actor can send a number of messages to other actors, create a number of actors or change its internal state. Each actor is equipped with a mailbox where messages are buffered. There are only three primitive operations each actor can perform: (1) create new actors; (2) send messages to other actors whose address the sender knows; and (3) become an actor with a new state and behaviour. All these operations are atomic. The only order inherent is a causal order, and the only guarantee provided is that messages sent by actors will eventually be processed by their receivers. In addition, in our framework we enforce synchronisation and coordination constraints (as explained in Section 6.1.2).

Scribble-based monitoring. In Fig. 6.1 (left) we recall the top-down development methodology for MPSTs runtime verification, introduced in Chapter 3 .

A distributed protocol is specified as a global Scribble protocol, which collectively defines the admissible communication behaviours between the participating entities, called *roles*, inside the protocol. Then, the Scribble toolchain is used to algorithmically project the global protocol to local specifications.

For each role a finite state machine (FSM) is generated from the local specification. The FSM prescribes the allowed behaviour for each role and is used at runtime to perform validation of the allowed interactions. When a party requests to start or join a session, the initial message specifies which role it intends to play. Its monitor retrieves the local specification based on the protocol name and the role. A runtime monitor ensures that each endpoint program conforms to the core protocol structure.

6.1.2. Session Actors Framework

We explain how we apply the above methodology. As observed in [STM14] in many actor systems an actor encapsulates a number of passive objects, accessed from the other actors through asynchronous method calls. Similarly, we introduce *multiparty roles*, which allow multiple local control flows inside an actor. We call an actor hosting *multiparty roles* a Session Actor.

Session Actors. The session actor architecture is depicted in Fig. 6.1 (right). To verify actor interactions, each role inside an actor is associated with a FSM, generated from a global protocol. Roles are enabled on an actor instance via two annotations, a protocol and a role annotation. The former specifies a role declaration, while the latter specifies the role behaviour. Without the introduced annotations a session actor behaves identically to a plain actor.

The figure shows the main actions performed by the framework and explained below:

1. `create` denotes the creation of a role inside an actor. The `@protocol` annotation creates a role object when the actor class is spawned. The role is associated to a protocol and contains proxies (remote references) for other actors (the other roles in the protocol) and a monitor.
2. `load` denotes loading a local protocol from the protocol storage when the role is created. The role monitor loads the protocol using the protocol name

and generates a FSM that is used for checking.

3. `check` denotes checking of a message. When a message is picked up from the actor queue, it is checked by the role monitor and, if correct, dispatched to its handler.
4. `dispatch` denotes dispatching the message to the message handler. When a message arrives in the actor queue, it is despatched to the according handler (method) w.r.t the meta information stored in the message (a target role and a message label).

Here is an example how the `@role` annotation is used. We consider the following partial protocol:

```
m2() from buyer to seller;
m1() from seller to buyer;
```

The message transfer declaration:

```
m2() from buyer to seller;
```

corresponds to a method declaration:

```
@role(seller, buyer)
def m2(self, seller, ...)
```

The first parameter of the role decorator is a reference to a role object, named `seller`. This reference can be used inside the message body. The second parameter is the name of the sending role (`buyer`) and is used only by the monitor to check that the message is sent from that particular role. The proxies of the other actors engaged in the protocol can be easily accessed inside the method body via the role reference `seller`. For example, `seller.buyer` is a proxy for the actor that implements the `buyer` role. When the call `seller.buyer.send.m2()` is performed, the monitor for the role `seller` checks the `seller` is allowed to send a message `m2` to the `buyer`.

In a nutshell, an actor can be transformed to a session actor by applying the following design methodology:

1. the global protocol, that the actor is associated with, is written in Scribble and projected to local specifications using the Scribble toolchain, presented in Chapter 2 and available from [scrb]. The tool generates one file (local protocol) per role.
2. the actor class is annotated with a `@protocol` decorator.

3. each method is annotated with a `@role` decorator.
4. interactions with other actors are performed via the role instance, specified in the `@role` decorator.

Protocol mailbox. Protocol mailboxes are the post offices of our systems and they exist for the duration of a protocol execution. Essentially they are virtual routers, which delegate messages according to their routing table. A protocol mailbox should be explicitly created by the developer using the `Protocol.create` primitive. This primitive generates a fresh protocol id (the name of the virtual router) and sends the id to the actors implementing the roles for the protocol.

The changes and additions that the MPST annotations bring to the original actor model are as follows (here we give a high level overview, while the technical details regarding their implementation are given in subsequent sections):

(a) different passive objects (called roles) that run inside an actor are given a session type (Section 6.3.2).

(b) an actor engages in structured protocol communications via protocol mailboxes (Section 6.3.3).

(c) an actor message execution is bound to a protocol structure (Section 6.3.4). This structure is checked via the internal FSM-based monitor.

Verification of Session Actors. To guarantee session fidelity (actors follow the behaviour prescribed by protocols) we perform two main checks. First, we verify that the type (a label and a payload) of each message matches its specification. In our implementation we have mapped message transfer labels to method invocations. For example,

```
m1(int)from A to B;
```

specifies that there is a method named `m1` with an argument of type `int` in the actor, implementing role `B`. Therefore, if such a method does not exist or its signature does not match, an error will be detected.

Second, we verify that the overall order of interactions is correct, i.e. interaction sequences, branches and recursions proceed as expected, respecting the explicit dependencies. Consider the following protocol:

```
m1() from A to B;
m2() from B to C;
```

The protocol specifies that there is a causality between m1 and m2 at the role B. More precisely, role B is not allowed to send the message m2() to C before receiving the message m1() from A. Thus, errors such as communication mismatches that violate the permitted protocol are ruled out.

If all messages comply to their assigned FSMs, the whole communication is guaranteed to be safe. If an actor does not comply, violations (such as deadlocks, communication and type mismatch) are detected dynamically. The monitor can either block the faulty messages (outgoing faulty messages are not dispatched to the network, incoming faulty messages are not dispatched to the message handlers) or issue a warning.

6.1.3. Use Case: Warehouse Management

To illustrate and motivate central design decisions of our model, we present the buyer-seller protocol from [HYC08] and extend it to a full warehouse management scenario. A warehouse consists of multiple customers communicating to a warehouse provider. It can be involved in a Purchase protocol (with customers), but can also be involved in a StoreLoad protocol with dealers to update its storage.

Scribble protocol. The interactions between the entities in the system are presented as two Scribble protocols, shown in Fig. 6.2 (left) and Fig. 6.2 (right). The protocols are called Purchase and StoreLoad and involve three (a buyer (B), a seller (S) and an authenticator (A)) and two (a store (S), a dealer (D)) parties, respectively. At the start of a purchase session, B sends login details to S, which delegates the request to an authentication server. After the authentication is completed, B sends a request quote req for a product to S and S replies with the product price. Then B has a choice to ask for another product (req), to proceed with buying the product (bye), or to quit (quit). By buying a product the warehouse decreases the product amount it has in the store. Products in stock are increased as prescribed by the StoreLoad protocol, Fig. 6.2 (right). The protocol starts with a recursion where a warehouse (in the role of S) has a choice to send a product request, of the form req(string, int), to a dealer D. The first argument of req specifies the name of a requested product as a string, the second argument is of type int and stands for the number of requested products. After receiving the request, D delivers the product (put in line 8). These interactions are repeated in a loop until

```
 1 global protocol Purchase(
 2   role B, role S, role A)
 3 {
 4 login(str) from B to S;
 5 login(str) from S to A;
 6 auth(str) from A to B, S;
 7 choice at B
 8   {req(str) from B to S;
 9    (int) from S to B;
10   }
11   or
12   {buy(str) from B to S
13    deliver(str) from S to B;
14   }
15   or
16   {quit() from B to S;}}
```

```
 1 global protocol StoreLoad
 2   (role D, role S)
 3 {
 4 rec Rec{
 5   choice at S
 6     {req(str, int) from S to D;
 7      put(str, int) from D to S;
 8 continue Rec;}
 9   or
10   {quit() from S to D;
11    acc() from D to S;}}}
```

Figure 6.2.: Global protocols in Scribble for (left) Purchase protocol and (right) StoreLoad protocol

S decides to quit the protocol (line 11).

Challenges. There are several challenging points implementing the above scenario. First, a warehouse implementation of the Store role should be involved in both protocols, therefore it can play more than one role. Second, initially the user does not know the exact warehouse it is buying from, therefore the customer learns dynamically the address of the warehouse. Third, there can be a specific restriction on the protocol that cannot be expressed as system constants (such as specific timeout depending on the customer). The next section explains the implementation of this example in Session Actors.

6.2. Session Actor Language

This section explains the main operations in the session actor language and its use case implementation in Python.

Session Actor language operations. Fig. 6.3 presents the main session actor operations and annotations. The central concept of the session actor is the *role*. A role can be considered as a passive object inside an actor and it contains meta information used for MPST-verification. A session actor is registered for a role via the `@protocol` annotation. The annotation specifies the name of the protocol and

Conversation API operation	Purpose
`@protocol(self_role, protocol_name, self_role, other_roles)ActorClass`	Annotating actor class
`@role(self_role, sender_role)msg_handler`	Annotating message handler
`Protocol.create(protocol_name, mapping)`	Initiating a conversation
`c.role.send.method(payload)`	Sending a message to an actor in a role
`join(self, role)`	Actor handler for joining a protocol
`actor.send.method(payload)`	Sending a message to a known actor

Figure 6.3.: Session Actor operations

all the roles that are part of the protocol. The aforementioned meta information is stored in a role instance created in the actor. To be used for communication, a method should be annotated with a role using a `@role` decorator and passing a role instance name. The instance serves as a container for references to all protocol roles, which allows sending a message to a role in the session actor without knowing the actor location. A message is sent via `c.role.send.method`, where `c` is the self role instance and `role` is the receiver of the message.

Sending to a role without explicitly knowing the actor location is enabled through a mechanism for actor discovery (explained in Section 6.3). When a session is started via the `create` method, actors receive an invitation for joining a protocol. The operation `join` is a default handler for invitation messages. If a session actor changes the join behaviour and applies additional security policies to the joiners, the method should be overloaded. Introducing actor roles via protocol executions is a novelty of our work: without roles and the actor discovery mechanism, actors would need additional configurations to introduce their addresses, and this would grow the complexity of configurations.

Listing 6.1: Session Actor implementation for the Warehouse role

```
@protocol(c, Purchase, seller, buyer, auth)
@protocol(c1, StoreLoad, store, dealer)
class Warehouse(SessionActor):
  @role(c, buyer)
  def login(self, c, user):
    c.auth.send.login(user)

  @role(c, buyer)
  def buy(self, c, product):
    self.purchaseDB[product]-=1;
    c.buyer.send.delivery(product.details)
    self.become(update, product)
  @role(c, buyer)
```

```
14    def quit(self, c):
15      c.buyer.send.acc()
16
17    @role(c1, self)
18    def update(self, c1, product):
19      c1.dealer.send.req(product, n)
20
21    @role(c1, dealer)
22    def put(self, c1, product):
23      self.purchaseDB[product]+=1:
24
25  def start_protocols():
26    c =   Protocol.create(Purchase, {seller:Warehouse,
27  buyer: Customer, auth: Authenticator})
28    c1 = Protocol.create(StoreLoad, {store: c.seller,
29   dealer: Shop})
30    c.buyer.send.start()
```

Warehouse service implementation. We explain the main constructs by an implementation of a session actor accountable for the Warehouse service. Listing 6.1 presents the implementation of a warehouse service as a single session actor that keeps the inventory as a state (`self.purchaseDB`). Lines 1–2 annotate the session actor class `Warehouse` with two protocol decorators – `c` and `c1` (for seller and store roles respectively). The instances `c` and `c1` are accessible within the warehouse actor and are holders for mailboxes of the other actors, involved in the two protocols.

All message handlers are annotated with a role and for convenience are implemented as methods. For example, the login method (line 5) is invoked when a `login` message (line 4, Fig. 6.2 (left)) is sent. The role annotation for `c` (line 4) specifies the originator of the message to be `buyer`.

The handler body continues following line 5, Fig. 6.2 (left) – sending a `login` message via the `send` primitive to the session actor, registered as a role `auth` in the protocol of `c`. Value `c.auth` is initialised with the `auth` actor mailbox as a result of the actor discovery mechanism (explained in the next section). The handling of `auth` (line 6, Fig. 6.2 (left)) and `req` (line 6, Fig. 6.2 (right)) messages is similar, so we omit it and focus on the `buy` handler (line 9–12), where after sending the delivery details (line 11), the warehouse actor sends a message to itself (line 12) using the primitive `become` with value `update`. Value `update` is annotated with another role `c1`, but has as a sender `self`. This is the mechanism used for switching between roles within an actor. Update method (line 18–19) implements the request branch (line 6–9, Fig. 6.2 (right)) of the `StoreLoad` protocol – sending a request to the `dealer` and handling the reply via method `put`.

The correct order of messages is verified by the FSM, embedded in `c` and `c1`. As a result, errors such as calling `put` before `update` or executing two consecutive updates, will be detected as invalid.

A protocol is started using the `Protocol.create` function. It takes, as parameters, (1) the name of a protocol and (2) the mapping between the role names and the actor classes implementing the roles. Note that for readability we use more descriptive role names –`buyer`, `seller` and `auth` instead of `B`, `S`, `A` respectively. Creating a protocol involves (1) creating a protocol mailbox with a newly generated protocol id; (2) spawning all actors (if they still have not been started) involved in the protocol; and (3) sending the address of the protocol mailbox (the protocol id) to all actors. Lines 26 starts the `purchase` protocol and spawns the Warehouse, Customer and Authenticator actors (Warehouse, Customer and Authenticator are names of actor classes). Lines 28 starts the `storeLoad` protocol, but since the Warehouse actor is already spawned, we do not pass the actor class, but a reference to an existing actor (`c.seller`). Then, `c.buyer.send.start()` sends a message to buyer to start the protocol interactions as the global protocol prescribes (the protocol prescribes the first interaction is from `buyer` sending a `login` message to the `seller`).

6.3. Implementation of Session Actors

This section explains our implementation of Session Actors. The key design choices follow the actor framework explained in Section 6.1.2. We have implemented the multiparty session actors on top of Celery [Cel] (a Python framework for distributed task processing) with support for distributed actors [Act]. Celery uses Advanced Message Queue Protocol (AMQP 0-9-1 [AMQ]) as a transport. The reason for choosing AMQP network as base for our framework is that AMQP middleware shares a similar abstraction with the actor programming model, which makes the implementation of distributed actors more natural.

6.3.1. AMQP Background

Although we have already introduced AMQP in Chapter 3, in this subsection we recall AMQP notions specifically needed for actor modeling. We first remind the key features of the AMQP model, also summarised in Chapter 3. In AMQP, messages are published by producers to entities, called exchanges (or mailboxes). Exchanges distribute message copies to queues using binding rules. Then AMQP brokers (virtual routers) deliver messages to consumers subscribed to queues. Exchanges can be of different types, depending on the binding rules and the routing strategies they implement.

We use three exchange types in our implementation: *round-robin* exchange (deliver messages, alternating to all subscribers), *direct* exchange (subscribers subscribe with a binding key and messages are delivered when a key stored in the message meta information matches the binding key of the subscription) and *broadcast* exchange (deliver a message to all subscribers).

Distributed actors are naturally represented in this AMQP context using the abstractions of exchanges. Each actor type is represented in the network as an exchange and is realised as a consumer subscribed to a queue based on a pattern matching on the actor id. Message handlers are implemented as methods on the actor class. Our distributed actor discovery mechanism draws on the AMPQ abstractions of exchanges, queues and binding.

6.3.2. Actor Roles

Our extensions to the actor programming model are built using Python advanced abstraction capabilities: two main capabilities are *greenlets* (for realising concurrency between roles) and *decorators* (for annotating actor types and methods).

A greenlet (or micro/green thread) is a light-weight cooperatively-scheduled execution unit in Python. A Python decorator is any callable Python object that is used to modify a function, method or class definition, annotated with `@` symbol. A decorator is passed the original object being defined and returns a modified object, which is then bound to the name in the definition. The decorators in Python are partly inspired by Java annotations.

A key idea of actor roles is that each role runs as a micro-thread in an actor (using Python greenlet library). A role is activated when a message is received and ready to be processed. Switching between roles is done via the `become` primitive (as demonstrated in Listing 6.1), which is realised as sending a message to the internal queue of the actor. Roles are scheduled cooperatively. This means that at most one role can be active in a session actor at a time.

Actors are assigned roles by adding the `@protocol` decorator to the actor class declaration. The annotation

```
@protocol(seller, "Purchase", "seller", ["buyer"])
class Warehouse
```

creates a private field `seller` on the `class Warehouse`. The field has a type `ActorRole`. `ActorRole` is a custom class that encapsulates an actor role behaviour. In this case, the behaviour of the `seller` role. We recall Fig. 6.1, which illustrates the structure of a session actor. In addition to the protocol name, and an `ActorRole` instance (role `A` in Fig. 6.1) also contains a corresponding FSM monitor (generated from a Scribble protocol) and the addresses to the other actors in the protocol. Thus, the other participating actors in the protocol are accessed using role names (for example, `seller.buyer` returns the address of the actor that implements `buyer`).

All communication interactions on the role instance `seller` should follow the interactions prescribed by the protocol `Purchase` for the role `seller`. Therefore:

1. all labels sent to the role `seller` in the protocol specification, are implemented as methods of the class `Warehouse` and
2. the above methods are annotated with `@role` decorator.

For example, the specification that a `buyer` sends a message `quit` to the `seller` requires that there is a method `quit` annotated with `@role(seller, buyer)`. The latter can be read as `@role(receiver_role=seller, sender_role=buyer)`.

Messages received in the actor mailbox are dispatched to the matching handlers (methods) only if the `@role` decorator for the method corresponds to the meta information about the sender and receiver roles in the message header. For example, the message dispatch as a result of the call `buyer.seller.send.quit()` will invoke the body of the method declared as: `@role(seller, buyer)def quit`.

6.3.3. Actor Discovery

Fig. 6.4 presents the network setting (in terms of AMQP objects) for realising the actor discovery for `buyer` and `seller` of the protocol `Purchase`. We use three AMQP exchange types, as explained in Section 6.3.1. For simplicity, we create some of the objects on starting of the actor system – round-robin exchange per

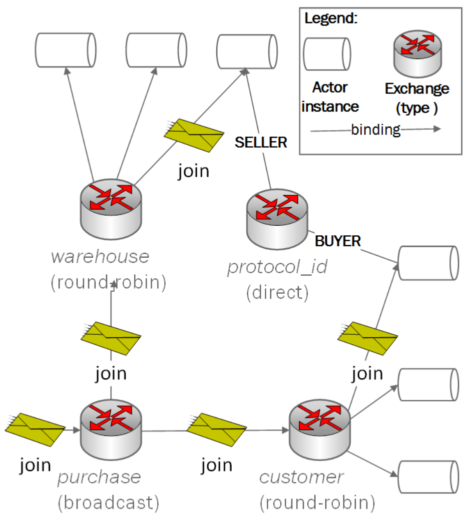


Figure 6.4.: Organising Session Actors into protocols

actor type (`warehouse` and `customer` in Fig. 6.4) and broadcast exchange per protocol type (`purchase` in Fig. 6.4). All spawned actors alternate to receive messages addressed to their type exchange. When session actors are started their exchange is bound to the protocol exchange according to their `@protocol` annotations (line 1 in Listing 6.1 binds `warehouse` to `purchase` in Fig. 6.4).

Exchanges are very cheap to create with a throughput of one million messages per second[1]. Therefore we do not consider their use as a bottleneck of our implementation. Messages are not stored in the exchange and therefore only the exchange throughput evaluation is relevant to performance concerns.

The session initiation takes advantage of the available AMQP infrastructure. The actor discovery wraps the boilerplate code for spawning new actors, creating new session id and exchanging the actor addresses between the involved participants. The exact exchange bindings for our running example are explained below.

We now explain the workflow for actor discovery. When a protocol is started, a fresh protocol id and an exchange with that id are created. An AMQP direct exchange (see Section 6.3.1) is used so that messages with a routing key are delivered to actors linked to the exchange with binding to that key (it corresponds to `protocol_id` in Fig. 6.4). Then `join` message is sent to the protocol exchange and delivered to one actor per registered role (`join` is a broadcast to `warehouse`

[1] http://www.rabbitmq.com/blog/2011/03/28/very-fast-and-scalable-topic-routing-part-2/

and customer in Fig. 6.4). On join, an actor binds itself to the protocol_id exchange with subscription key equal to its role (bindings seller and buyer in Fig. 6.4). When an actor sends a message to another actor within the same session (for example c.buyer.send.ack() in Listing 6.1), the message is sent to the protocol id exchange (stored in c) and from there delivered to the buyer actor.

6.3.4. Preservation through FSM Checking

Before a message is dispatched to its message handler, the message goes through a monitor. We use the same monitor implementation, as the one presented in Section 3.3.1. We recall the monitoring process in Fig. 6.5. Each message contains meta information (also referred as a conversation header in Chapter 3), containing the role name the message is intended for and the id of the protocol the message is a part of. When an actor joins a protocol, the FSM, generated from the Scribble compiler (as shown in Fig. 6.1) is loaded from a distributed storage to the actor memory.

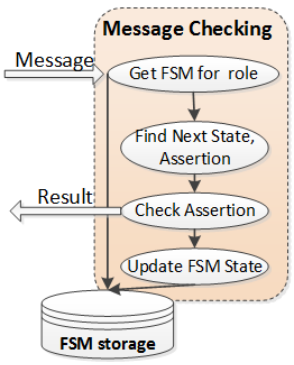


Figure 6.5.: Session Actors Monitoring

Then the checking goes through the following steps. First, depending on the role and the protocol id the matching FSM is retrieved from the actor memory. Next the FSM checks whether the message labels/operators (already a part of the actor payload) and sender and receiver roles are valid.

The *check assertions* step verifies that if any constraints on the size/value of the payload are specified in Scribble, they are also fulfilled. If a message is detected as wrong the session actor throws a distributed exception and sends an error message back to the sending role and does not pass the message to its handler for processing. This behaviour can be changed by implementing the wrong_message method of the SessionActor class.

6.4. Evaluations of Session Actors

This section reports on the performance of our framework. The goal of our evaluation is two fold. First, we compare our host distributed actor framework [Cel] with a mainstream actor library (AKKA [akk]). Second, we show that our main contribution, runtime verification of Scribble protocols, can be realised with reasonable cost. The full source code of the benchmark protocols and applications and the raw data are available from the project page [setb].

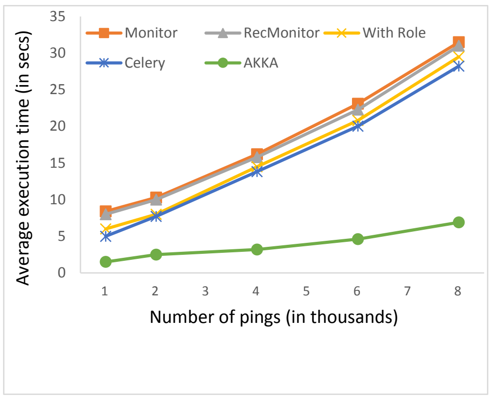
6.4.1. Session Actors Performance

We test the overhead of message delivery in our implementation using the ping-pong benchmark [IS12] in which two processes send each other messages back and forth. The original version of the code was obtained from Scala `pingpong.scala`[2] and adapted to use distributed AKKA actors (instead of local). We distinguish two protocols. Each pingpong can be a separate session (protocol FlatPingPong) or the whole iteration can be part of one recursive protocol (RecPingPong). The protocols are given in Fig. 6.6 (right). This distinction is important only to session actors, because the protocol shape has implications on checking. For AKKA actors the notion of session does not exist and therefore the two protocols have the same implementation.

Set up. We run each scenario 50 times and measured the overall execution time (the difference between successive runs was found to be negligible). The creation and population of the network was not measured as part of the execution time. The client and server actor nodes and the AMQP broker were each run on separate machines (Intel Core2 Duo 2.80 GHz, 4 GB memory, 64-bit Ubuntu 11.04, kernel 2.6.38). All hosts are interconnected through Gigabit Ethernet and latency between each node was measured to be 0.24 ms on average (ping 64 bytes). The version of Python used was 2.7, of Scala – 2.10.2 and of the AKKA-actors – 2.2.1.

Results. Fig. 6.6 (left) compares three separate benchmark measurements for session actors. The base case for comparison, annotated as ”Celery”, is a pure actor implementation without the addition of roles. ”With Role” measures the overhead of roles annotations without having the monitor capabilities enabled (without verifying communication safety). The two main cases, ”Rec Monitor” and ”Monitor”,

[2] `http://scala-lang.org/old/node/54`



```
global protocol FlatPingPong(
  role C, role S)
{
ping(str) from C to S;
pong(str) from S to C;
}


global protocol RecPingPong(
  role C, role S)
{
rec Loop{
  ping(str) from C to S;
  pong(str) from S to C;
continue Loop; }
}
```

Figure 6.6.: The PingPong benchmark, showing the overhead of message delivery (left) and The FlatPingPong and RecPingPong protocol (right)

measure the full overhead of session actors. This separation aims to clearly illustrate the overhead introduced by each of the additions, presented in this chapter: roles annotations, Scribble-based runtime verification and actor discovery. Note that the FSMs for the recursive and for the flat protocol have the same number of states. Therefore, the observed difference in the performance is a result of the cost of actor discovery.

The difference between the performance of AKKA and Celery is not surprising and can be explained by the distinct nature of the frameworks. AKKA runs on top of JVM and the remote actors in these benchmarks use direct TCP connections. On the other hand, Celery is Python-based framework which uses a message middleware as a transport, which adds additional layer of indirection. Given these differences, Celery actors have reasonable performance and are a viable alternative.

Regarding our additions we can draw several positive conclusions from the benchmarks: (1) the cost of the FSM checking is negligible and largely overshadowed by the communication cost (such as latency and routing); and (2) the cost of the mechanism for actor discovery is reasonable, given the protocol load.

States	100	1000	10000
Time (ms)	0.0479	0.0501	0.0569

Table 6.1.: Execution Time for checking protocol with increasing length

6.4.2. Monitoring Overhead

In this subsection, we discuss the row overhead of monitor checking, which is a main factor that can affect the performance and scalability of our implementation. Knowing the complexity of checking protocols of different length is useful to reason about the applicability of the model.

We have applied two optimisations to the monitor, presented in Chapter 3, which significantly reduce the runtime overhead. First, we pre-generate the FSM based on the global protocol. Second, we cache the FSM protocols the first time when a role is loaded. Therefore, the slight runtime overhead (up to 10%), observed in the previous section is a result of the time required to choose the correct FSM among all in-memory FSMs inside the actor and the time required to check the FSM transition table that the message is expected. The former has a linear complexity on the number of roles running inside an actor (the runtime searches through all in-memory FSMs) and the latter has a linear complexity on the number of states inside an FSM (the monitor performs a search for a matching transition among all FSM transitions).

Table 6.1 shows the approximate execution time for checking protocols of increasing length of interactions. The protocol for testing is the request-reply protocol, given below:

```
msg1() from A to B; msg2()from B to A; ... msg_n() from B to A)
```

We evaluate the above protocol for *n* ranging over 100, 1000 and 10 000 respectively. Then we report the time taken to complete each of the protocols. We omit results for *n* smaller than 100 since the execution time is negligible.

6.4.3. Applications of Session Actors

As a practical evaluation of our framework, we have implemented two popular actor applications and adapted them to session actors.

The first application is a distributed chat with multiple chat rooms, where a number of clients connect to a reactive server and execute operations based on their privileges. Depending on privileges some clients might have extended number of allowed messages. We impose this restriction dynamically by annotating the restricted operations with different roles. An operation is allowed only if its actor has already joined a session in the annotated role.

The second use case is a general MapReduce for counting words in a document. It is an adaptation from an example presented in the official website for celery actors[3]. The session actor usage removes the requirement for the manual actor spawning of Reducers actors inside the Mapper actor, which reduces the code size and the complexity of the implementation.

Listing 6.2: The WordCount protocol and its implementation in Python

```
#=======Code For The Mapper Role=========
@protocol(c, WordCount, R, A, M)
class Mapper(Actor):
  @role(c)    # invoked on protocol start
  def join(self, c, file, n):
    self.count_document(self,file, n)

  @role(c, self)
  def count_document(self, c, file, n):
    with open(file) as f:
      lines = f.readlines(), count = 0
      for line in lines:
        reducer = c.R[count%n], count+=1
        reducer.send.count_lines(line)

#=======Code For Creating a protocol=======
c = Protocol.create(WordCount, {M=Mapper,
  A=Aggregator}, {R:(Reducer, 10)})
c.M.send.join(file="file1.txt", n=10)

#=======Code For Aggregator role===========
@protocol(c, WordCount, A, M, R)
class Aggregator(Actor):
  @role(c, master)
  def aggregate(self, c, words):
  for word, n in words.iteritems():
    self.result.setdefault(word, 0)
    self.result[word] += n
    # when a treshhold is reached
```

[3]http://cell.readthedocs.org/en/latest/getting-started/index.html

```
30    # print the results
31   c.close()
32
33 #=======Code For Reducer role==============
34 @protocol(c, WordCount, R, A, M)
35 class Reducer(Actor):
36   @role(c, M)
37   def count_lines(self, c, line):
38   words = {}
39     for word in line.split(" "):
40     words.setdefault(word, 0)
41     words[word] += 1
42     c.A.send.aggregate(words)
```

We give in Listing 6.2 the implementation of the latter example. For sake of space and clarity, the implementation reported here abstracts from technical details that are not important for the scope of this work. The protocol is started on line 26 specifying the type of the actors for each role and passing the arguments for the initial execution – `n` (the number of reducers) and `file` (the name of the source file to be counted). The `Mapper` implements the `join` method, which is invoked when the actor joins a protocol. Since the `Mapper` is the starting role, when it joins, it sends a message to itself to proceed with the execution of `count_document`. Note that spawning and linking of actors are not specified in the code because the actor discovery mechanism accounts for it. The protocol proceeds by `Mapper` sending one message for each line of the document. The receiver of each message is one of the `Reducers`. Each `Reducer` counts the words in its line and sends the result to the `Aggregator` (`c.A`, line 48), which stores all words in a dictionary and aggregates them. When a threshold for the result is reached, the `Aggregator` prints the result and stops the session explicitly.

Our experiences with session actors are promising. Although they introduce new notions, i.e. a protocol and a role, we found that their addition to a class-based actor framework is natural to integrate without requiring a radical new way of thinking. The protocol and role annotations are matched well with typical actor applications, and they even result in simplifying the code by removing the boiler-plate for actor discovery and coordination. The protocol-oriented approach to actor programming accounts for the early error detection in the application design and the coordination of actors. The runtime verification guarantees and enforces the correct order of interactions, which is normally ensured only by hours of testing.

No	Use Case Name	MPST feature required	paper
1	Ping Pong	standard	[HMB+11]
2	Counting Actor	standard	[NYH13]
3	Fibonacci	recursive subsession with dynamic role creation	[DH12]
4	Big	parallel; join	[HMB+11]
5	Thread Ring	Pabble (parameterised MPST)	[NY14c]
6	KFork (throughput)	subsessions	[DH12]
7	Dining Philosophers	parameterised parallel; group role	new
8	Sleeping Barber	parameterised parallel; group role	new
9	Logistic Map Series	parameterised parallel with join	new
10	Bank Transaction	parallel composition with join	new
11	Cigarette Smoker	parameterised parallel; group role	new

Table 6.2.: List of implemented Savina benchmarks

6.5. Use Cases

In this section we evaluate the applicability of our framework to verify use cases which contain common communication patterns found in actor languages. We select the use cases from an actor benchmark suit, Savina [IS14]. Savina aims to identify a representative set of actor applications. The use cases range from popular micro benchmarks to classic concurrency problems. Each benchmark tests a different aspect of the actor implementation (e.g. actor creation, throughput, mailbox contention) and presents a different interaction patterns (e.g. master-slave, round-robin communication, synchronous request-reply).

We express twelve out of the sixteen benchmarks, presented in the micro and concurrency benchmarks sections of the suit. Standard primitives for multiparty session types (recursion, choice, point-to-point interaction and parallel composition) are not sufficient to specify most of the use cases, but more advanced features such as assertions [BHTY10], subsessions [DH12] and dynamic role creation [DY11] leverage the expressive capabilities of Scribble. The required extensions are already proposed in the theoretical literature of MPSTs, and can be expressed using combinations of various theoretical formalisms. Here we (1) identify (a combination of) the MPST extensions needed to express actors patterns; (2) specify them in Scribble; and (3) implement end-point programs in Python.

First, we explain the advanced Scribble features, pointed in Table 6.2. In the table, the *parameterised parallel composition* stands for the use of the construct `par [i:1..N] {body}`, which is a shorthand for `N` parallel `body` branches, where `i` is substituted with values from `1` to `N` inside the `body`. When the `par` is followed by another block as in Listing 6.5, we call this block a join block (denoted as *parallel with join* in Table 6.2). The sequence `par {block 1} and {block 2} block 3` means that for `block3` to be executed, `block1` and `blcok2` should be completed. For example, in Listing 6.5 the message transfer `done() from S to A;` can be completed only after both `A` and `B` have sent a message `done` to `S`.

Another Scribble feature exploited in our examples is a *subsession*. It is used to create modular Scribble code and is realised as a macro expansion. It is often combined with *a group role* (Table 6.2, examples 7, 8, 11). A group role, for example, `role Philosophers[1..N]`, specifies that there will be `N` roles with the same type in the protocol. The roles share the same implementation, but at runtime there will be `N` instances of that role and their behaviour might depend on initially assigned indices.

Next we present the examples. We first describe a global Scribble protocol for each use case based on specifications in [IS14] and the implementations on github [Sav], and then provide a couple of implementations in Python in Appendix B. We also demonstrate faulty executions, that can be detected by our framework, on the Sleeping barber example in Fig. 6.7. The use cases in Scribble and corresponding Python code can be found in [setb].

Table 6.2 shows a summary, specifying the communication pattern and the MPST extension (if needed). For each use case, we give a brief description and the global Scribble protocol. The previous version of Scribble has to be extended with (1) parametrised parallel composition and (2) subsession with dynamic spawning of roles [DH12].

Listing 6.3: the Counting Protocol

```
global protocol Counter (role P, role C)
{
  rec Loop { value(int) from P to C;}
  result(int) from C to P;}
}
```

Counting Actor. This use case expresses an elaborated request-reply pattern, measuring the time taken to send *n* consecutive messages from a `Producer` actor to a `Counter` actor. The accumulated value in the `Counter` is returned to the `Producer`. From an actor perspective, the use case tests the mail box implementation which needs to store the pending messages yet to be processed. Using session types the communication interaction is succinctly specified with recursion and point-to-point communication, expressing the causalities between sending of a value (`val`) and the final reply (`res`).

Listing 6.4: the Fibonacci Protocol

```
global protocol Fib(role P, role C)
{
  request(int) from P to C;
  par { Fib(C as P, new C);}
  and { Fib(C as P, new C);}
  result(int) from C to P; }}
}
```

Fibonacci. This use case involves the recursive creation of a predefined number of actors using the naive recursive formula for computing fibonacci numbers. The idea of the protocol is that an actor receives a request to compute a value, then it spawns two additional actors and waits for their response. Finally, it returns the sum of the responses to the requester. The example tests the actor creation/destruction in the actor implementation. The global Scribble protocol is given in Listing 6.4. After receiving a request (`req`) from a `Parent` actor, a `Child` actor invokes the same protocol type (`Fib`) with a new `Child` actor. To express the new role creation, we use nested protocols (thereafter called subprotocols), formally presented in [DH12]. A subprotocol definition accepts either roles already active in the protocol (`C as Parent`) or new roles of a known type (`new Child`). To express the fork-join behaviour required by the example we use parallel composition with continuation (the actions after a `par` in a protocol body). Thus the specified protocol imposes causality between sending the response (`res`) and the completion of the proceeding interactions, but no causality between the subprotocols in the parallel branches.

Listing 6.5: The Big protocol

```
global protocol Big (role A,
  role B,role Sink as S)
{
  par { ping() from A to B;
        pong() from B to A;
        done() from A to S;}
  and { ping() from B to A;
        pong() from A to B;
        done() from B to S;}
  done() from S to A;
  done() from S to B;
}
```

Big. The use case presents a many-to-many message passing scenario. Several actors are spawned, each of which sends a ping message to another random actor. Then the recipient actor responds with a pong message. A Sink actor is notified (via done message) about the completion of these interactions. When the Sink receives all replies, it terminates the other actors. In an actor implementation, the benchmark measures the effects on contention on a recipient's actor mailbox. In session types the use case requires a join primitive, expressed in Scribble as a continuation after the parallel branches, similarly to the Fibonacci use case. Note that the round robin sending cannot be precisely expressed in Scribble. We have simplified the use case, giving only the type for two ping-pong actors (A and B in Listing 6.5) and one Sink actor. The protocol can be extended for more ping-pong actors if the configuration is fixed in advance.

Listing 6.6: The ThreadRing protocol

```
global protocol Ring(role Worker[1..N])
{
  rec Loop {
    data(int) from Worker[i:1..N−1] to Worker[i+1];
    data(int) from Worker[N] to Worker[1];
  continue Loop;}
}
```

Thread Ring. In the ThreadRing use case an integer token message is passed around a ring of N connected actors. The token value is decremented each time an actor receives it. The interactions continue until the token becomes zero. From an actor perspective, the benchmark tests the raw efficiency of message sending since it removes all contention of resources except between the sending and receiving actors. To express the protocol for each N ring configuration we use an extension of Scribble, called Pabble [NY14c], where multiple participants can be grouped in the same role and indexed. Indeed, the same use case is already presented in [NY14c]. In the protocol in Listing 6.6, Worker[i:1..N−1] declares iteration from 1 to N−1, in this sequence, on the interaction (Data(int) from Worker[i] to Worker[i+1]) over the bound variable i.

Listing 6.7: The KFork protocol

```
global protocol KFork(role master as M, role Worker[1..K])
{
  par[i:1..K] {
    rec Loop {
      choice at M {
        data(int) from M to Worker[i];}
      or {
        end from M to Worker[i];}}
}
```

KFork. The application first creates K worker actors and then sends each actor a total of *n* messages in a round robin manner. Actors terminate after processing all messages. The benchmark tests the message throughput in the implementation. The global multiparty protocol is expressed as a parallel composition of K binary protocols with an identical communication pattern, recursive interaction between a Master and a Worker. For more concise specification, since all parallel branches describe an identical communication pattern, we leverage the parallel construct by introducing indexing on parallel branches. The syntax `par[i:1..N] {body}` defines replication of body for each value of i between 1 and K. Listing 6.8 shows the global protocol for KFork using the new syntax.

Listing 6.8: Refactored KFork Protocol

```
global protocol KFork(role Master as M, role Worker[1..K])
{
  par[i:1..K] { Sub(M, W[i]);}
}
global protocol Sub(role M, role W)
{
  rec Loop {
    choice at M {
      data(int) from M to W;}
    or {
      end from M to W;}}
}
```

Session patterns for concurrent group execution (Refactoring KFork). We refactor the KFork global protocol, extracting the `par` body into a subprotocol. The global protocol with a group role is represented as a parallel composition of subprotocols. Indeed, this is an emerging pattern in all concurrent problems surveyed in the benchmark suit. The pattern appears in four use cases: Sleeping Barber, LogisticMap, BankTransaction and DiningPhilosophers.

Listing 6.9: Dining Philosophers, Bank Transaction, Logistic Map

```
global protocol DiningPhilosophers(
  role arbitrator as A, role Philosopher[1..N])
{
  par[i:1..n] { DiningPhilosopher(A, Philosopher[i]); }
}
global protocol BankTransactions(
  role Teller as T, role Accounts[1..N])
{
  par[i:1..n−1] { BankTransaction(T, A[i], A[i+1]); }
}
global protocol LogisticMap(
  role Master as M, role Series[1..N], role Ratio[1..N])
{
  request from M to S[1..N];
  par[i:1..N] { SeriesMaster(S[i], R[i]);}
  response from S [1..N] to M;
  stop from M to S[1..N];
}
```

A global protocol with a group role is split into smaller multiparty protocols - one for each role in the group role set. In addition, in LogisticMap and SleepingBarber a join multicast is required to signal the completion of the interaction and to end the protocol. In Listing 6.9 we give the four global protocols, where the recurring session pattern can be clearly observed. In the rest of the section we focus on the interactions between one of the roles from the group role set and give the protocols (DiningPhilosopher, SleepingBarberOneCustomer, BankTransaction, SeriesMaster respectively) which do not require the use of a group role.

Listing 6.10: The DiningPhilosophers protocol

```
global protocol DiningPhilosopher(
  role Arbitrator as A, role Philosopher as Ph)
{
  rec L { choice at Ph {
          rec M {
            req() from Ph to A;
              choice at A {
                yes() from A to Ph;
                continue L;}
              or {
                no() from A to Ph;
                continue M;}}}
```

```
                    or { done() from Ph to A;}}
        }
```

The Dining Philosophers. In this classic example, `N` philosophers are competing to eat while sitting around a round table. There is one fork between each two philosophers and a philosopher needs two forks to eat. The challenge is to avoid deadlock, where no one can eat, because everyone is possessing exactly one fork. In Savina, the problem is implemented using `Arbitrator` actor that coordinates the resources (the forks) between `Philosopher` actors. The `Arbitrator` either accepts (`yes()`) a `Philosopher` request or rejects (`no()`) it, depending on the current resources allocated. If rejected, the `Philosopher` keeps sending request (`req`) until it is accepted. The global protocol in Listing 6.10 specifies the interactions between an `Arbitrator` and one `Philosopher`, defining the causalities using recursion and choice.

Sleeping Barber. The use case presents another classic concurrency problem. A barber is waiting for customers to cut their hair. When there are no customers the barber sleeps. When a customer arrives, he wakes the barber or sits in one of the waiting chairs in the waiting room. If all chairs are occupied, the customer leaves. Customers repeatedly visit the barber shop until they get a haircut. The key element of the solution is to make sure that whenever a customer or a barber checks the state of the waiting room, they always see a valid state. The problem is implemented using a `Selector` actor that decides which is the next customer, `Customer` actors, a `Barber` actor and a `Room` actor. Note that, differently from the previous use cases, the protocol between the `Customers` and the other actors is multiparty (not binary). To guarantee a valid synchronisation, precise message sequence should be followed. While the use cases so far demonstrated succinct recursion and request-reply patterns, interleaved together, this one demonstrates a communication structure with long sequence of interactions, where preserving the causalities is the key to achieving deadlock free communication.

The implementation is ∼200 LOC in Akka (∼100 LOC in Python), where each of the four actors sends messages to the other three actors. The implementation is error-prone, because the type of the actors and whether they can be safely compose is not obvious. Sending the wrong message type, sending to the wrong actor or not sending in the correct message sequence may lead to deadlocks, errors which initial cause is hard to be identified or wrong computation results. Errors can even go unnoticed resulting in orphan actors. Here we give examples of several faulty executions and explain how they are prevented by our framework (for every error example, we highlight in Fig. 6.7 a wrong implementation of a message transfer and the corresponding line in the global protocol):

Deadlock (line 10): After receiving message `full` the Customer `C` sends a message `returned` to the Room `R` instead of sending it to the Selector `S`. Then the `C` will wait to be reentered again, while `S` will wait to receive `returned` or `done` from `C` (according to Line 17), resulting in a deadlock. Moreover, the other actors are stuck as well (`R` is waiting on `S`; `B` is waiting on `R`). Our framework will detect that the message `returned` is sent to the wrong actor.

Unexpected termination (line 17): In this scenario, `C` again sends a wrong message to `R`, but contrary to the above example, it sends a message that `R` can handle (the message `done`). If there is no verification at `R` on the sender of the messages, `R` will execute the `done` handler and as a result will terminate (`done` should be sent by `S` to terminate the actor `R` when all customers have been processed). When the Barber `B` tries to send `next` message to `R` it will get a runtime exception that the actor `R` does not exist. Such an exception can be very misleading because it points to a problem between the Barber and the Room, while the actual error is in the code for the Customer. Our framework will detect this inconsistency at the right (at role `C`) place therefore will not allow the error to be propagated, masking the real cause.

Orphan actors (line 24): The actor `R` sends a wrong message (`don` instead of `done`) to `B` (or doesn't send a message at all). Therefore, `B` will not terminate. However, everybody else will and so will the protocol, leaving an orphan actor `B`. Our monitor will detect the wrong message that `B` is sending.

Listing 6.11: The CigaretteSmoker protocol

```
global protocol CigaretteSmoker(role Arbiter as A,
  role Smoker[1..N] as S)
{
  rec Loop {
    choice at A {
      start_smoking() from A to S[i];
      started_smoking() from S[i] to A;
      continue Loop;
    } or {
      exit() from A to S[i..n]; }}}
```

```
global protocol SleepingBarber(role barber as B,
role selector as S, role room as R)
{rec L {
    choice at S {
    par {
    S introduces C;}
    enter() from S to R;
    choice at R {
      full() from R to C;
      returned() from C to S;
      continue L; }
    } or {
    wait() from R to C;
       enter() from R to B;
       start() from B to C;
       stop() from B to C;
       done() from C to S;
       next() from B to R;
    }
      wait() from R to B;
      continue L;
    } or {
      done() from S to R;
      done() from R to B;   }}
```

```
# Deadlock
class CustomerActor...
  @role(c, S)
  def full(self, c):
    # c.S.send.returned()
    c.R.send.returned()

...
  @role(c, B)
  def stop(self, c):
    # c.S.send.done()
    c.R.send.done()

# Orphan Actors
class RoomActor...
    @role(c, S)
  def done(self, c):
    # c.B.send.done()
    c.B.send.don()
```

Figure 6.7.: The Sleeping Barber protocol (left) and corresponding faulty implementations (right)

Cigarette Smoker. This problem involves `N` smokers and one arbiter. The arbiter puts resources on the table and smokers pick them. A smoker needs to collect *k* resources to start smoking. The challenge is to avoid deadlock by disallowing a competition between smokers from picking up resources. This is done by delegating the control to the arbiter, who decides (in a random manner) which smoker to send the resource to. The session type for the use case, given in Listing 6.11, combines round-robin pattern, sending a random smoker a message to smoke (`StartSmoking`), with multicast, iterating through all the smokers notifying them to exit (`Exit from A to S[i..n];`).

Listing 6.12: The BankTransaction protocol

```
global protocol BankTransaction(role Teller as T,
  role SrcAcc as S, role DestAcc as D)
{
    credit from T to S;
    debit from S to D;
    reply from D to S;
    reply from S to T;
}
```

Bank Transaction. The communication pattern in this use case is a synchronous request-reply chain. Three actors (a `Teller`, a source account `SrcAcc` and a destination account `DestAcc`) synchronise to perform a debit from one account to another. The `Teller` is not allowed other operations until a confirmation that the transaction is complete is received from the source account. The two synchronous request-reply patterns are expressed as a message sequence in the BankTransaction protocol in Listing 6.12.

Logistic Map Series. This is another example of synchronous request-reply pattern. The goal is to calculate a recurrence relation: $x_{n+1} = rx_n(1-x_n)$. The implementation requires an actor for a manager (`Master`), a set of term actors (`Series`) and a set of ratio actors (`Ratio`). The `Ratio` actors encapsulate the ratio r and compute the next term x_{n+1} from the current term x_n. The `Master` sends request to the `Series` actor to compute a result. Then `Series` actors delegate the calculation to a `Ratio` actor and wait for synchronous reply to update their value of x. Thus, the MasterSeries subprotocol is an instance of a request-reply pattern (from `Series` as a requester to `Ratio` role as a replier), which we have already shows and thus we omit.

Listing 6.13: The ProducerConsumer protocol

```
global protocol PCBounded(
  role Buffer as B,
  role Producer as P,
  role Consumer as C)
{
rec Loop {
  produce() from B to P;
  par {
    choice at P {
      dm() from P to B;
      continue Loop;
    } or {
      exit() from P to B;
    }
  } and {
    dm() from B to C;
    more() from C to B;}}
}
```

Producer Consumer with Bounded Buffer. The overall scenario of this use case is that producers push work into the buffers, while consumers pull from it. The implementation uses actors for producers, consumers and a buffer. The `Buffer` actor is concurrently receiving messages from both `Producers` and `Consumers`.

Thus, the benchmark measures also the contention overhead of the actor implementation. The Scribble protocol is given in Listing 6.13. To represent the pattern (producers are sending messages concurrently, but there is causality between the message send from a `Producer` and the message send to the `Consumer`). We use a combination of recursion and parallel branches. In each recursive execution, we require `Consumers` and `Producers` to send or receive concurrently. At the same time, the `Producer` can keep sending (starting a new recursive execution), before the `Consumer` has finished their branch.

6.6. Related Work

Behavioural and session types for actors and objects There are several theoretical works that have studied behavioural types for verifying actors [MV11, Cra]. The work [Cra] proposes a behavioural typing system for an actor calculus where a type describes a sequence of inputs and outputs performed by the actor body. In [MV11], a concurrent fragment of Erlang is enriched with sessions and session types. Messages are linked to a session via correlation sets (explicit identifiers, contained in the message), and clients create the required references and send them in the service invocation message. The developed typing system guarantees that all within-session messages have a chance of being received. The formalism is based on only binary sessions.

Several recent papers have combined session types, as specification of protocols on communicating channels, with object-oriented paradigm. A work close to ours is [GVR$^+$10], where a session channel is stored as a field of an object, therefore channels can be accessed from different methods. They explicitly specify the (partial) type for each method bound to a channel. Our implementation also allows modularised sessions based on channel-based communication. Their work [GVR$^+$10] is mainly theoretical and gives a static typing system based on binary session types. Our work aims to provide a design and implementation of runtime verification based on multiparty session types, and is integrated with existing development frameworks based on Celery [Cel] and AMQP [AMQ].

The work in [Cai08] formalises behaviours of non-uniform active objects where the set of available methods may change dynamically. It uses an approach based on spatial logic for a fine grained access control of resources. Method availability in [Cai08] depends on the state of the object in a similar way as ours.

Other Actor frameworks The most popular actor's library (the AKKA framework [akk] in Scala studied in [TDJ13]) supports FSM verification mechanism (through inheritance) and typed channels. Their channel typing is simple so that it cannot capture structures of communications such as sequencing, branching or recursions. These structures ensured by session types are a key element for guaranteeing deadlock freedom between multiple actors. In addition, in [akk], channels and FSMs are unrelated and cannot be intermixed; on the other hand, in our approach, we rely on external specifications based on the choreography (MPST) and the FSMs usage is internalised (i.e. FSMs are automatically generated from a global type), therefore it does not affect program structures.

Several works study an extension of actors in multicore control flows. Multi-threaded active objects [HHI13] allow several threads to co-exist inside an active object and provide an annotation system to control their concurrent executions. Parallel Actor Monitors (PAM) [STM14] is another framework for intra actor parallelism. PAMs are monitors attached to each actor that schedule the execution of messages based on synchronisation constraints. Our framework also enables multiple control flows inside an actor as [HHI13, STM14]. In addition, we embed monitors inside actors in a similar way as [STM14, HHI13] embed schedulers. The main focus of the monitors in [HHI13, STM14] is scheduling the order of the method executions in order to optimise the actor performance on multi-core machines, while our approach aims to provide explicit protocol specifications and verification among interactions between distributed actors in multi-node environments.

The work [SPH10] proposes a technique to have multiple cooperatively scheduled tasks within a single active object similar to our notion of cooperative roles within an actor. The approach is implemented as a Java extension with an actor-like concurrency where communication is based on asynchronous method calls with objects as targets. They resort to RMI for writing distributed implementations and do not allow specifying sequencing of messages (protocols) like ours.

The work [RYC$^+$06] proposes a three-layered architecture of actor roles and coordinators, which resemble roles and protocol mailboxes, respectively, in our setting. Their specifications focus on QoS requirements, while our aim is to describe and ensure correct patterns of interactions (message passing).

Comparing with the above works, our aim is to provide effective framework for multi-node environments where actors can be distributed transparently to different machines and/or cores.

Several extensions of the actor model suggest more complex synchronization of the actor inbox such as multi-headed messages with guards [SLVW08], join-style actors [FG00] and synchronous replies [VA01]. These works focus on more expressive pattern matching for precise and correct dispatch of the messages to the corresponding message handlers implemented in an actor. Thus, the synchronisation constraints are specified on the local actor behaviour. The emphasize of our work is on expressing synchronization constraints globally, as part of the overall choreography of an actor system, and inferring (through projection) the local behaviour of a single actor.

6.7. Concluding Remarks

We propose an actor specification and verification framework based on multiparty session types, providing a Python library with actor communication primitives. Cooperative multitasking within sessions allows for combining active and reactive behaviours in a simple and type-safe way. The framework is naturally integrated with Celery [Cel] which uses advanced message queue protocol (AMQP [AMQ]), and uses effectively its types for realising key mechanisms such as actor discovery. We demonstrated that the overhead of our implementation is very small. We then show that programming in session actors is straightforward by implementing and rewriting use cases from [akk]. The work presented in this chapter links FSMs, actors and choreographies in a single framework. As actor languages and frameworks are becoming more popular, we believe that our work would offer an important step for writing correct large-scale actor-based communication programs.

7. Recoverability for Monitored MPST processes

Safety-critical distributed systems need to run *all the time*, i.e. even in the presence of failure, the overall system should stay alive. Despite the importance of fast and correct recovery in such systems, it is still difficult and error-prone to implement fault-tolerant distributed components.

Let it Crash recovery model. A popular fault-tolerance model is the *Let it crash* model, adopted by the programming language Erlang. In Erlang, rather than trying to handle and recover from all possible exceptional and failure states, one can instead let processes crash and let the runtime automatically recycle them back to their initial state. Failures and error propagation are managed by organising the running processes in a hierarchical structure, called a *supervision* tree, where each process and its dependencies are monitored by a parent process, called *supervisor*.

While the *Let it crash model* has gained popularity and has recently been adopted in other commercial languages and frameworks (Go, Akka/Scala), it still faces two major technical problems. First, the correctness of recovery strategies relies on the assumption that a global process structure (i.e. a supervision structure) of the system is correctly written by a programmer. A recent study reveals a misconfiguration of the supervision structures is a common source of errors for recovery [Nys09b]. Second, supervision strategies are statically given; hence they often recover too few or too many processes, and are unable to capture the dynamic nature of the communication dependencies between processes. This often leads to a redundant and/or unsound recovery mechanism (as shown and explained in Table 7.1).

In this chapter we propose a novel recovery strategy, tackling the challenges mentioned above. The strategy is realised as part of a Scribble-based runtime verification framework in Erlang, capacitated not only to ensure safety of monitored processes, but also to optimise the communication overhead during recovery after a process failure.

$$G = \begin{pmatrix} \mathtt{A}_1 \to \mathtt{A}_2 \\ \mathtt{A}_2 \to \mathtt{A}_3 \\ \cdots \\ \mathtt{A}_n \to \mathtt{C} \end{pmatrix} \mid \begin{pmatrix} \mathtt{B}_1 \to \mathtt{B}_2 \\ \mathtt{B}_2 \to \mathtt{B}_3 \\ \cdots \\ \mathtt{B}_n \to \mathtt{D} \end{pmatrix}$$

$$\mathtt{C} \to \mathtt{E}; \mathtt{D} \to \mathtt{E};$$

$$\mathtt{E} \to \mathtt{C} : \begin{cases} \mathsf{accept}.\mathtt{E} \to \mathtt{D} : \mathsf{reject} \\ \mathsf{reject}.\mathtt{E} \to \mathtt{D} : \mathsf{accept} \end{cases}$$

```
%% CODE FOR A, B, C, D
handle_cast(A, msg)
  A1 = get_next()%failure 1
  A1 ! msg.
handle_cast(A, accept)
  io:format('accept').
%% CODE E
init(_,msg)
  %failure 2
  receive {C, msg1}→
    io:format(msg1/msg2)
      receive {D, msg2} →
        io:format(msg1/msg2)
        C ! accept,
        D ! reject.%failure 3
```

Figure 7.1.: Trading Negotiation Global Protocol (left) and Erlang Implementation (right)

Our approach. Our recovery strategy relies on two design ideas: (1) messages that should be resent are identified based on the dependencies in the type structure of a process and (2) only affected participants are notified and recovered. To realise these ideas we propose a novel algorithm, which analyses the communication flow, given as a multiparty protocol, to extract the causal dependencies between processes. The algorithm works by traversing a *dependency graph*, automatically inferred from a given global protocol. We calculate all dependencies in the graph, affected by a given state. On failure, the processes from the calculated paths are recovered. Thus, only a minimal amount of data is propagated, that is, the global point of failure. Although the algorithms is static, we use runtime monitors to track process states and invoke the recovery procedure when a process crash is detected.

Trading negotiation example. We start by illustrating the difficulty of sound and efficient recovery on chained parallel interactions which is a common topology found in Erlang applications. Fig. 7.1 (right) shows an Erlang program for a trading negotiation protocol, while Fig. 7.1 (left) shows the protocol written as a global multiparty session type [1]. The notation $\mathtt{A}_\mathtt{i} \to \mathtt{A}_{\mathtt{i}+1}$ denotes a message exchange from participant $\mathtt{A}_\mathtt{i}$ to participant $\mathtt{A}_{\mathtt{i}+1}$, and $G \mid G'$ denotes a parallel execution of protocol G and protocol G'. A sequence of messages is denoted by semicolon ";" and $\{\mathsf{label}_1 : G_1,\ \mathsf{label}_2 : G_2\}$ represents a choice of G_1 or G_2 with label label_1 or label_2. Messages are ordered on the sender and receiver side. For example,

[1]We use the syntax of global types as given in Chapter 2 with addition of ; and we omit the label of the message when a branch is singleton.

$\mathtt{C} \to \mathtt{E}; \mathtt{D} \to \mathtt{E}$ describes messages to be processed by E in the specified order.

The trading negotiation process is split into three phases: In the first phase, two groups of participants, the team of Alice ($\mathtt{A}_i$) and the team of Bob ($\mathtt{B}_i$) forward messages to their group leaders, C and D respectively, with their suggested trading quote. The communication between the groups is independent (represented by a parallel composition of two global protocols). In the second phase, the leaders of the groups notify the trader E regarding their proposals (represented by $\mathtt{C} \to \mathtt{E}; \mathtt{D} \to \mathtt{E}$). Finally, E chooses the best quota and sends accept or reject to each group leader (represented by a choice). Note that since the parallel compositions end with participants C and D, respectively, there is the ordering from the chain of $\mathtt{A_i}$ to $\mathtt{E} \to \mathtt{C}$ and from the chain of $\mathtt{B_j}$ to $\mathtt{D} \to \mathtt{C}$, respectively.

In the above protocol the dependencies between the processes are dynamic and change with the execution of the protocol. We consider, as examples, three possible failures and give the set of affected participants in each case using our MPST-induced approach. Table 7.1 summarises the result and compare it with two Erlang recovery strategies.

Scenario 1: $\mathtt{A_i}$ fails before sending the message to $\mathtt{A_{i+1}}$ (line 3) $\mathtt{A_i}$ affects only its predecessors. Thus, for example, if $\mathtt{A_3}$ fails, only $\mathtt{A_1}$ and $\mathtt{A_2}$ are restarted. (Line 10).

Scenario 2: E fails at receiving the message from C (line 10) All participants are restarted. Note that since the network is asynchronous C might have already sent the message to E, although E might not have selected the message from the queue. When E fails, the queue will be erased and the messages will be lost. Thus, inevitably C must be informed about E's failure.

Scenario 3: E fails before sending the message to D (Line 15). The protocol prescribes that E has already received the quota from Alice's group. Therefore, only D should resend its quota. We recover everyone on the path $\mathtt{B_1}, \ldots, \mathtt{B_n}, \mathtt{D}$.

Erlang recovery strategies. In Erlang, there are three types of supervisions. In *one-for-one* supervision, if a worker fails, only this worker is restarted by the supervisor; in *all-for-one* supervision, if any worker dies, all the workers are restarted; and in *rest-for-one* supervision, if a worker terminates, only the rest of the workers (the processes followed by the terminated process) are terminated.

In Fig. 7.2 we present two representative supervision hierarchies with combined strategies and compare them with our approach in Table 7.1.

The rest-for-one supervision is suitable for disjoint groups of chained inter-

	Fig. 7.2 (a) restarts	Fig. 7.2 (b) restarts	Our approach
Scenario 1	$\mathtt{A}_1 \ldots \mathtt{A}_\mathtt{i}$	$\mathtt{A}_1 \ldots \mathtt{A}_\mathtt{i}$	$\mathtt{A}_1 \ldots \mathtt{A}_\mathtt{i}$
Scenario 2	everyone	only E (unsound)	everyone
Scenario 3	everyone (inefficient)	only E (unsound)	$\mathtt{B}_1 \ldots \mathtt{B}_\mathtt{n}, \mathtt{D}, \mathtt{E}$

Table 7.1.: Comparison between Erlang recovery and MPST-induced recovery

actions. Hence we connect Alice's group $\mathtt{A}_i$ by a rest-for-one supervisor and Bob's group $\mathtt{B}_i$ by another supervisor of the same type. Then a worker E is grouped with the rest by all-for-one supervisor (Fig. 7.2(a)) or one-for-one supervisor (Fig. 7.2(b)). The recovery driven by (a) or (b) is, as explained below, either inefficient or unsound.

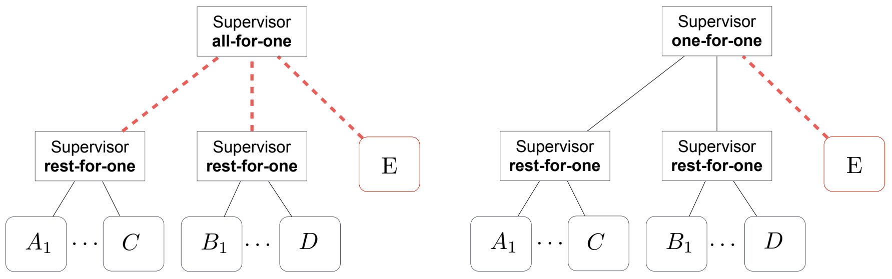


Figure 7.2.: Erlang supervision strategies: (a) all-for-one and (b) one-for-one

Suppose E dies as assumed in Scenario 2. If we chose the supervision Fig. 7.2(b), only E is restarted from the beginning. However, since C has already sent the message to E, the message by others will be lost and E will be stuck (deadlock). Consider that we chose Fig. 7.2(a). Then all processes are restarted from the beginning, which works in Scenario 2. However, in Scenario 3, $\mathtt{A}_\mathtt{i}$'s group has already completed the negotiation and thus it is redundant to repeat all interactions.

To summarise, the advantages of our approach are three-fold. First, executions ensured by our protocol recovery strategy are safe by construction. While supervision trees are manually built, and recovery strategies are sometimes implemented in an ad-hoc manner, our proposed recovery algorithm generates supervision structures automatically and is grounded on a sound theoretical foundations. Second, our strategy guarantees small computation cost resulting in reduced overall execution times. Finally, we reduce the communication cost by notifying only the relevant participants. As shown in Table 7.1 part of the error propagation can be avoided when our restart strategy is employed.

We provide a prototype implementation in Erlang on top of Erlang's built-in fault-tolerance semantics. Though we assume some conditions from Erlang such as asynchronous messages passing and ordered queues (see details in Section 7.1), our approach is general and our recovery algorithm is language-independent: as explained in Section 7.2, our algorithm can be flexibly tuned to various assumptions on queues and messaging semantics.

Contributions and outline. We outline the structure of this chapter, summarising the contributions of each part.

Section 7.1 We propose a recovery framework based on MPSTs and show how session type-based analysis is used to provide a fault-tolerant recovery.

Section 7.2 We formalise a recovery strategy (using communicating automata [BZ83, LTY15, DY12]) and we give semantics for recoverable properties.

Section 7.4 We provide a design and implementation of our recovery strategy on top of runtime monitoring in Erlang. More precisely, we adapt the Scribble-based runtime verification for actor programs, presented in Chapter 6, to Erlang processes and facilitate it with recovery capabilities.

Section 7.5 We implement several use cases and show that our recovery strategy is more efficient for common message-passing protocols when compared to the Erlang *all-for-one* supervision.

7.1. Overview of Multiparty Protocol-Induced Recovery

Our framework involves two stages: processing a given global protocol (type) and runtime supervision. The global type analysis from the first stage is used for runtime monitoring and recovering during the second stage.

7.1.1. Global Protocol Processing

The global type processing involves (1) creating a *global recovery table* (GRT) from a given global protocol (type) and (2) projecting the global protocol into local types and creating a finite state machine per each local type. Each local type corresponds to a local protocol of each participant. The GRT prescribes which part(s) of the global protocol to be recovered on failure, and the set of processes to be notified, while the finite state machines are used to track the current state of each process during the session execution.

To generate a GRT, first we create a dependency graph. A dependency graph is a directed graph that models the actual causal dependencies between the actions in a protocol. The dependency graph is built by syntactic traversal of the global type. The nodes of the graph model the states of the protocol, and the edges model the causal dependencies between the states. The recovery analysis is performed on the dependency graph. Our analysis is built on two insights:

All input-output dependencies before the failed node should be recovered. Since we model stateless processes, a forwarded message should be resent by its initial sender, not by intermediary one. Consider the following example where we underline the failed point:

$$\mathtt{A} \to \mathtt{B};\ \mathtt{B} \to \underline{\mathtt{C}}; \tag{7.1}$$

If C fails to receive the message from B, then A should also resend the message to B. This is because B's mail box is emptied and B's output data to C may depend on the message from A.

However if the programmer wishes not to recover all input-output dependent participants (e.g. the case there is no data dependency between the receiving and sending actions at the same participant), one can annotate the global type, for example, as follows:

$$\mathtt{A} \xrightarrow{\star} \mathtt{B};\ \mathtt{B} \to \underline{\mathtt{C}}; \tag{7.2}$$

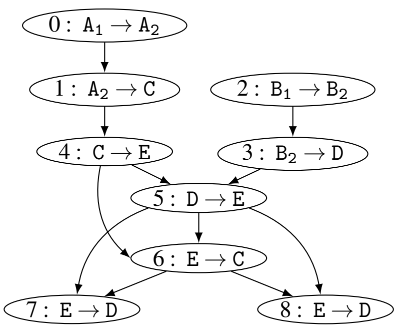
which means A does not have to be recovered (see Remark 7.2.2 in Section 7.2).

All messages in the queue of the failed process should be resent. When a process fails, its queue is emptied. At compile time (when the graph processing is done), the state of the queue is unknown. Consider the following two examples:

$$\mathtt{A} \to \underline{\mathtt{B}};\ \mathtt{C} \to \mathtt{B}; \quad \text{and} \quad \underline{\mathtt{A}} \to \mathtt{B};\ \mathtt{C} \to \mathtt{B}; \tag{7.3}$$

If B fails when receiving the message from A, the message from C might be already in B's queue. That is why our analysis will list both A and C as potential participants which should recover. Hence we notify A and C by sending the *request for recovery*. Since A already sent the message to B, A and B should be recovered. However C has two choices: if C has still not sent the message to B, it can ignore the *request for recovery* message; otherwise C should be recovered since its message in B's queue was lost. Similarly if A fails, then both B and C should be notified.

Now consider the following protocol which has additional intermediate commu-


$0: \mathtt{A}_1 \to \mathtt{A}_2$
$1: \mathtt{A}_2 \to \mathtt{C}$
$2: \mathtt{B}_1 \to \mathtt{B}_2$
$4: \mathtt{C} \to \mathtt{E}$
$3: \mathtt{B}_2 \to \mathtt{D}$
$5: \mathtt{D} \to \mathtt{E}$
$6: \mathtt{E} \to \mathtt{C}$
$7: \mathtt{E} \to \mathtt{D}$
$8: \mathtt{E} \to \mathtt{D}$

node	role	reset points
0	$\mathtt{A}_1$	$\{\mathtt{A}_1 : 0\}$
0	$\mathtt{A}_2$	$\{\mathtt{A}_2 : 0, \mathtt{A}_1 : 0\}$
1	$\mathtt{A}_2$	$\{\mathtt{A}_2 : 1, \mathtt{A}_1 : 0\}$
1	C	$\{\mathtt{C} : 1, \mathtt{A}_2 : 1, \mathtt{A}_1 : 1\}$
3	$\mathtt{B}_2$	$\{\mathtt{B}_2 : 3, \mathtt{B}_1 : 2\}$
3	D	$\{\mathtt{D} : 3, \mathtt{B}_2 : 2, \mathtt{B}_1 : 2\}$
4	C	$\{\mathtt{C} : 1, \mathtt{A}_2 : 1, \mathtt{A}_2 : 1\}$
5	D	$\{\mathtt{D} : 3, \mathtt{B}_2 : 2, \mathtt{B}_1 : 2\}$
4	E	$\{\mathtt{C} : 1, \mathtt{A}_2 : 1, \mathtt{A}_1 : 1, \mathtt{D} : 3, \mathtt{B}_2 : 2, \mathtt{B}_1 : 2\}$
5	E	$\{\mathtt{C} : 1, \mathtt{A}_2 : 1, \mathtt{A}_1 : 1, \mathtt{D} : 3, \mathtt{B}_2 : 2, \mathtt{B}_1 : 2\}$
6	E	$\{\mathtt{C} : 1, \mathtt{A}_2 : 1, \mathtt{A}_1 : 1, \mathtt{D} : 3, \mathtt{B}_2 : 2, \mathtt{B}_1 : 2\}$
6	C	$\{\mathtt{C} : 1, \mathtt{A}_2 : 1, \mathtt{A}_1 : 1, \mathtt{D} : 3, \mathtt{B}_2 : 2, \mathtt{B}_1 : 2\}$
7	E	$\{\mathtt{C} : 1, \mathtt{A}_2 : 1, \mathtt{A}_1 : 1, \mathtt{D} : 3, \mathtt{B}_2 : 2, \mathtt{B}_1 : 2\}$
7	D	$\{\mathtt{C} : 1, \mathtt{A}_2 : 1, \mathtt{A}_1 : 1, \mathtt{D} : 3, \mathtt{B}_2 : 2, \mathtt{B}_1 : 2\}$
8	E	$\{\mathtt{C} : 1, \mathtt{A}_2 : 1, \mathtt{A}_1 : 1, \mathtt{D} : 3, \mathtt{B}_2 : 2, \mathtt{B}_1 : 2\}$
8	D	$\{\mathtt{C} : 1, \mathtt{A}_2 : 1, \mathtt{A}_1 : 1, \mathtt{D} : 3, \mathtt{B}_2 : 2, \mathtt{B}_1 : 2\}$

Figure 7.3.: Dependency graph (a) and Global Recovery Table (b) for Trading Negotiation

nications to (7.3):

$$\mathtt{A} \to \underline{\mathtt{B}}; \mathtt{B} \to \mathtt{D}; \mathtt{D} \to \mathtt{C}; \mathtt{C} \to \mathtt{B}; \tag{7.4}$$

Here there is an input-output chains from the failed node at B (underlined) to C. Since the output on C is guarded by failed B, C does not have to recover since the output on C depends on the output from B, which is not sent yet.

7.1.2. Global Recovery Table and Algorithm

We give an overview of our recovery approach, following our running example. Fig. 7.3 (a) shows the dependency graph for our example and Fig. 7.3 (b) shows the global recovery table, generated by our algorithm. The algorithm explores all paths of the session graph connected to the failed node. Since the syntax is finite, the length of such paths is limited so that the algorithm terminates. For each path, the algorithm works recursively on the edges, and maintains a dictionary that records the recovery points for each participant. The GRT records the failed node (corresponding to the node of the global graph), which role has failed, and a reset (recovery) point of all roles, which depend on the failed role.

7.1.3. Runtime Supervision

In our framework, a supervision is setup at runtime when a session is started.

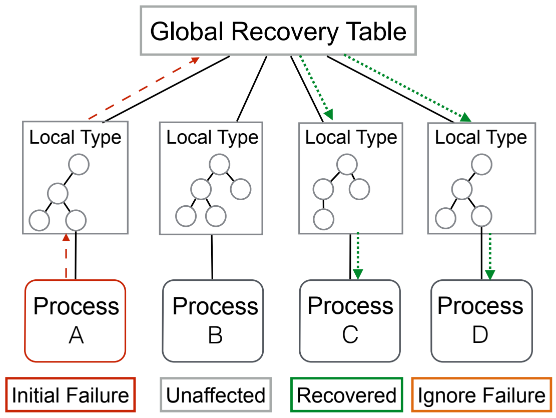


Figure 7.4.: Recovering messaging between the supervisors, and processes

Two types of entities are created: local process supervisors and processes. The local process supervisors (also called monitors) track the state of the process they supervise. Each local supervisor has a finite state machine created from the local type (during the global graph processing stage). Whenever the process performs a communication action, the monitor inspects the current state.

In Fig. 7.4, we list four possible actions that the local supervisors (LS) can take after a failure.

Once a process fails, its LS is notified. The LS performs a lookup in the Global Recovery Table (GRT) and notifies the LSs of the affected processes by sending them a *request for recovery*, containing the failed state. When the local supervisors receive *request for recovery* they query the GRT to retrieve their new state. If this state has not been reached yet, they "ignore" the *request for recovery*; otherwise they "restart" the process. The other processes remain *unaffected* (the second left-most Fig. 7.4).

As an example, consider the case in (7.3). The initial failure message for `B`'s failure is sent to `B`'s LS; Then `B`'s LS sends the recovery messages to `A`'s LS and `C`'s LS. Then `A` is restarted (the second right-most Fig. 7.4); and if `C` has not sent the message to `B` yet, then role `B` remains unaffected, otherwise it is recovered.

7.2. Semantics and Properties of Multiparty Induced Recovery

This section first presents the syntax of multiparty session types. After giving the basic ideas of our recovery algorithm, we formalise it by defining the causal relation of global types and labelled transitions of local types.

7.2.1. Global and Local Types

Syntax. We follow that exact syntax of global and local types, presented in Chapter 2. For convenience, the syntax is recalled below:

$$\begin{array}{lcl} G & ::= & \mathtt{A}{\to}\mathtt{B}\colon\{a_i.G_i\}_{i\in I} \mid G|G \mid \mid \mu\mathtt{t}.G \mid \mathtt{t} \mid \mathtt{end} \\ T & ::= & \mathtt{B}!\{a_i.T_i\}_{i\in I} \mid \mathtt{B}?\{a_i.T_i\}_{i\in I} \mid \mu\mathtt{t}.T \mid \mathtt{t} \mid \mathtt{end} \mid \end{array}$$

We remind the main constructs and notations. A *global type*, written $G, G', ..,$ describes the whole conversation scenario of a multiparty session as a type signature, and a *local type*, written by $T, T', ..,$ abstracts session communication structures from each end-point's view. $\mathbb{A}$ $(a, b, c, ...)$ denotes a finite alphabet and $\mathscr{P}$ is a set of *participants* fixed throughout the chapter. $A, B, \dots \in \mathscr{P}$ denote participants and $a_i \in \mathbb{A}$ corresponds to the usual message labels in session type theory. Note that we omit the carried types from the syntax, presented in this chapter, as we are not directly concerned with typing processes. The basic interaction type (also referred to as global branching) $\mathtt{A}{\to}\mathtt{B}\colon\{a_i.G_i\}_{i\in I}$ states that participant A can send a message with one of the a_i labels to participant B and that interaction described in G_i follows. We require $A \neq B$ to prevent self-sent messages and $a_j \neq a_k$ for all $j \neq k \in I$. Recursive types $\mu\mathtt{t}.G$ are for recursive protocols, assuming that type variables $(\mathtt{t}, \mathtt{t}', \dots)$ are guarded in the standard way, i.e. they only occur under branchings. Type $G_1 \mid G_2$ is a parallel composition. Type $\mathtt{end}$ represents session termination (often omitted). The function $\mathsf{id}(G)$ gives the participants of G.

Concerning local types, the *branching type* $\mathtt{p}?\{a_i.T_i\}_{i\in I}$ specifies the reception of a message from $\mathtt{p}$ with a label among the a_i. The *selection type* $\mathtt{p}!\{a_i.T_i\}_{i\in I}$ is its dual. The remaining type constructors are as for global types. When branching is a singleton, we write $\mathtt{A} \to \mathtt{B} : a; G'$ for global, and $B!a.T$ or $B?a.T$ for local.

For completeness, we also give the definition of projection, which formalises the formal correspondence between local and global types. The algorithms follows the one in [BCD+08, HYC08]. We define the projection with the merging operator from [DY11] to allow branching of the global type to actually contain different interaction patterns.

Definition 7.2.1 (Projection) The *projection of G onto* $\mathtt{A}$ (written $G \upharpoonright \mathtt{A}$) is defined as:

$$\mathtt{A} \to \mathtt{B} : \{a_i.G_i\}_{i \in I} \upharpoonright C = \begin{cases} \mathtt{A}!\{a_i.G_j \upharpoonright \mathtt{C}\}_{i \in I} & \mathtt{C} = \mathtt{A} \\ \mathtt{B}?\{a_i.G_i \upharpoonright \mathtt{C}\}_{i \in I} & \mathtt{C} = \mathtt{B} \\ \sqcup_{i \in I} G_i \upharpoonright \mathtt{C} & \text{otherwise} \end{cases}$$

$$(\mu \mathtt{t}.G) \upharpoonright \mathtt{p} = \begin{cases} \mu \mathtt{t}.G \upharpoonright \mathtt{p} & G \upharpoonright \mathtt{p} \neq \mathtt{t} \\ \mathtt{end} & \text{otherwise} \end{cases}$$

$$(G_1 \mid G_2) \upharpoonright \mathtt{C} \; = G_1 \upharpoonright \mathtt{C} \mid G_2 \upharpoonright \mathtt{C} \quad \text{if } \mathtt{C} \notin G_1 \text{ or } \mathtt{C} \notin G_2; \textit{ and } \mathsf{fv}(G_i) = \emptyset$$

$$\mathtt{t} \upharpoonright \mathtt{C} \; = \; \mathtt{t} \quad \mathtt{end} \upharpoonright \mathtt{C} = \; \mathtt{end}$$

where $\mathsf{fv}(G)$ denotes a set of free type variables in G. For projection of branchings, we appeal to a merge operator along the lines of [DY11], written $T \sqcup T'$, ensuring that if the locally observable behaviour of the local type is dependent of the chosen branch then it is identifiable via a unique choice/branching label.

7.2.2. Errors Generated by an Incorrect Recovery

We explain the errors that can occur by a faulty recovery process and explain the essence of our algorithm. Fig. 7.5 shows some global types and three possible cases that a recovery process introduces.

In the types, the point of recovery is marked by red and the point of error is marked by orange. We look at several popular approaches to recovery [Nys09a]: (1) recovery by resending the undelivered messages, (2) restarting the failed processes from the beginning and (3) restarting all processes. We demonstrate errors caused by unsound failure handling.

Recovery by resending a message (a deadlock). The process A fails before receiving the message from B. An intuitive recovery strategy might be to resend the unsuccessfully delivered message from B to A, not taking into account the existence of other parties (C and D). When A recovers, all contents of A's queue are deleted. However, since messaging is asynchronous, it is possible that C has already sent the message to A. This message will be lost when the queue is deleted. The process

$$\begin{array}{ll} G_1 = & \mathtt{B} \to \mathtt{C};\ \mathtt{A} \to \mathtt{C}; \\ & \mathtt{B} \to \mathtt{A};\ \mathtt{C} \to \mathtt{A}; \\ & \mathtt{A} \to \mathtt{D};\ \mathtt{D} \to \mathtt{C}; \\ \mathtt{A}: & \mathtt{C!.B?.C?.D!} \\ \mathtt{B}: & \mathtt{C!.A!} \\ \mathtt{C}: & \mathtt{B?.A?.A!.D?} \\ \mathtt{D}: & \mathtt{A?.C!} \end{array}$$

$$\begin{array}{ll} G_2 = & \mathtt{B} \to \mathtt{C};\ \mathtt{A} \to \mathtt{C}; \\ & \mathtt{B} \to \mathtt{A};\ \mathtt{A} \to \mathtt{D}; \\ & \mathtt{D} \to \mathtt{C}; \\ \mathtt{A}: & \mathtt{C!.B?.C?.D!} \\ \mathtt{B}: & \mathtt{C!.A!} \\ \mathtt{C}: & \mathtt{B?.A?.A!.D?} \\ \mathtt{D}: & \mathtt{A?.C!} \end{array}$$

$$\begin{array}{ll} G_3 = & \mathtt{E} \to \mathtt{F}\{l_1 : \mathtt{A} \to \mathtt{B};\ \mathtt{A} \to \mathtt{C};\ \mathtt{C} \to \mathtt{B}; \\ & \qquad \mathtt{B} \to \mathtt{A};\ \mathtt{A} \to \mathtt{B};, \\ & \qquad l_2 : \mathtt{A} \to \mathtt{C};\ \mathtt{C} \to \mathtt{B};\ \mathtt{B} \to \mathtt{A}; \\ & \mathtt{A} \to \mathtt{B};\ \mathtt{B} \to \mathtt{C};\} \\ \mathtt{A}: & \mathtt{B!.C!.B?.B?} \\ \mathtt{B}: & \{l_1 : \mathtt{A?.C?.A!.A!},\ l_2 : \mathtt{C?.A!.A?.A!}\} \\ \mathtt{C}: & \mathtt{A?.B!} \end{array}$$

Figure 7.5.: Recovery errors: (1) deadlock (2) orphan message error and (3) reception error (We omit the labels from local and global types except branching.)

C is at a state of receiving a message from D, after receiving the resend message from B. But A will be stuck waiting to receive the message from C. The processes end up in a deadlock.

Recovery by restarting the processes (orphan message error). Here we assume the same failure with the above case, but a different recovery strategy. Instead of resending the failed message we recover both affected processes by restarting them from the beginning. No messages are lost and no deadlock occurs in this case. However, both A and B will repeat their interactions ($\mathtt{B} \to \mathtt{C}$ and $\mathtt{A} \to \mathtt{C}$). Since C has already received these messages, the orphan messages will stay in the queue of C and will not be consumed.

Recovery might not guarantee success (reception errors). As a result of non-determinism in the global type, recovering one choice might be correct, but another choice might get stuck. In (3), there is a race condition at B, which can first receive from A or C. An initial execution might proceed with B receiving the message from A first. After recovery, B might receive first the message from C, and therefore taking a branch that is not implemented by A.

7.2.3. A Dependency Analysis on Global Types

As shown by examples in Section 7.1.3 and Section 7.2.2, the recovery algorithm needs a dependency analysis of global types. We define the two key relations ($\prec\!\!\prec_{\mathtt{IO}}$ and $\lhd$) used in the algorithm in this subsection.

Session graphs. Global types can be seen isomorphically as *session graphs*, that we define in the following way. First, we annotate in G each syntactic occurrence of subterms of the form $\mathtt{A}{\to}\mathtt{B}:\{a_i.G_i\}_{i \in I}$ with a node name (denoted by $\mathtt{n}_1, \mathtt{n}_2, \ldots$) Then, we inductively define a function `nodes` that gives a set of nodes (or the special node `end`) for each of the syntactic subterm of G as follows:

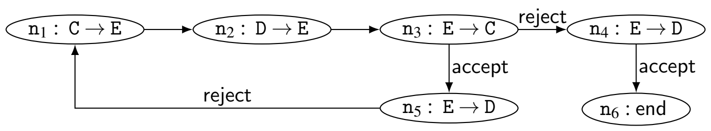
$$\begin{array}{ll} \mathtt{nodes}(\mathtt{end}) = \mathtt{end} & \mathtt{nodes}(G_1 \mid G_2) = \mathtt{nodes}(G_1) \cup \mathtt{nodes}(G_2) \\ \mathtt{nodes}(\mu\mathtt{t}.G') = \mathtt{nodes}(G') & \mathtt{nodes}(\mathtt{n} : \mathtt{A} \to \mathtt{B} : \{a_i.G_i\}_{i \in I}) = \mathtt{n} \\ \mathtt{nodes}(\mathtt{t}) = \mathtt{nodes}(\mu\mathtt{t}.G') & \textit{(if the binder of } \mathtt{t} \textit{ is } \mu\mathtt{t}.G' \in G\textit{)} \end{array}$$

Prefixes $\mathtt{n}$ and $\mathtt{n}'$ are distinct if they correspond to different syntax occurrences in G. We define G as a session graph in the following way: for each subterm of G of the form $\mathtt{n}' : \mathtt{A} \to \mathtt{B} : \{a_i.G_i\}_{i \in I}$, we have edges from $\mathtt{n}'$ to each of the $\mathtt{nodes}(G_i)$ for $i \in I$. We also define the functions $\mathtt{pfx}(\mathtt{n}_i)$, $\mathsf{id}(\mathtt{n}_i)$ and $\mathtt{ch}(\mathtt{n}_i)$ that respectively give the prefix ($\mathtt{A} \to \mathtt{B} : k$), participants ($\mathtt{A}, \mathtt{B}$) and label ($k$) that correspond to $\mathtt{n}_i$.

Example 7.2.2 (Session graph) To illustrate session graphs on recursive global types, we augment the main body of our running example with a recursion in the first case of the choice, as shown below:

$$\mu\mathtt{t}.\mathtt{C} \to \mathtt{E}; \mathtt{D} \to \mathtt{E}; \mathtt{E} \to \mathtt{C} : \begin{Bmatrix} \mathsf{accept}.\mathtt{E} \to \mathtt{D} : \mathsf{reject}.\mathtt{t}, \\ \mathsf{reject}.\mathtt{E} \to \mathtt{D} : \mathsf{accept}.\mathsf{end} \end{Bmatrix}$$

Then its graph representation with the initial node $\mathtt{n}_1$ is given as:



The recursion call yields a cycle in the graph, while branching gives the edges a_2 and a_3.

The edges of a given session graph G define a successor relation between nodes, written $\mathtt{n} \prec \mathtt{n}'$ (omitting G). Paths in this session graph are referred to by the sequence of nodes they pass through. The empty path is ε. The transitive and reflexive closure of $\prec$ is $\prec\!\!\prec$.

Causality and causality chains. We define causality relations in a given G by the relations $\lhd$ and $\prec_{\mathtt{IO}}$. The dependency relation $\lhd$ represents the order between two nodes which has a common participant; and the IO-relation $\prec_{\mathtt{IO}}$ asserts the order between a reception by a principal and the next message it sends. Formally,

$$\begin{array}{ll} \mathtt{n}_1 \lhd \mathtt{n}_2 & \text{if } \mathtt{n}_1 \prec\!\!\prec \mathtt{n}_2 \text{ and } \mathsf{id}(\mathtt{n}_1) \cap \mathsf{id}(\mathtt{n}_2) \neq \emptyset \\ \mathtt{n}_1 \prec_{\mathtt{IO}} \mathtt{n}_2 & \text{if } \mathtt{n}_1 \prec\!\!\prec \mathtt{n}_2 \text{ and } \mathtt{pfx}(\mathtt{n}_1) = \mathtt{p}_1 \to \mathtt{p} : a_1 \text{ and } \mathtt{pfx}(\mathtt{n}_2) = \mathtt{p} \to \mathtt{p}_2 : a_2 \end{array}$$

An *input-output dependency (IO-dependency)* from $\mathtt{n}_1$ to $\mathtt{n}_n$ is a chain $\mathtt{n}_1 \prec_{\mathtt{IO}} \cdots \prec_{\mathtt{IO}} \mathtt{n}_n$ $(n \geq 1)$ and we write $\prec\!\!\prec_{\mathtt{IO}}$ for a reflexive and transitive closure of $\prec_{\mathtt{IO}}$.

7.2.4. Algorithm and Semantics of Recovery

LTS for a local type. To formulate the semantics of recovery, we start from the labelled transition relation between local (endpoint) types, as given in Chapter 2.

We extend the observable actions $(\ell, \ell', \ldots)$ to denote a *recovery message* †. The full grammar for actions is given below.

$$\ell \ ::= \ \mathtt{AB}!a \mid \mathtt{AB}?a \mid \boxed{\dagger}$$

Next we add a recovery rule to the definition of LTS over local types. The relation $T \xrightarrow{\ell} T'$, for the local type of participant $\mathtt{A}$, is defined as, and we $\boxed{\text{highlight}}$ the new rule:

$$\mathtt{B}!\{a_i.T_i\}_{i\in I} \xrightarrow{\mathtt{AB!a\langle S\rangle}} T_j \quad (j \in I) \qquad \lfloor \text{LSEL} \rfloor$$

$$\mathtt{B}?\{a_i.T_i\}_{i\in I} \xrightarrow{\mathtt{AB?a\langle S\rangle}} T_j \quad (j \in I) \qquad \lfloor \text{LBRA} \rfloor$$

$$T[\mu\mathtt{t}.T/\mathtt{t}] \xrightarrow{\ell} T' \ \ \textit{imply} \ \ \mu\mathtt{t}.T \xrightarrow{\ell} T' \qquad \lfloor \text{LREC} \rfloor$$

$$\boxed{T \xrightarrow{\dagger} T' \ \ \textit{imply} \ \ \varphi(G, \mathtt{A}, T)[\mathtt{A}] = T' \qquad \lfloor \text{REC} \rfloor}$$

The first three rules are standard. The rule $[\text{REC}]$ represents the case participant $\mathtt{A}$ fails at the point of T and recovers as T'. It is defined by the function $\varphi(G, \mathtt{B}, T)[\mathtt{A}] = T'$ which means, given global type G, the participant $\mathtt{A}$ recovers to T' if it fails at T.

The main function $\varphi(G, \mathtt{B}, T_{\mathtt{B}})[\mathtt{A}]$ returns the minimum occurrence of affected nodes of participant $\mathtt{A}$ when $T_{\mathtt{B}}$ of participant $\mathtt{B}$ fails. In $[\text{REC}]$, since participant $\mathtt{A}$ fails at T, we calculate $\varphi(G, \mathtt{A}, T)[\mathtt{A}]$. To calculate a set of affected nodes, we use the dependency relations defined in Section 7.2.3. We define $\varphi(G, \mathtt{A}, T)[\mathtt{A}]$ in Definition 7.2.3 later.

Affected nodes. In this paragraph, we define the algorithm to decide the set of the affected nodes $\mathscr{N}$ when node $\mathtt{n}_i$ in global type G failed. Below (1) $\mathscr{N}^{\leftarrow}$ is a set of newly affected nodes going backwards an IO-dependency from $\mathtt{n}_i$; (2) $\mathscr{N}^{\rightarrow}$ is a set of newly added affected nodes going forward an IO-dependency from $\mathtt{n}_i$ and

(3) $\mathscr{S}$ is a set of non-affected nodes. Recall also that $\mathtt{n} \lhd \mathtt{n}'$ means that $\mathtt{n}'$ depends on $\mathtt{n}$; and $\mathtt{n} \prec\!\!\!\prec_{\mathtt{IO}} \mathtt{n}'$ means that there is an IO-chain from $\mathtt{n}$ to $\mathtt{n}'$.

To avoid confusion with the given examples in the algorithm we denote participants range over $\mathtt{p}$, $\mathtt{q}$ and $\mathtt{r}$, and we assume $\mathtt{p}$ is the failed participant. We also write $\mathtt{n} = \mathtt{p} \to \mathtt{q}$ if $\mathtt{pfx}(\mathtt{n}) = \mathtt{p} \to \mathtt{q} : k$.

1. Initialisation Set $\mathscr{N} = \mathscr{N}^{\to} = \{\mathtt{n} \mid (\mathtt{n}_i \lhd \mathtt{n} \text{ and } \mathtt{n} = \mathtt{r} \to \mathtt{p}) \text{ or } \mathtt{n} = \mathtt{n}_i\}$

Set $\mathscr{S} = \{\mathtt{n}' \mid ((\mathtt{n}_i \lhd \mathtt{n} \text{ and } \mathtt{n} = \mathtt{p} \to \mathtt{r}) \text{ or } \mathtt{n} = \mathtt{n}_i \neq \mathtt{n}') \text{ and } \mathtt{n} \prec\!\!\!\prec_{\mathtt{IO}} \mathtt{n}'\}$

2. Main Body Set $\mathscr{N}^{\leftarrow} =$

$\{\mathtt{n} \mid \mathtt{n} \prec\!\!\!\prec_{\mathtt{IO}} \mathtt{n}' \text{ and } (\mathtt{n}' \lhd \mathtt{n}'' \text{ and } \mathtt{n}' \in \mathscr{S} \text{ and } \mathtt{n}'' \in \mathscr{N}^{\to} \setminus \mathscr{S}) \text{ or } \mathtt{n}' \in \mathscr{N}^{\to} \setminus \mathscr{S}\}$

Set $\mathscr{N}^{\to} = \{\mathtt{n} \mid \mathtt{n}' \lhd \mathtt{n} \text{ and } (\mathtt{n}' \neq \mathtt{n}_i \text{ or } \mathsf{id}(\mathtt{n}) \not\ni \mathtt{p}) \text{ with } \mathtt{n}' \in \mathscr{N}^{\leftarrow}\}$.

3. Final Condition Stop if $\mathscr{N}^{\leftarrow} = \mathscr{N}^{\to} = \varnothing$

otherwise set $\mathscr{N}^{\to} = \mathscr{N}^{\to} \setminus \mathscr{N}$ and $\mathscr{N} = \mathscr{N} \cup \mathscr{N}^{\leftarrow}$ and repeat from Step 2.

Below we explain each step with examples.

Step 1: Initialisation The affected nodes $\mathscr{N}$ should include the failed node $\mathtt{n}_i$ and all nodes of the form $\mathtt{r} \to \mathtt{p}$ succeeding the failed node. For example, suppose global type $\underline{\mathtt{A}} \to \underline{\mathtt{B}}; \mathtt{C} \to \mathtt{A}$ and $\mathtt{A}$ or $\mathtt{B}$ failed at the underlined point. Then the participant $\mathtt{C}$ should be notified by the reason explained in equation (7.3).

The set of non-affected nodes $\mathscr{S}$ is set at the initialisation. By the reason explained in the equation (7.4), all nodes in the IO-dependencies starting with a message sent from the failed participant should not be affected (nodes of the form $\mathtt{A} \to \mathtt{r}$). The first part is the case where the failed node $\mathtt{n}_i$ is of the form $\mathtt{A} \to \mathtt{r}$. For example, in the case where $\mathtt{A}$ fails in $\underline{\mathtt{A}} \to \mathtt{B}; \mathtt{B} \to \mathtt{C}$, the node $\mathtt{B} \to \mathtt{C}$ is added to $\mathscr{S}$. Another case is that the failed node is of the form $\mathtt{r} \to \mathtt{A}$. For example, if $\mathtt{A} \to \underline{\mathtt{B}}; \mathtt{B} \to \mathtt{A}; \mathtt{A} \to \mathtt{C}$, then both $\mathtt{B} \to \mathtt{A}$ and $\mathtt{A} \to \mathtt{C}$ are added in $\mathscr{S}$.

Step 2: Main Body This step is repeated until the final condition is met. The set $\mathscr{N}^{\leftarrow}$ is constructed from IO-dependencies preceding any of the affected nodes ($\mathscr{N}^{\to} \setminus \mathscr{S}$) by the reason explained in (7.1). The recovered node $\mathtt{n}'$ is a backward node from the failed node $\mathtt{n}$ in an IO-dependency. In addition, if the first node $\mathtt{n}'$ belongs to $\mathscr{S}$, it should be excluded, but if it depends node $\mathtt{n}''$ which does not belong to $\mathscr{S}$ but the backward nodes from $\mathtt{n}'$ on an IO-dependency should be added in $\mathscr{N}^{\leftarrow}$.

As an example, consider the case $\mathtt{B} \to \underline{\mathtt{C}}; \mathtt{C} \to \mathtt{D}; \mathtt{E} \to \mathtt{C}; \mathtt{E} \to \mathtt{D}$. Participant $\mathtt{E}$ should be recovered because of $\mathtt{E} \to \mathtt{C}$ (since the queue of $\mathtt{C}$ is erased). Therefore participant $\mathtt{D}$ should be recovered. However, if we recover $\mathtt{D}$ from state $\mathtt{E} \to \mathtt{D}$, there will be an orphan message because the states of $\mathtt{D}$ and $\mathtt{C}$ are misaligned. Hence $\mathtt{D}$ should be recovered at the point of $\mathtt{C} \to \mathtt{D}$.

The set $\mathscr{N}^{\rightarrow}$ is constructed starting from $\mathscr{N}^{\leftarrow}$ forward. The reason is that every participant that appears after the recovery point which are not in the IO-dependencies should be informed. For example, suppose $\underline{\mathtt{A}} \to \mathtt{B}; \mathtt{C} \to \mathtt{A}; \mathtt{A} \to \mathtt{D};$. Then input from C and output to D might be lost, hence C and D should be added to $\mathscr{N}^{\rightarrow}$.

We exclude the case $\mathtt{n}' = \mathtt{n}_i$ and the nodes of forms $\mathtt{A} \to \mathtt{r}$ and $\mathtt{r} \to \mathtt{A}$ (by the condition $\mathsf{id}(\mathtt{n}) \not\ni \mathtt{p}$). This is because they have been already added in $\mathscr{S}$ in Step 1 or in $\mathscr{S}$. For example, suppose $\mathtt{A} \to \underline{\mathtt{C}}; \mathtt{E} \to \mathtt{C}; \mathtt{D} \to \mathtt{C}$ and $\mathtt{A} \to \underline{\mathtt{C}}; \mathtt{E} \to \mathtt{C}; \mathtt{C} \to \mathtt{D}$. Then $\mathtt{D} \to \mathtt{C}$ and $\mathtt{C} \to \mathtt{D}$ is added in Step 1.

Step 3: We stop if there is no additional affected node. Otherwise we define a new set of forward additional nodes ($\mathscr{N}^{\rightarrow}$) as the nodes excluding the current affected nodes ($\mathscr{N}$). Then we update the set of affected nodes ($\mathscr{N}$) with the set of backward nodes ($\mathscr{N}^{\leftarrow}$) that were calculated in Step 2.

Below we write $G/\mathtt{n}_\mathtt{q}$ to denote a subterm (subgraph) of G whose occurrence is $\mathtt{n}_\mathtt{q}$. For example, if we take the session graph in Example 7.2.2, $G/\mathtt{n}_5 = \mathtt{C} \to \mathtt{A} : a_6.\mathtt{end}$ and $G/\mathtt{n}_2 = \mathtt{C} \to \mathtt{A} : a_4; \mathtt{A} \to \mathtt{C} \colon a_5; G$, writing the outputs by the syntax of global types.

Write $T \supseteq T'$ to denote T is a subterm of T' (e.g. $\mathtt{B}!b.\mathtt{end} \subseteq \mathtt{A}?a.\mathtt{B}!b.\mathtt{end}$). We define $\max(\{T_i\}_{i\in I}) = T_j$ if for all $i \in I$, $T_j \supseteq T_i$.

Definition 7.2.3 (Recovery point) *Assume that $\mathscr{N}$ is the set of affected nodes when $\mathtt{n}_i$ in global type G failed. Assume the participant $\mathtt{p} \in \mathsf{id}(\mathtt{n}_i)$ failed and $G/\mathtt{n}_i 1 \downarrow_= T_\mathtt{p}$. We then define, for each $\mathtt{q} \in \mathscr{P}$, (1) $\varphi(G, \mathtt{p}, T_\mathtt{p})[\mathtt{q}] = \max(\{G/\mathtt{n}_\mathtt{q} \mathtt{q} \downarrow \mid \mathtt{n} \in \mathscr{N}, \mathtt{q} \in \mathsf{id}(\mathtt{n})\})$; or (2) $\varphi(G, \mathtt{p}, T_\mathtt{p})[\mathtt{q}] = \emptyset$ if $\mathtt{q} \notin \mathsf{id}(\mathtt{n})$ for all $\mathtt{n} \in \mathscr{N}$.*

LTS over a configuration. We define the LTS for a configuration which consists of a collection of local types and FIFO queues. The item (1) below defines the standard asynchronous communication rules from communicating finite state machines [BZ83]. The participant A enqueues a value to FIFO queue $w_{\mathtt{AB}}$ of channel Aq, and participant B dequeues a value from $w_{\mathtt{AB}}$. In addition, we define the two cases when participant A fails at $T_\mathtt{A}$ (item 2 below). The item (a) is the case when participant B needs to recover as local type $T'_\mathtt{B}$. In this case, we clean up its input queues. The item (b) is the case participant B does not need to recover. In this case, we do not have to change the configuration for B. Note by the case (2-a), failed participant A always clean up its input queues.

Definition 7.2.4 (A configuration and its LTS) A configuration $s = (\vec{T};\vec{w})$ of a system of local types $\{T_{\mathtt{A}}\}_{\mathtt{A}\in\mathscr{P}}$ is a pair with $\vec{T} = (T_{\mathtt{A}})_{\mathtt{A}\in\mathscr{P}}$ and $\vec{w} = (w_{\mathtt{AB}})_{\mathtt{A}\neq\mathtt{B}\in\mathscr{P}}$ with $w_{\mathtt{AB}} \in \mathbb{A}^*$. The *initial configuration* of G is $s = (\vec{T};\vec{w})$ with $w_{\mathtt{AB}} = \varepsilon$ and $T_{\mathtt{A}} = G\downarrow_{\mathtt{A}}$.

A *final configuration* is $s = (\vec{T};\vec{\varepsilon})$ with $T_i = \mathtt{end}$. We then define the transition system for configurations starting from the initial configuration of G. For a configuration $s = (\vec{T};\vec{w})$, the visible transitions of $s \xrightarrow{\ell} s' = (\vec{T}';\vec{w}')$ are defined as:

1. $T_{\mathtt{A}} \xrightarrow{\mathtt{AB}!a} T'_{\mathtt{A}}$ and $w'_{\mathtt{AB}} = w_{\mathtt{AB}} \cdot a$ and $T'_{\mathtt{A}'} = T_{\mathtt{A}'}$ for all $\mathtt{A}' \neq \mathtt{A}$; or
   $T_{\mathtt{B}} \xrightarrow{\mathtt{AB}?a} T'_{\mathtt{B}}$ and $w_{\mathtt{AB}} = a \cdot w'_{\mathtt{AB}}$ and $T'_{\mathtt{A}'} = T_{\mathtt{A}'}$ for all $\mathtt{A}' \neq \mathtt{B}$
   with $w'_{\mathtt{A}'\mathtt{B}'} = w_{\mathtt{A}'\mathtt{B}'}$ for all $\mathtt{A}'\mathtt{B}' \neq \mathtt{AB}$; or
2. $T_{\mathtt{A}} \xrightarrow{\dagger} T'_{\mathtt{A}}$ then
   a) if $\varphi(G,T_{\mathtt{A}},\mathtt{A})[\mathtt{B}] = T''_{\mathtt{B}}$ then $T'_{\mathtt{B}} = T''_{\mathtt{B}}$ and $w'_{\mathtt{CB}} = \varepsilon$ for all $\mathtt{r} \neq \mathtt{B}$; and
   b) if $\varphi(G,T_{\mathtt{A}},\mathtt{A})[\mathtt{B}] = \emptyset$ then $T'_{\mathtt{B}} = T_{\mathtt{B}}$ and $w'_{\mathtt{CB}} = w_{\mathtt{CB}}$ and $w'_{\mathtt{BC}} = w_{\mathtt{BC}}$ for all $\mathtt{C} \neq \mathtt{B}$

We denote $s \xrightarrow{\ell_1\cdots\ell_n} s'$ (or $s \to^* s'$) for $s \xrightarrow{\ell_1} s_1 \cdots s_{n-1} \xrightarrow{\ell_n} s'$.

A configuration s is *reachable* if $s_0 \to^* s$.

Remark 7.2.1 (Asynchrony of recovery) *While the rules for recovery in Definition 7.2.4 (2-a) looks synchronous, combining enqueue and dequeue rules in (1), the definition represents asynchronous semantics. It can model the situation where the recovery message is stored in the queue and propagated to the participants.*

Example 7.2.5 (Trading Negotiation) *We recall the example from Fig. 7.3.*

(1) Participant $\mathtt{B}_2$ fails at node 3. *Then by Definition 7.2.3, $\varphi(G,\mathtt{B}_2,T_{\mathtt{B}_2})[\mathtt{B}_2] = T_{\mathtt{B}_2}$; $\varphi(G,\mathtt{B}_2,T_{\mathtt{B}_2})[\mathtt{B}_1] = T_{\mathtt{B}_1}$ where $T_{\mathtt{B}_1} = G \upharpoonright \mathtt{B}_1$ and $T_{\mathtt{B}_2} = G \upharpoonright \mathtt{B}_2$. Also for all $\mathtt{p} \notin \{\mathtt{B}_1,\mathtt{B}_2\}$, $\varphi(G,\mathtt{B}_2,T_{\mathtt{B}_2})[\mathtt{A}] = \emptyset$. By Definition 7.2.4(2), $T_{\mathtt{B}_2} \xrightarrow{\dagger} T_{\mathtt{B}_2}$. We also set the input queues of $\mathtt{B}_1$ and $\mathtt{B}_2$ to be empty. Hence $w'_{\mathtt{B}_1\mathtt{B}_2} = w'_{\mathtt{B}_2\mathtt{B}_1} = \varepsilon$. By (2-a), we set $T'_{\mathtt{B}_1} = T_{\mathtt{B}_1}$. Except $\mathtt{B}_1$ and $\mathtt{B}_2$, the queues and local types are unchanged. Note that if $\mathtt{p} \neq \mathtt{B}_1$, $w_{\mathtt{AB}_2}$ is empty before $\mathtt{B}_2$ fails (since $\varphi(G,\mathtt{B}_2,T_{\mathtt{B}_2})[\mathtt{A}] = \emptyset$). The cases when $\mathtt{A}_2$ fails at node 1 and $\mathtt{C}$ fails at node 4 can be calculated similarly.*

(2) Participant $\mathtt{E}$ fails at node 4. *By the algorithm, (1) nodes 0 and 1 are added to $\mathscr{N}$ since there is a backward IO-dependency $\mathtt{n}_0 \prec\!\!\prec_{\mathtt{IO}} \mathtt{n}_1 \prec\!\!\prec_{\mathtt{IO}} \mathtt{n}_4$; also node 5 is added to $\mathscr{N}^{\rightarrow}$ since $\mathtt{n}_4 \lhd \mathtt{n}_5$. From node 5, there is a backward IO-dependency $\mathtt{n}_2 \prec\!\!\prec_{\mathtt{IO}} \mathtt{n}_3 \prec\!\!\prec_{\mathtt{IO}} \mathtt{n}_5$, hence nodes 2 and 3 are also added to $\mathscr{N}$. Hence for all $\mathtt{p}$, $\varphi(G,\mathtt{B}_2,T_{\mathtt{B}_2})[\mathtt{A}] = G \upharpoonright \mathtt{p}$. By Definition 7.2.4 (2-a), all participants will restart from the beginning of the protocol.*

Remark 7.2.2 (Reducing a backwards IO-dependency.) *As we discussed in (7.1) the programmer might not wish to recover all participants in the backwards IO-dependencies from the failed node, for example, when there is no data dependency. In this case, we annotate a global type as* $\mathtt{A}\xrightarrow{\star}\mathtt{B} : a; G$ *where* $\star$ *represents this node does not have to recover. For example, if we annotate like* $\mathtt{A}\xrightarrow{\star}\mathtt{B} : a; \mathtt{B} \to \mathtt{C} : b$*, even* $\mathtt{C}$ *failed,* $\mathtt{A}$ *does not have to recover.*

In this case, we do not add node $\mathtt{A} \to \mathtt{B}$ *to a set of the recovery nodes* $\mathscr{N}$ *when* $\mathtt{B} \to \mathtt{C} : b$ *fails. The rest of the algorithm is unchanged.*

7.2.5. Towards Transparency of Recovery

One remaining technical question is *transparency* of the recovery procedure. Transparency means that once a configuration recovers to some state after a failure, there are always transitions which can reach another state unaffected by failures. Hence we can recover the configuration as if there were no failure. More precisely, the property of transparency is formalised in the following statement.

> *Transparency:* Suppose s_0 is the initial configuration of G. Assume $s_0 \xrightarrow{\vec{\ell}}{}^{*} s \xrightarrow{\dagger} s'$. Then there exists $s' \xrightarrow{\vec{\ell}_1} s''$ where $s_0 \xrightarrow{\vec{\ell}_2}{}^{*} s''$ and $\vec{\ell}_2$ does not contain $\dagger$.

Thus, if we restart a set of processes after a failure of some process, all processes will recover from some point of a complete subprotocol of the original G. Our intuition is grounded on two conjectures. First, a set of local types defined by φ for a given failed type $T_{\mathtt{p}}$ and G, i.e. $\{\varphi(G,\mathtt{p},T_{\mathtt{p}})[\mathtt{q}]\}_{\mathtt{q}\in\mathscr{P}}$ should form a projection of a subgraph of G. Second, if $\varphi(G,\mathtt{p},T_{\mathtt{p}})[\mathtt{q}]$ is empty, then its queue must be empty before recovery.

7.3. Erlang Programming with Multiparty Session Protocols

We implement the recovery semantics and algorithm (as explained in Section 7.2.4) in a new Erlang library. We use the practical incarnation of session types, the specification language Scribble, presented in Chapter 2, to write global protocols. The main components of the library are summarised in Table 7.2.

- **Scribble module** (`sribblec`): a module for processing global Scribble protocols. The module takes a Scribble protocol as an input and generates (1)

Component	Usage
gen_protocol	−behaviour(gen_protocol)
protocol_supervisor	−bevaiour(protocol_supervisor)
role	role:send
monapp	mon_app:configure
scribblec	scribblec:create_rec_table ()

Table 7.2.: Components of a library for protocol-induced applications

local types (following 7.2.1) and (2) global recovery tables (following the recovery algorithm from Section 7.2.4).

- **Monitoring runtime** (`monapp` application and `role` modules): these components implement the runtime semantics for protocol verification and recovery. The monitor application (`monapp`) creates a monitor process (called `role`) per Erlang process. A role checks, at runtime, that the messages sent and received by the process correspond to the local type. In the case of failure, roles restart their respective processes following the semantics given in Section 7.2.4.
- **An interface for local processes** (`gen_protocol behaviour`): this module provides the basic functionality for a process to be eligible for verification and recovery.

For developers to use the system there are two requirements (the figure numbers correspond to the respective implementation for each requirement for the Trading Negotiation protocol in Section 7):

1. Define the process interactions into a Scribble protocol (Listing 7.1)
2. Implement a `gen_protocol` process for each role in a protocol. The role and the protocol must be specified as part of the process initialisation (Fig. 7.6). The process is also required to implement message handlers for all interactions in the protocol (Fig. 7.3).

We explain the processing of Scribble protocols and how to program with `gen_protocol` in Section 7.3.1 and Section 7.3.2 respectively.

7.3.1. Scribble Protocols for Recovery and Verification

The correspondence between Scribble constructs and their multiparty session types (MPST) is given in Chapter 2. The global Scribble protocol for the Trading Negotiation example, is shown in Listing 7.1.

Listing 7.1: Scribble Protocol for Trading Negotiation

```
global protocol Trading (
  role A, role B,
  role C, role D){
par {
  quote(int) from A to C;
}
and
{
  quote(int) from B to D;
}
  quote(int) from C to E;
  quote(int) from D to E;
  choice at E {
    accept() from E to C;
    accept() from E to D;
  or {
    reject() from E to C;
    reject() from E to D;
  }
}}
```

Next we highlight the details important for runtime semantics.

Protocol Processing Once the protocol is written, the Scribble tool automatically checks the correctness of the protocol and generates local specifications. We then translate them to finite state machines (FSMs) to allow efficient state tracking (see Fig. 7.4). While our recovery analysis is solely based on global protocols, we use the correspondence between local types and finite state machines [DY12] to monitor and track the execution state for each role for monitoring.

Storing global recovery tables Our tool parses a global Scribble protocol to create a dependency graph (as explained in Section 7.2.3). Then it calculates the corresponding Global Recovery Table (GRT), following Section 7.2.4, and stores it in a database. This is done statically before an application is started. We generate a GRT per protocol. The scheme for the database table, holding the GRT records, follows the shape, presented Fig. 7.3; e.g. for every protocol state, a map of affected roles and their reset points is stored. For a persistent storage, we use Mnesia, which is optimised for fast query processing on a read-only tables where locks are not needed, as in our case. At runtime the GRT records are accessed only when a role has to retrieve its

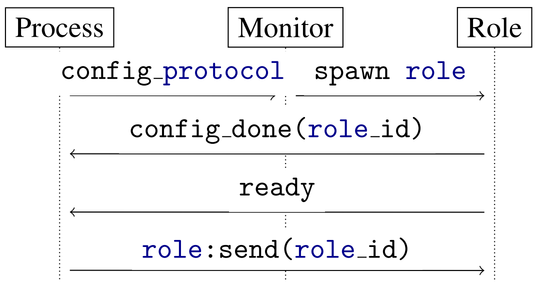


Figure 7.6.: Process configuration calls diagram

new state due to a process failure.

7.3.2. Constructs and Erlang API

Roles. A `role` intercepts all incoming/outgoing messages associated to its linked process; Roles correspond to the local supervisors shown Fig. 7.4 in Section 7.1. When a role is spawned, it is parameterised with a local protocol. One local protocol is translated to one FSM. Every time a process sends a message, the role checks it is expected w.r.t. the current state; if so the current state moves to the next state and the message is sent to the destination role.

Endpoint processes. Endpoint (`gen_protocol`) processes implement the business logic for a role in a protocol. The endpoint processes are reactive. They define message handlers and react upon received message. The order of messages is not specified since all protocols are statically defined, and the verification process (the `role` associated to the endpoint process) ensures the messages follow the order in the protocol.

For an endpoint process to be part of a protocol, it should implement a custom behaviour `gen_protocol`. Behaviours in Erlang are similar to abstract classes; they encapsulate a common pattern (behaviour) and expose a set of required methods to be implemented. We implement the `gen_protocol` behaviour as an event server. It receives messages from the process mailbox and dispatches them to the message handlers defined in the process. For example, if a message of the form `{<123>, sum, 1, 2}` is received in the mailbox of the process with `id <123>` the event server dispatches the message, and invokes the function `sum(1, 2)` for that process. To comply with the `gen_protocol` a module should define several methods: `init`, `config_done`, `ready` and `terminate`.

Listing 7.2: Process configuration code snippet

```
1  -behaviour(gen_protocol).
2
3  init({Role})→
4    Pr = {self(),
5        [{?Protocol,Role,?Roles,?Methods}]},
6    ok = monapp:config_protocol(Pr),
7    {ok, #state}.
8
9  config_done({Id}, State)→
10   {noreply, State#state{role = id}}.
11
12 ready(_,State) →
13   role:send(State#state.session, ?C, quote, 100),
14   {noreply, State}.
```

Configuration. Fig. 7.6 displays the sequence of configuration calls performed when a process is initialised and Listing 7.2 (right) shows the corresponding code snippet that should be implemented by `gen_protocol` processes. Process initialisation is performed in the function `init`. On `init` a process sends its configuration details to the monitoring application (line 6), namely its process id (`self()`), the name of a protocol (`?Protocol`) and the name of a role (`?Role`). The `?Methods` parameter is a mapping from Scribble labels to message handlers. For example, the `?Methods` parameter for role `E` is defined as

```
-define(METHODS, [{quote:quote},  {accept:accept}, {reject:reject}].
```

This mapping is used at runtime, prior to starting any protocol interaction, to check that all interactions have corresponding handlers.

When the monitoring application receives a configuration call, it spawns a process of type `role`. The `role` process replies to the endpoint process, that initiated the configuration, with its id (`role_id`) (line 9). Then the `role_id` is saved as part of the endpoint process state (line 10). When all roles are configured a message `ready` is sent to all endpoint processes and the processes can start interacting. For example, the Trading protocol in Fig. 7.1 prescribes that the roles `A` and `B` are the first to send a message. Line 13 in Listing 7.2 shows the `ready` implementation for role `A`. The other processes, e.g `C`, `D` and `E` are started in a waiting state and therefore the `ready` handler is implemented as:

```
ready(_, State) → {ok, State}
```

Communication between endpoint. Endpoint processes communicate via the following API function:

```
role:send(Id, role_name, method_name, Args)
```

The first parameter, `Id`, is the id of a `role` linked to a process, `role_name` is the name of the destination role as given in the protocol. For example, line 13 in Listin 7.6 specifies sending a message to role `C` (?C). The parameter `method_name` is a label for the message being sent. For example, if a protocol specifies `sum(int , int)` then `method_name` is `sum`. The last parameter, `Args`, stands for payload arguments.

Message handlers as callbacks. The rest of an endpoint process implementation consists of defining message handlers for protocol messages.

Listing 7.3: Message handlers for endpoint processes

```
% Handlers for C and D
quote({msg, Val}, State) →
  role:send(State#state.role, ?E, quote, Val).

accept({msg, _}, State) → {ok, State}
reject({msg, _}, State) → {ok, State}

% Handlers for E
quote({msg, Val}, State) when State.prev==undef →
  {noreply, State#state{prev=Val}};

quote({msg, Val}, State) when State#state.prev < Val →
  role:send(State#state.role, roleE, accept, empty),
  role:send(State#state.role, roleD, reject, empty),
  {noreply, State};

quote({msg, Val}, State) when State#state.amount > Val →
  role:send(State#state.role, roleE, reject, empty),
  role:send(State#state.role, roleD, accept, empty),
  {noreply, State}.
```

Fig. 7.3 displays the callbacks required for the modules implementing the endpoint processes for roles `C`, `D`, and `E`. Note that `A` and `B` do not have any interactions after sending an initial `quote` message and no other handlers except `ready` are needed. The `C` process and the `D` process implement the function `quote`, `accept` and `reject`. In `quote` both `C` and `D` simply resend the received message to the process `E`. The internal choice on `E` described in the protocol is implemented as a guard on the message handler `quote`. The guard compares the values received from `C` and `D` and sends `accept` to whoever sends the highest quote.

Starting a protocol. The last requirement is implementing a supervisor for the endpoint processes. A supervisor declaration specifies the initial parameters for the endpoint processes as the one given below:

```
-module(trading_supervisor).
-behaviour(protocol_supervisor).
init([]) → {ok, [{roleA, {roleA,start_link,[A]},
        ...{{roleE, {roleA,start_link,[E]}]}.
```

The behaviour (`protocol_supervisor`) is a custom behaviour of type supervisor and takes the one-for-one-strategy. The `init` function returns the list of processes that should be started as part of the protocol. The parameter lists has the form `module`, `initial function`, `arguments`.

Finally to start a protocol we execute trading_supervisor:start (). It spawns all endpoint processes defined in the supervisor.

7.4. Implementation of Multiparty Induced Recovery

This section summarises the implementation of monitoring and recovering.

7.4.1. Protocol Checking

Fig. 7.7 highlights the calls between components on sending a message. More specifically, the diagram shows role processes are implemented as gen_server. The behaviour gen_server is an Erlang/OTP behaviour, which is a generic implementation of an event-loop server and messages are sent asynchronously using the Erlang call gen_server:cast . First, the message is sent to the process role, where it is checked for correctness and resent to the destination role via (1) gen_server:cast (RoleId, Msg). The destination role dispatches the message to a message handler in the destination process via (2) gen_protocol:cast (id, Msg)).

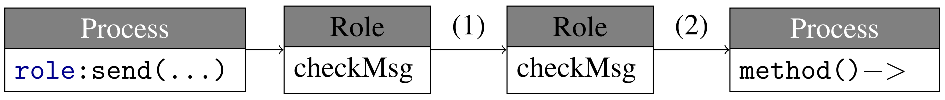


Figure 7.7.: Sending a message from one process to another.

The message is dispatched to the role process that matches the process id, given as a first argument of gen_server:cast . The role process checks whether the message has a matching transition in the local FSM. If it does, the message is sent to the destination process invoking (2) gen_protocol:cast (id, Msg). Note that the process of generation of FSMs from Scribble protocols and the design of a Scribble-based monitor, follows the ones presented in Chapter 2 and Chapter 6, respectively. In this section we focus only on technical challenges that are specific to implementing a monitor module in Erlang.

Selective receive. The behaviour gen_server is an event server with a FIFO queue and its receive semantics correspond to the asynchronous unordered receives as described by $\mathtt{A} \to \mathtt{B} \mid \mathtt{C} \to \mathtt{B}$ in a global type.

On the other hand, the runtime configuration semantics presented in Section 7.2.4 require selective receive as represented by the global type $\mathtt{A} \to \mathtt{B}; \mathtt{C} \to \mathtt{B}$. Consider the case when role B receives a message from C before receiving a message from A, the role (monitor) process will tag the message as an invalid one since it is an unexpected message w.r.t the current state. To prevent this behaviour we implement selective receive on top of the gen_server FIFO queues. We borrow the idea from the selective receive implementation as developed in core Erlang [Scra]. The idea is to store the early arrived messages in an in-memory queue (hereafter saved queue). If a message arrives too early (in case of unordered delivery), it is stored in the saved queue. Every time the server is awakened to read messages from its FIFO queue, it first checks the saved queue for any messages that are ready to be processes and only after that reads from the FIFO queue.

7.4.2. Supervision

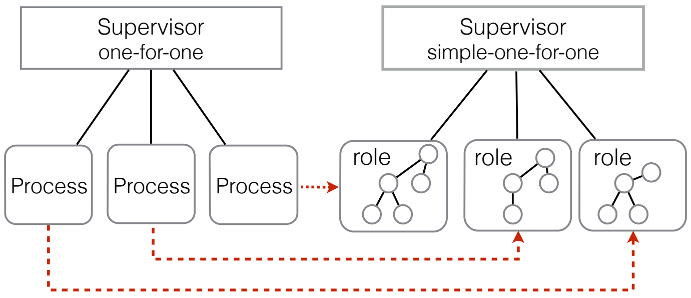


Figure 7.8.: Supervision hierarchy

Supervision. We build our recovery strategy on top of the runtime protocol verification, provided by role and gen_process. Decoupling of a protocol checker

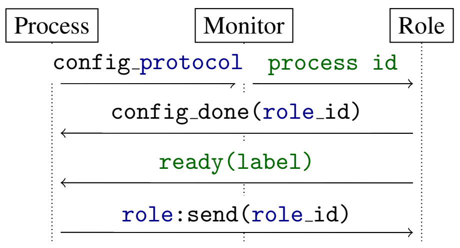


Figure 7.9.: Process restart

(role) and an endpoint process (gen_process) is essential for the recovery. This way processes do not send messages directly to the other endpoints and as a result when a process fails only its role is notified about the new process id. Therefore the failure is transparent to the other endpoint processes.

The implementation of our recovery mechanism draws on two Erlang features:

1. Erlang links: a mechanism for creating a bidirectional link between processes. It ensures that terminating processes emit exit signals to all linked processes. We use link to connect a running process to its role.
2. dynamic supervision strategy (simple−one−for−all): a strategy which is similar with one−for−all strategy, but allows all workers to be started dynamically. This strategy can be applied only if all of the workers have the same type. We use simple−one−for−one supervision to spawn and supervise roles.

A running system with three participants is shown in Fig. 7.8. The figure displays the supervision structure and the links (denoted as dotted red arrows) created by our runtime. It shows three processes, grouped by a protocol_supervisor (Supervisor one-for-one in Fig. 7.8), which is an extension of the Erlang's one−to−one supervisor. The latter is needed to prevent a cascading failure. When a role is created during process initialisation (as explained in Fig. 7.6), the role receives the process id and links to it (invoking the built-in Erlang function link(processId)). The link ensures the role will be notified if the process fails.

Failure handling. The recovery mechanism after a process fails consists of three key parts:

1. **Notification of a failure.** A role receives the system message 'EXIT' when a process fails (this is handled by the Erlang built-in mechanism of links). Then the role broadcasts the failed state to the other affected roles (message

'FAIL'). The set of affected roles is retrieved from the GRT. Only the state number is broadcasted, because it uniquely identifies the state. The projection ensures a unique correspondence between local states and states in the global dependency graph.

2. **Restart a process.** Fig. 7.9 displays the sequence of configuration calls when a process init function is invoked after a restart. The restart specific actions are marked in `green`. First, the process registers with the monitor. Then the monitor checks that a role exists. If it exists, the monitor sends the id of the new process to the existing role.
3. **Obtain reset points.** A a role queries the GRT when it receives a message 'FAIL' or 'EXIT'. As a result, the role retrieves the reset state for the new process. A state represents a node in the local FSM and has the form: (state number, state kind, scribble label, role); where state kind specifies if this is a sending or a receiving state. The role updates its FSM and invokes the process function ready passing the label name (gen_protocol:handle_cast (ready, ProcessId, label)).

7.5. Use Cases and Evaluations

Example	Source	#roles	#states	GRT	Affected roles	G
Trading Negotiation (TN)	p.1	2∗n+1	2∗n+4	0.17	in Table 1	Y
MapReduce (MR)	[IS14]	n+1	n+2	0.11	W[1] ... W[n]	Y
Ring (Rg)	[IS14]	n	2∗n	0.16	A[n], A[n−1]	Y
Calculator (Calc)	[HWT14]	n+1	4∗n	0.75	A[1]	N
Resource Allocator (Cig)	[IS14]	n	n+1	$+e$	A[1], R	Y

Table 7.3.: Use cases performance summary (GRT generation time is calculated for $n = 100$ and the last column (G) indicates if results of our approach are better than the Erlang all-for-one strategy

The aim of the evaluation in this section is to demonstrate the applicability of our recovery strategy (called hereafter *protocol-recovery*) to several typical concurrency patterns from the literature [IS14, HWT14]. The overhead of *protocol-recovery* comes from (1) the overhead of propagating the error and (2) the lookups performed on the global recovery table.

We compare *protocol-recovery* against the Erlang *all-for-one* supervision strategy. In summary, the overhead of *protocol-recovery* is very small and it outperforms when recovering fewer processes, resulting in faster protocol execution times than the *all-for-one* strategy. Although comparing with *all-for-one* may sound limited, no other static recovery strategy can be safely applied to the examples without resulting in an inconsistent (unsound) monitor state.

7.5.1. Evaluation Methodology

We present four examples with different communication patterns. We implement the programs using the Erlang API, presented in Section 7.3. For each example, we give the global type and discuss the result of *protocol-recovery* in terms of (1) number of recovered participants and (2) overhead. For the latter, we compare the execution times for completing the protocol without a failure, and with a failure followed by a subsequent recovery. The results of our evaluation are summarised in Table 7.3 and Fig. 7.10. Table 7.3 shows, in this order, the number of roles for each protocol (#roles), the number of states in the dependency graph (#states), the time (in seconds) for calculating the Global Recovery Table (GRT), the roles that will be notified (Affected roles), and the performance improvement of our runtime when comparing with *all-for-one* supervision (G). Fig. 7.10 (left) shows the results of executing the four scenarios of the Trading Negotiation example; and Fig. 7.10 (right) shows the benchmark results of other use cases.

Setup. We repeated each scenario 50 times and measured the overall execution time. Each example is parameterised on an input size `n` (as shown Fig. 7.3) and the results displayed Fig. 7.10 are for `n`=100. Programs are written using Erlang, version 17.0. All processes run on the same Erlang node with a Mnesia database running on a separate node. The configuration for the machine is Ubuntu 13.04 64bits GNU/Linux; 8 Cores: Intel(R) Core i7-4770 CPU @ 3.40GHz 16Gb of RAM.

7.5.2. Use Cases and Benchmarks

Trading Negotiation (Fig. 7.1 in Section 7). We spawn 100 processes for Alice's and Bob's group. The results of an execution without recovery of the protocol is given in Fig. 7.10 (left) (denoted as Sc0). The failing scenarios (displayed Fig. 7.10 (left) as Sc1, Sc2 and Sc3) correspond to the ones from the scenarios in Section 7. In case of recovering fewer processes, as in Sc1 and Sc3, the protocol completes faster if we apply *protocol-recovery* than if we restart all interactions by the *all-for-one* supervision. In case of Sc2, we need to restart all processes. In this case, *protocol-recovery* induces only a small overhead 0.9% in comparison to *all-for-*

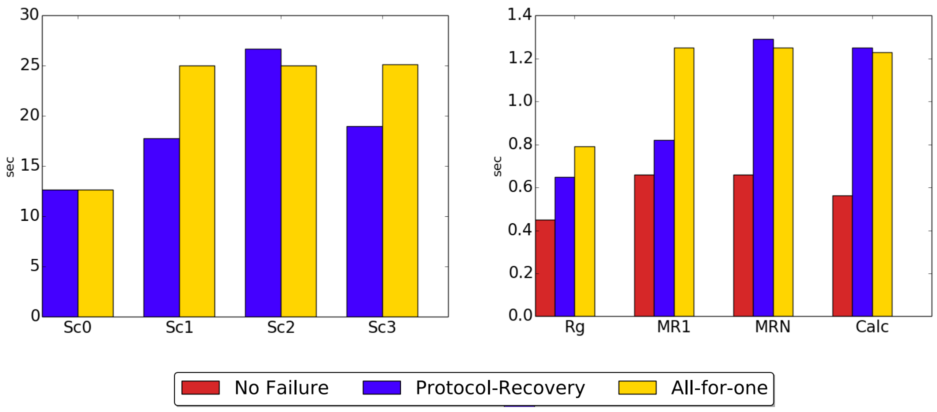
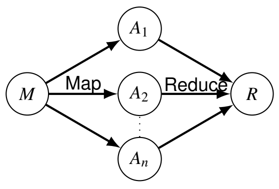


Figure 7.10.: Execution time for Trading Negotiation (left) and Ring (RG), MapReduce (MR1 and MRN) and Calculator (Calc) (right)

one supervision.

MapReduce. Listing 7.4 shows the implementation of a map-reduce protocol. Its global type is given as follows using the notation from Chapter 2:

$$\texttt{Master} \xrightarrow{\star} \texttt{Worker}[\texttt{i} : \texttt{1..n}] : \texttt{map};\ \texttt{Worker}[\texttt{i}] \to \texttt{Master} : \texttt{result};$$



The protocol describes a master (`Master`) sending `map` messages to worker process (`Worker[i]`) ($1 \leq i \leq n$). Notation `Worker[i : 1..n]` means `i` is parameterised ranging over `1` to `n` and `i` binds the rest of the free occurrences of `i` in the rest of the global type. $\star$ is the annotation for ignoring recovery as explained in the equation (7.2) in Section 7.1.3 and in Remark 7.2.2. Each worker performs a task and notify the `Master` by sending a `result` message. All threads are executed in parallel, and `Master` gathers all results.

Failure. The process `Master` fails after sending `map` to everyone. Without protocol annotations, our algorithm calculates that the protocol should be restarted from the beginning since the final result of the value might be depended on the value initially sent by `Master`. However, if we annotate the protocol as above, `Master` can recover from a state where it waits to receive a `result` from a worker. Therefore, if workers still have not sent `result`, they do not need to recompute their task.

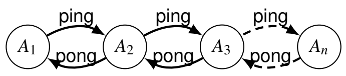
Results. The execution time with recovery depends on the number of lost messages in `Master`'s mailbox and the executions of this scenario vary depending on the the computation intensity of the task performed by the workers. As shown in Fig. 7.10 (bars MRN and MR1), when workers respond immediately after receiving a `map` message (denoted MRN in Fig. 7.10), *protocol-recovery* behaves similar to *all-for-one* strategy because almost all `result` messages are resent. However, when workers perform more computation intensive tasks (denoted MR1 in Fig. 7.10) (in our benchmark each `Worker`[`i`] sorts a list of 10000 elements), after a failure, *protocol-recovery* outperforms the *all-for-one* recovery taking only 20% of the time to complete the protocol (the time before the recovery starts).

Listing 7.4: Map Reduce Protocol Implementation

```
init({Num, Total})→
  {registered, Id} = monscr:register(self()),

  Pr = {Id,[{?PROT, role_name(Num),
    generate_others(Num, Total),  ?METHODS}]},

  ok = monscr:config_protocol(Pr),
  {ok, #state{count = 0, num=Num, total=Total}}.

ready(_,S) when S#state.num =:= 1→
  % send map to all roles
  [role:send(S#state.role,role_name(Num),map, val)
  || Num <- lists:seq(2, S.total)],
  logger:log("Failure_Point")
  {noreply, S};

ready(_,S)  when S#state.num > 1 → {noreply, S}.

map({msg, Val}, S) →
  % sort a list of sort_list(File)
  role:send(S#state.role,role_name(1), result, val),
  {noreply, S};

result({msg, Val}, S)  →
  {noreply, S#state{count = S#state.count + 1}}.

% The protocol ends
result({msg, Val}, State)
  when S#state.count =:= S#state.total →
    {noreply, S};
```

Ring. Next we consider a common pattern of chained interactions between the number of `n` processes where each process $\mathtt{A}[\mathtt{i}]$ sends a `ping` message to its immediate neighbour $\mathtt{A}[\mathtt{i}+1]$. When a process $\mathtt{A}[\mathtt{n}]$ receives a `ping` message, it starts a chain of `pong` messages. The implementation is shown in Listing 7.5 and the global type for the protocol is given below:

$$\mathtt{A}[\mathtt{i}:1..\mathtt{n}-1]\xrightarrow{\star}\mathtt{A}[\mathtt{i}+1]:\mathtt{ping};$$
$$\mathtt{A}[\mathtt{i}:\mathtt{n}..2]\rightarrow\mathtt{A}[\mathtt{i}-1]:\mathtt{pong};$$



Failure. We consider a failure at process `k` before sending a `pong` message. We insert a check point on the chain of `pong` messages.

Result. The total of `n-k+1` processes are restarted (these are processes `n, n−1, ..., k`). Fig. 7.10 (bar Rg) shows the execution time when `k = n−1` which requires restart of only two processes and thus the significant performance gain (*protocol-recovery* outperforms the *all-for-one* recovery by 52%).

Distributed Calculator. This protocol is a modification of the ring protocol. Processes cooperate to solve an equation. Each process ($\mathtt{A}[\mathtt{i}]$) calculates an expression (by sending `expr` to a calculator `C`) and to resend the continuation `cont` (the rest of the equation) to its neighbour. Then the process $\mathtt{A}[\mathtt{i}]$ waits for `result` from `C` and for the `result` of their neighbour $\mathtt{A}[\mathtt{i}+1]$. When both are received it sends the total result to $\mathtt{A}[\mathtt{i}-1]$. Its global type is given as:

$$\mathtt{A}[\mathtt{i}:1..\mathtt{n}-1]\rightarrow\mathtt{C}:\mathtt{expr};\mathtt{A}[\mathtt{i}]\rightarrow\mathtt{A}[\mathtt{i}+1]:\mathtt{cont};\mathtt{C}\rightarrow\mathtt{A}[\mathtt{i}]:\mathtt{result};$$
$$\mathtt{A}[\mathtt{n}]\rightarrow\mathtt{C}:\mathtt{expr};\mathtt{C}\rightarrow\mathtt{A}[\mathtt{n}]:\mathtt{result};\mathtt{A}[\mathtt{i}:\mathtt{n}..2]\rightarrow\mathtt{A}[\mathtt{i}-1]:\mathtt{result};$$

Failure. `C` fails after processing a message form $\mathtt{A}[\mathtt{i}]$.

Result. All processes are recovered since they are all connected by the IO-dependencies and each output is depended by the previous input. Since all processes are restarted from the beginning, the usage of *protocol-recovery* for this use case does not result in a performance gain.

Listing 7.5: Ring Protocol Implementation

```
ready(_,S)  when S#state.num > 1 → {noreply, S}.
ready(_,S)  when S#state.num =:= 1 →
  role:send(S#state.role,role_name(2), ping, val)
  {noreply, S}

ping({msg, Val}, S) →
```

```
  % sort a list of sort_list(File)
  role:send(S#state.role,next_role(S), ping, val),
  {noreply, S};

ping({msg, Val}, S)
  when S#state.num =:= S#state.total →
   role:send(S#state.role, next_role(S), pong, val),
   {noreply, S}.

pong({msg, Val}, S)
  sort_list(get_numbers()), %failure point
  role:send(S#state.role,prev_role(S), pong, val),
  {noreply, S};

% The protocol ends
pong({msg, Val}, S)
  when S#state.count =:= 1 → {noreply, S}.
```

Listing 7.6: Resource Allocator Protocol Implementation

```

ready(_,S) →
   Num = get_random(),
  role:send(S#state.role, Num, start, val)
  {noreply, S};

started({From, val},S) →
  case is_enough() of
    no →  Next = get_random(),
     role:send(S#state.role, Next, exit, val),
      role:send(S#state.role, Next, start, val)
    yes →
      role:send(S#state.role, Next, exit, val)
  end
  {noreply, S};

start(_,S) →
  role:send(S#state.role, ?ARB, exit, my_num())
  {noreply, S};

exit(_,S) →
 {noreply, S};
```

Resource Allocator (Cigarette Smoker in [IS14]). This is a variant of the request-reply pattern. The implementation is given in Listing 7.6. The protocol has one Arbiter, $\mathtt{A[i]}$ and $\mathtt{n}$ processes, $\mathtt{C[i:1..n]}$. At random, $\mathtt{A}$ chooses a process $\mathtt{C[i]}$ and sends `request` to which $\mathtt{C[i]}$ replies with `request`. Its global type is given as follows:

$$\mu\mathtt{t.A} \to \mathtt{C[i} \in \mathtt{Random(1,n)]}\{\mathtt{start.C[i]} \to \mathtt{A:started;t}, \\ \mathtt{exit.C[i]} \to \mathtt{A:done;end}\}$$

Failure. The process $\mathtt{A}$ fails after sending `start` to $\mathtt{C[i]}$.

Result. At any given time only two processes are interacting. Thus only $\mathtt{C[i]}$ is affected by $\mathtt{A}$'s failure. By the same reason, if $\mathtt{C[i]}$ fails, only $\mathtt{A}$ is affected. The *protocol-recovery* strategy reduces the number of redundant restarts (since the processes do not perform work, while waiting for the message from $\mathtt{A}$), it is not relevant to measure the performance and thus this example is not presented Fig. 7.10 (bar Calc)). By applying *protocol-recovery* only two processes (at most) are restarted/notified in any case of failure.

Summary. Our *protocol-recovery* strategy outperforms Erlang *all-for-one* strategy when there are more intensive local computations; and protocols are more parallelised (i.e. they are more disconnected, hence there are less IO-dependencies from the failing node). It incurs a little overhead comparing to the case with no failure. The motivation of our work is not performance gain, but an automatic error prevention both at the process level (we assure processes are safe and conform to a protocol); and at the level of the supervision trees and dynamic process linking (we create supervision trees and link processes dynamically based on a protocol structure). As discussed in Section 7, writing the supervision trees and linking them to processes are a common source of errors as mentioned in [Arm02]. Specific performance optimisations (such as using persistent storage to cache the intermediate messages) and recovery actions (such as changing the database connection if this was the reason for the initial failure) are orthogonal/complementary concerns to our work and interesting topics for future investigation.

7.6. Related Work

Checkpoint and Recovery Algorithms. Our recovery method which calculates a recovery point is related to checkpoint techniques studied in software recovery for message-passing systems [EAWJ02] and for web-service choreographies [VM14b, VM14a]. The survey [EAWJ02] categorises their checkpointing protocols as uncoordinated, coordinated and communication-induced. Our approach belongs to the third class, where the coordination is done in a lazy fashion by piggy-backing control information on application messages where usually a local checkpoint is on the fly associated to a consistent global checkpoint. Both works [EAWJ02, VM14b] study how to reach a consistent global state. A global checkpoint consists of a set of local checkpoints, one for each process, from which a distributed computation can be restarted after a failure. In [EAWJ02], recovery relies on the knowledge and skillset of the user; and in [VM14b], the faulty service has to restart from the beginning. Our approach does not require these assumptions that rely on the user and the restart point is not required to be always from the beginning.

The work [VM14a] classifies different interaction patterns and defines a checkpoint based on executed patterns. The supported patterns are nesting, sequencing, concurrent and iterative. The checkpoints are generated using rules (heuristic) defined on the pattern structures. For example, for each request-reply a checkpoint is generated in the initiator after receiving the reply. Their tool takes a choreography of web services (given as a UML diagram) as an input and generates checkpointing locations in the given choreography (the initial UML diagram augmented with checkpoint locations) as outputs. Although such an approach may reduce the recovery time, it requires additional assumptions and restrictions at the code level. The targets of the applications studied in [EAWJ02, VM14b, VM14a] are clearly different from ours.

As a different approach, in [HYH08], binary session types are used at runtime to calculate the lost messages that can happen in an asynchronous delegation. When the connection is moved, the messages that may be lost could be regarded as a failure, and the session types from both sides provide checkpoints of recovery and are used to work out where the last synchronised state is, thus the relevant messages can be re-sent to get back to the correct state. This approach only accounts for delegations between two parties and is not applicable to recover multiple processes at once in the situation where messages are propagated following a global protocol and stored in mailboxes some of which are emptied at the failure.

Verification and Recovery in Erlang. A few practical works target to improve the recovery mechanisms of Erlang supervision structures. Currently, to obtain the process structure of an application, one must manually inspect the source code or rely on external documentation. To resolve this issue, the work [Nys09a, Nys09b] presents a fault-detection analysis on the process structure of an Erlang application. It checks that a supervision structure is constructed in a way that guarantees a recovery from process failures. It implements a tool which statically extracts a process trees from the source code and analyses it to determine the effect of process failures. It identifies the effect of a particular process failure on the entire process structure, that shows which processes will be terminated and restarted and whether the structure itself is restored to the situation before the failure. In our work, we define a protocol first for checking contracts between a client and servers, and implement an automatic recovery strategy based on that protocol. In this way, we can always guarantee both communication safety and an optimal recovery point statically for a given session typed program. Since theoretically it is possible to construct a global type from a set of local endpoint types [LTY15], it is an interesting future work to combine both approaches to develop a software life cycle for, e.g. legacy code applications.

The work [CFG11] proposes a runtime monitoring framework for Erlang called Elarva to detect a message which does not conform to their specification language. Their main aim is porting their former synchronous monitoring Larva [Lar] for object-oriented languages to asynchronous monitoring for Erlang. They do not aim to study efficiency of recovery.

Modelling recovery or Erlang based on session types. There are several works which use session types for modelling recovery and Erlang. The work [GP15] investigates the integration of constructs for runtime adaptations in a session type discipline and presents a session type framework for adaptable processes. The processes can be suspended, restarted, upgraded or discarded at runtime. The work extends a session π-calculus with primitives for located and updated processes. A located process $l[P]$ evolves autonomously until it is updated by an update action on l. The authors prove session consistency (the update does not disrupt active session behaviour) and give an encoding of the Erlang supervision trees (*one-for-one* strategy and *one-for-all* strategy) where located processes correspond to workers and updated processes to supervisors. To implement the supervision model, every time a worker establishes a session with a client, it also connects with its supervisor. The processes are updated only if there is no active session and therefore the type system ensures that no session is disrupted during active communication. Their work is theoretical, and neither the adaptation mechanism nor the typing system is implemented. Our approach proposes a new method for recovery with dynamic checking and repairing active sessions, and does not aim to model existing supervision strategies. We implemented and evaluated our algorithm in Erlang.

The work [AR12] investigates a use of session types for an analysis of deadlock freedom and typable communication in the presence of dynamically changing code. The session calculus is enriched with region annotation $f(t)$ which represents the code block to be updated. The arrival of an update is treated as an event external to the program. They proved processes are safe and live when the processes with empty queues are updated and typed by a set of local types projected from some global type. This means that processes are always updated from some initial point, unrelated to previous states. Our approach uses a more fine-grained, static dependency analysis of global types for performing an optimal recovery for Erlang without any assumption of queue conditions. In addition, our algorithm provides a way to recover processes from the middle of an existing session with respect to the current states (queues) of processes.

The work [MV11] proposed a concurrent fragment of Erlang enriched with sessions and session types. Messages are linked to a session via correlation sets (explicit identifiers, contained in the message), and clients create the required references and send them in the service invocation message. The developed typing system guarantees that all within-session messages have a chance of being received. The formalism is based on only binary sessions and does not support recovery. The typing system is not implemented.

7.7. Concluding Remarks

In this work, we propose an algorithm to analyse and extract causal dependencies from a given multiparty session protocol, and use it to ensure that communicating processes are recovered from consistent states in the presence of a failure. The messages to be resent and the set of processes to be notified are computed statically based on the dependencies in the process structures. To our best knowledge, this is the first work to apply session types not only for error-detection, but also for defining a recovery strategy and optimising recovery overhead. Our approach can automatically generate supervision structures in Erlang from types, and we implement the recovery strategy on top of runtime monitoring. Our programming model uses session type-based abstractions (1) to reduce complexity which is often introduced when process dependencies are manually specified and (2) to increase robustness of the system which is often undermined when unsound supervision strategies are used. We demonstrate that our recovery strategy can be applied to common concurrency patterns to reduce the recovery overhead as well as to guarantee safe executions after recovery.

8. Conclusion

The following recaps some of the main technical challenges tackled and the new session programming features developed through the course of this thesis.

Formalising Scribble In Chapter 2 we showed the correspondence between the protocol specification language Scribble and its formal counterpart, multiparty session types (MPST). We prove that a trace of global specification is equivalent to a trace of configuration of local specifications. In addition, we gave an encoding of local specifications to communicating state machines (CFSMs). This allowed us to reuse well-established results to show that a traces produced by a global specification is equivalent to a trace of a system of CFSMs. As a consequence, Scribble specifications can be used for local verification of processes without the need of runtime synchronisation.

Runtime framework for session programming Chapter 3 discusses the core design elements regarding implementation of a session-based API (*conversation API*) in a dynamically-typed language, Python. Communication safety of the whole system is guaranteed at runtime by monitors that check execution traces comply with an associated protocol. The central idea, enabling the verification of distributed Python components, is embedding session type specific information, called *conversation header*, as part of the message payload. Through a runtime layer for protocol management Scribble protocols are loaded and translated to CFSMs such that during a program execution, messages emitted by the programs are checked against corresponding CFSMs.

An advantage of the session programming proposed by this thesis is that it permits non-disruptive integration to existing code base, as demonstrated by our evaluation. We have integrated the Python library for communication programming in a real-world cyber-infrastructure project [OOIa]. Consideration for performance is crucial when considering runtime verification techniques. The benchmarks used to evaluate the performance of Python programs with embedded runtime monitors show that dynamic checking against Scribble specifications can be realised with negligible overhead.

Extensions The examination of session programming in practice (in particular the integration of the framework in a real world project) has led to the development of new Scribble features beyond the basic primitives of the theory.

Chapter 4 demonstrated a new construct, *interrupt*, which permits exception-like patterns in Scribble protocols. We integrate the extension in our runtime and extend the conversation API in Python. We introduced the notion of *session scopes*, which syntactically splits the protocol into sub-regions, allowing certain messaging to act on the regions as a whole and thus permitting controllable races, traditionally disallowed by the theory of session types. Scopes are mapped naturally to Python's exception handling mechanism (*with* statement). The generation of CFSMs from local protocols, presented in Chapter 3, is extended to support interruptible protocol scopes by generating nested CFSM structures. By annotating messages with scopes and by tracking the local progress of a protocol, the runtime at each endpoint in the session is able to resolve discrepancies in protocol state by discarding incoming messages that have become irrelevant due to an asynchronous interrupt. Although performed independently by each distributed monitor, this mechanism preserves global safety for the whole session of distributed components. We showed the soundness of our framework by proving *session fidelity*, asserting that the decentralised verification of a system always conforms to its global specification in the presence of nested interrupt, multiparty protocols and continuations to interruptible blocks. The extension allowed expressing a new range of communication patterns, mainly in the context of data streaming and publish-subscribe protocols, which could not be represented before.

In Chapter 5 we extended the runtime framework to support specification and verification of real-time protocols. We presented a syntactic checker for global protocols. The checker rules out protocols with unsatisfiable temporal constraints that would intrinsically force well-intentioned principals to either stop performing actions or continue by violating the protocols constraints. The implementation of temporal constraints is general as it uses an SMT solver to check time satisfiability formulas, defined over the structure of timed global protocol. We validated the usability of our verification by surveying temporal patterns distilled from the literature. Handling protocol violations, including time constraints, is naturally integrated with the standard Python mechanism for exception handling. We demonstrated that transparency and overhead are closely related in the timed scenario. We also gave an approximation on the number of interactions that can be performed before a violation occurs.

Actor programming In Chapter 6 we presented the design and implementation of Scribble-based verification for actor programming through the usage of the runtime mechanism and infrastructure presented in Chapter 6. Scribble verification is enabled on actor instances via code annotations. In particular, code annotations map session operations to operations on actor mailboxes, promoting an intuitive implementation methodology for Scribble specifications. The central idea is the notion of a *role*, which is a passive object inside an actor and encapsulates information used for Scribble verification. Two specific characteristics of the presented model are (i) actors are location-transparent and can connect to other actors without explicitly knowing their addresses and (ii) actors can implement multiple protocols, thus our framework allows cooperative multithreading inside a single actor. We surveyed a popular benchmark suit for actor programming and we showed that Scribble is expressive enough to capture typical concurrency patterns. In addition, we demonstrated detection of orphan messages, deadlock and unexpected termination for actor systems.

Recoverability In the last part of the thesis we applied the runtime monitoring for actors from Chapter 6 to Erlang programs. In addition, we developed a theory for recoverable session types. More precisely, we proposed an algorithm to analyse and extract causal dependencies from a Scribble protocol, and use it to ensure that Erlang processes are recovered from consistent states in the presence of a failure. We showed that our recovery strategy is not only sound, but also more efficient for common message-passing protocols comparing to a built-in static recovery strategy in Erlang.

8.1. Future Work

The accomplishments of this thesis offer a basics for the investigation of many further topics. The following outlines some of the most interesting directions related to the aspects of runtime verification and Scribble in general. We focus only on a few general extensions to Scribble and give potential applications for session monitoring. Future work, specific to the extensions presented in this thesis, has already been covered in the respective chapters.

Beyond Scribble annotations In Chapter 3 we demonstrated the usage of a Scribble extension for assertions on message payloads. The extension was presented as annotations on message interactions. A promising future research topic in this context is adding value-dependent session types to Scribble. As [TY16] shows value-dependent session types can be used for self-certifiable code. Another interesting topic for investigation would be to integrate a synthesis tool such as Rosetta [TB14] to reduce the number of runtime checks on assertions performed by the monitor. The idea is inspired by [Thi16], which demonstrates that finding a simplified runtime check on a dependently typed language with refinement types amounts to a synthesis problem.

We also believe it is promising to apply Scribble annotations in a broader context. Here we give two potential applications of Scribble annotations for developing additional verification and analysis techniques.

- Integration with object capabilities for memory management in the style of [CW16]. One of the challenges in an active objects based languages for parallel programming is memory management. The requirement of actor systems that actors do not share state has undesirable performance implications. A promising idea, explored in the Encore language [BCC$^+$15] is to share objects between actors by controlling capabilities (for access) and obligations (for deallocation) of a shared state. The integration of capability management and session types, as to enable checking the validity of passing an object between actors while avoiding expensive program analysis, is an interesting research problem. There are a number of technical components that need to be developed. First, session types should be extended with meta information regarding: (1) who can access an object (who has the capability of accessing an object in terms of reading and writing) and (2) who is responsible to delete an object (who has the obligation of deleting an object). Both capabilities and obligations can be represented as annotations on message interactions in a Scribble protocol, where objects are the payloads exchanged between actors (represented as roles in a protocol). Second, session type support for the ENCORE language is needed. We envision such integration as a library for annotating protocols on active object classes, inspired by the design of session actors, presented in Chapter 6. Examples of properties to be enforced by a session type system with capability annotations are: (1) before issuing a write capability for an object all read capabilities for that object should be released; (2) during write no other actors will use the object for read (only one actor can write to an object at any given time); (3) when a delete obligation is fulfilled the type system guarantees no dangling references for the objects.

- Integration with commitment logic for policy enforcement in the style of [DCS07]. The idea of a governance framework based on session monitors where agents exchange commitments attached to network messages originates from our collaboration with OOI [OOIa]. In such a framework session monitors are used as a base to ensure the validity of the agreed upon communicating parties. Message interactions are annotated with commitments associated to the label of an interaction, or represented as a function of the label and the message payload. From session monitor perspective, commitments are treated as a black box. Once the protocol is checked by the session monitor, annotations are passed for processing to their respective library. Ensuring consistency of commitments usage in Scribble protocols is challenging and requires the addition of dependent types to the language.

Session types for communication optimisations Static and dynamic performance optimisations based on session types are promising topics for future investigation. In Chapter 7 we already explored one application of session types for potential optimisations in cases of failure. The rich communication structure offered by Scribble protocols invites optimisations of other communication mechanism. One example is reducing the size of the transferred metadata. Message payloads with matching conversation headers can be packed together, therefore decreasing the size of the overall communication cost. Although the latter optimisation is directly applicable to our framework for runtime verification, buffering and batch processing of messages might create a bottleneck and increase the monitor overhead and thus requires careful examination where several heuristics of the system are taken into consideration. Another potential area for optimisation is for reducing latency between communicating parties by generating topologies from protocols. Roles that communicate frequently can be positioned in the same federation clusters (in case of middleware), nodes (in case of cloud computing), or cores (in case of multi-core programming). The work [FDY14] explores the latter idea in the context of multi-core systems suggesting session types annotations on classes for static calculation of communication cost for sending and receiving of objects. The monitoring infrastructure in Erlang presented in Chapter 7 invites for exploring a similar technique where processes can be dynamically assigned Erlang nodes depending on the protocols they are participating.

Session types in practice Here we discuss some preliminary ideas to streamline the adoption of session types in mainstream projects and languages. We are currently developing a global repository of protocols described in Scribble, along with a web service for protocol downloading. Note that, throughout the thesis we do not dictate how types/protocols are offered to the monitor leaving this technical detail unexplored. As long as offered services come with a protocol, monitoring can then commence as described in this thesis once a connection has been established. However, having an infrastructure for protocol retrieval is an important consideration when applying session types in the context of a heterogeneous system with processes written in different languages. In addition, as part of the ABCD project [ABC] we are working to establish a common benchmark suit for session programming, which will allow a comparison of session type implementations (both based on language extensions and on APIs). The aim is to demonstrate the robustness, functionality, and overall applicability of session types (and Scribble, in particular) through a diverse overview of use cases, ranging over various domains and exhibiting different computational needs, as well as to facilitate the integration of the tools and technologies developed in the session type community. For such an integration, having a common message format, as the abstract representation of *session messages* proposed in Chapter 3, is essential. We believe the project can be used to unify session type implementations towards a practical session middleware framework connecting incompatible communicating programs.

Bibliography

URL resources last accessed on 2016-10-20.

[AAC$^+$05] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to AspectJ. *SIGPLAN Not.*, 40(10):345–364, 2005.

[ABC] ABCD project webpage. `http://groups.inf.ed.ac.uk/abcd/`.

[Act] Cell - actors for celery. `http://cell.readthedocs.org/`.

[ADM12] Davide Ancona, Sophia Drossopoulou, and Viviana Mascardi. Automatic generation of self-monitoring mass from multiparty global session types in Jason. In *DALT'12*. Springer, 2012.

[AFK87] Krzysztof R. Apt, Nissim Francez, and Shmuel Katz. Appraising fairness in languages for distributed programming. In *POPL*, pages 189–198. ACM, 1987.

[Agh86] Gul Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986.

[AGMK10] S. Akshay, Paul Gastin, Madhavan Mukund, and K. Narayan Kumar. Model checking time-constrained scenario-based specifications. In *FSTTCS*, volume 8 of *LIPIcs*, pages 204–215. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010.

[akk] Akka - Scala actor library. http://akka.io/.

[AMQ] Advanced Message Queuing Protocol homepage. `http://www.amqp.org/`.

[AMST97] Gul Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7:1–72, 1997.

[AR12] Gabrielle Anderson and Julian Rathke. Dynamic software update for message passing programs. In *APLAS*, volume 7705 of *LNCS*, pages 207–222. Springer, 2012.

[Arm02] Joe Armstrong. Getting Erlang to talk to the outside world. In *ACM SIGPLAN Workshop on Erlang*, ERLANG, pages 64–72. ACM, 2002.

[ATdM07] Pavel Avgustinov, Julian Tibble, and Oege de Moor. Making trace monitors feasible. *SIGPLAN Not.*, 42(10):589–608, 2007.

[BBNL08] Michele Boreale, Roberto Bruni, Rocco Nicola, and Michele Loreti. Sessions and pipelines for structured service programming. In *FMOODS '08*, volume 5051 of *LNCS*, pages 19–38. Springer, 2008.

[BCC$^+$15] Stephan Brandauer, Elias Castegren, Dave Clarke, Kiko Fernandez-Reyes, Einar Broch Johnsen, Ka I Pun, Silvia Lizeth Tapia Tarifa, Tobias Wrigstad, and Albert Mingkun Yang. Parallel objects for multicores: A glimpse at the parallel language encore. In *SFM*, volume 9104 of *LNCS*, pages 1–56. Springer, 2015.

[BCD$^+$08] Lorenzo Bettini, Mario Coppo, Loris D'Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. Global progress in dynamically interleaved multiparty sessions. In *CONCUR*, volume 5201 of *LNCS*, pages 418–433. Springer, 2008.

[BCD$^+$13] Laura Bocchi, Tzu-Chun Chen, Romain Demangeon, Kohei Honda, and Nobuko Yoshida. Monitoring networks through multiparty session types. In *FMOODS*, volume 7892 of *LNCS*, pages 50–65. Springer, 2013.

[BDY12] Laura Bocchi, Romain Demangeon, and Nobuko Yoshida. A multiparty multi-session logic. In *TGC*, volume 8191 of *LNCS*, pages 97–111. Springer, 2012.

[BFM98] Howard Bowman, Giorgio P. Faconti, and Mieke Massink. Specification and verification of media constraints using UPPAAL. In *DSV-IS*, pages 261–277. Springer, 1998.

[BGG04] Luciano Baresi, Carlo Ghezzi, and Sam Guinea. Smart monitors for composed services. In *ICSOC*, pages 193–202. ACM, 2004.

[BHTY10] Laura Bocchi, Kohei Honda, Emilio Tuosto, and Nobuko Yoshida. A theory of design-by-contract for distributed multiparty interactions. In *CONCUR*, volume 6269 of *LNCS*, pages 162–176. Springer, 2010.

[BLY15] Laura Bocchi, Julien Lange, and Nobuko Yoshida. Meeting deadlines together. In *CONCUR*, volume 42 of *LIPIcs*, pages 283–296. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

[BLZ03] Laura Bocchi, Cosimo Laneve, and Gianluigi Zavattaro. A calculus for long-running transactions. In *FMOODS*, volume 2884 of *LNCS*, pages 124–138. Springer, 2003.

[BMA15] Daniela Briola, Viviana Mascardi, and Davide Ancona. Distributed runtime verification of JADE multiagent systems. In *IDC 2014*, volume 570 of *SCI*, pages 81–91. Springer, 2015.

[BY07] Martin Berger and Nobuko Yoshida. Timed, distributed, probabilistic, typed processes. In *APLAS*, volume 4807 of *LNCS*, pages 158–174. Springer, 2007.

[BYY14a] Laura Bocchi, Weizhen Yang, and Nobuko Yoshida. Timed multiparty session types. In *CONCUR*, volume 8704 of *LNCS*, pages 419–434. Springer, 2014.

[BYY14b] Laura Bocchi, Weizhen Yang, and Nobuko Yoshida. Timed multiparty session types. Technical Report 2014/3, Department of Computing, Imperial College London, May 2014.

[BZ83] Daniel Brand and Pitro Zafiropulo. On communicating finite-state machines. *J. ACM*, 30(2):323–342, 1983.

[Cai08] Luís Caires. Spatial-behavioral types for concurrency and resource control in distributed systems. *Theor. Comput. Sci.*, 402(2-3):120–141, 2008.

[Car09] Marco Carbone. Session-based choreography with exceptions. *Electr. Notes Theor. Comput. Sci.*, 241:35–55, 2009.

[CBD$^+$12] Tzu-Chun Chen, Laura Bocchi, Pierre-Malo Deniélou, Kohei Honda, and Nobuko Yoshida. Asynchronous distributed monitoring for multiparty session enforcement. In *TGC*, volume 7173 of *LNCS*, pages 25–45. Springer, 2012.

[CDVM11] María-Emilia Cambronero, Gregorio Díaz, Valentín Valero, and Enrique Martínez. Validation and verification of web services choreographies by using timed automata. *J. Log. Algebr. Program.*, 80(1):25 – 49, 2011.

[Cel] Celery. `http://www.celeryproject.org//`.

[CFG11] Christian Colombo, Adrian Francalanza, and Rudolph Gatt. Elarva: A monitoring tool for Erlang. In *RV*, volume 7186 of *LNCS*, pages 370–374. Springer, 2011.

[CGY10] Sara Capecchi, Elena Giachino, and Nobuko Yoshida. Global escape in multiparty session. In *FSTTCS*, volume 8 of *LIPICS*, pages 338–351. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010.

[Che13] Tzu-Chun Chen. *Theories for Session-based Governance for Large-scale Distributed Systems*. PhD thesis, Queen Mary, University of London, 2013.

[CHY08] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured interactional exceptions in session types. In *CONCUR*, volume 5201 of *LNCS*, pages 402–417. Springer, 2008.

[CKGJ13] Saoussen Cheikhrouhou, Slim Kallel, Nawal Guermouche, and Mohamed Jmaiel. A survey on time-aware business process modeling. In *ICEIS (3)*, pages 236–242. SciTePress, 2013.

[CR07] Feng Chen and Grigore Rosu. MOP:An Efficient and Generic Runtime Verification Framework. In *OOPSLA*, volume 42, pages 569–588. ACM, 2007.

[Cra] Silvia Crafa. Behavioural types for actor systems. `arXiv:1206.1687`.

[CVB$^+$16] Tzu-Chun Chen, Malte Viering, Andi Bejleri, Lukasz Ziarek, and Patrick Eugster. A type theory for robust failure handling in distributed systems. In *FORTE*, volume 9688 of *LNCS*, pages 96–113. Springer, 2016.

[CW16] Elias Castegren and Tobias Wrigstad. Reference capabilities for concurrency control. In *ECOOP*, volume 56 of *LIPIcs*, pages 5:1–5:26. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

[dBdGJ$^+$14] Frank S. de Boer, Stijn de Gouw, Einar Broch Johnsen, Andreas Kohn, and Peter Y. H. Wong. Run-time assertion checking of data- and protocol-oriented properties of Java programs: An industrial case study. *T. Aspect-Oriented Software Development*, 11:1–26, 2014.

[DCS07] Nirmit Desai, Amit K. Chopra, and Munindar P. Singh. Representing and reasoning about commitments in business processes. AAAI, pages 1328–1333. AAAI Press, 2007.

[DH12] Romain Demangeon and Kohei Honda. Nested protocols in session types. In *CONCUR*, volume 7454 of *LNCS*, pages 272–286. Springer, 2012.

[DHH$^+$15] Romain Demangeon, Kohei Honda, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. Practical interruptible conversations: distributed dynamic verification with multiparty session types and Python. *FMSD*, 46(3):197–225, 2015.

[DY11] Pierre-Malo Deniélou and Nobuko Yoshida. Dynamic multirole session types. In *POPL*, pages 435–446, 2011.

[DY12] Pierre-Malo Deniélou and Nobuko Yoshida. Multiparty session types meet communicating automata. In *ESOP*, volume 7211 of *LNCS*, pages 194–213. Springer, 2012.

[DY13] Pierre-Malo Deniélou and Nobuko Yoshida. Multiparty compatibility in communicating automata: Characterisation and synthesis of global session types. In *ICALP*, volume 7966 of *LNCS*, pages 174–186. Springer, 2013.

[EAWJ02] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, 2002.

[FDY14] Juliana Franco, Sophia Drossopoulou, and Nobuko Yoshida. Calculating communication costs with sessions types and sizes. In *ICCSW*, volume 43 of *OASICS*, pages 50–57. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2014.

[FG00] Cédric Fournet and Georges Gonthier. The join calculus: A language for distributed mobile programming. In *APPSEM*, LNCS, pages 268–332. Springer-Verlag, 2000.

[FHR13] Yliès Falcone, Klaus Havelund, and Giles Reger. A tutorial on runtime verification. In *Engineering Dependable Software Systems*, volume 34 of *NATO*, pages 141–175. IOS Press, 2013.

[GBE07] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous Java performance evaluation. *SIGPLAN Not.*, 42(10):57–76, 2007.

[GCN$^+$07] Yuan Gan, Marsha Chechik, Shiva Nejati, Jon Bennett, Bill O'Farrell, and Julie Waterhouse. Runtime monitoring of web service conversations. In *CASCON*, pages 42–57. ACM, 2007.

[GDNP97] Carlos Gregorio-Rodríguez, Luis Fernando Llana Díaz, Manuel Núñez, and Pedro Palao-Gostanza. Testing semantics for a probabilistic-timed process algebra. In *AMAST*, volume 1231 of *LNCS*, pages 353–367, 1997.

[GDZ12] Nawal Guermouche and Silvano Dal-Zilio. Towards timed requirement verification for service choreographies. In *CollaborateCom*, pages 117–126. IEEE, 2012.

[GG07] Carlo Ghezzi and Sam Guinea. Run-time monitoring in service-oriented architectures. In *Test and Analysis of Web Services*, pages 237–264. Springer, 2007.

[GMNK09] Paul Gastin, Madhavan Mukund, and K. Narayan Kumar. Reachability and boundedness in time-constrained MSC graphs. In *Perspectives in Concurrency Theory*, pages 157–183. Universities Press, 2009.

[GP15] Cinzia Di Giusto and Jorge A. Pérez. Disciplined structured communications with disciplined runtime adaptation. *Sci. Comput. Program.*, 97:235–265, 2015.

[GVR$^+$10] Simon J. Gay, Vasco Thudichum Vasconcelos, António Ravara, Nils Gesbert, and Alexandre Z. Caldeira. Modular session types for distributed object-oriented programming. In *POPL*, pages 299–312. ACM, 2010.

[HBH$^+$10] Sylvain Hallé, Tevfik Bultan, Graham Hughes, Muath Alkhalaf, and Roger Villemaire. Runtime verification of web service interface contracts. *Computer*, 43(3):59–66, 2010.

[HHI13] Ludovic Henrio, Fabrice Huet, and Zsolt István. Multi-threaded active objects. In *COORDINATION*, volume 7890 of *LNCS*, pages 90–104. Springer, 2013.

[HHN$^+$14a] Kohei Honda, Raymond Hu, Rumyana Neykova, Tzu-Chun Chen, Romain Demangeon, Pierre-Malo Deniélou, and Nobuko Yoshida. Structuring communication with session types. In *COB*, volume 8665 of *LNCS*, pages 105–127. Springer, 2014.

[HHN$^+$14b] Kohei Honda, Raymond Hu, Rumyana Neykova, Tzu-Chun Chen, Romain Demangeon, Pierre-Malo Deniélou, and Nobuko Yoshida. Structuring communication with session types. In *COB 2014*, volume 8665 of *LNCS*, pages 105–127. Springer, 2014.

[HMB$^+$11] Kohei Honda, Aybek Mukhamedov, Gary Brown, Tzu-Chun Chen, and Nobuko Yoshida. Scribbling interactions with a formal foundation. In *ICDCIT*, volume 6536 of *LNCS*, pages 55–75. Springer, 2011.

[HNYD13] Raymond Hu, Rumyana Neykova, Nobuko Yoshida, and Romain Demangeon. Practical Interruptible Conversations: Distributed Dynamic Verification with Session Types and Python. In *RV'13*, volume 8174 of *LNCS*, pages 130–148. Springer, 2013.

[HWT14] Jiansen He, Philip Wadler, and Philip Trinder. Typecasting actors: From Akka to TAkka. In *SCALA@ECOOP Workshop 2014*, pages 23–33. ACM, 2014.

[HY16] Raymond Hu and Nobuko Yoshida. Hybrid session verification through endpoint API generation. In *FASE*, volume 9633 of *LNCS*, pages 401–418. Springer, 2016.

[HYC08] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *POPL'08*, pages 273–284. ACM, 2008.

[HYH08] Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-Based Distributed Programming in Java. In *ECOOP'08*, volume 5142 of *LNCS*, pages 516–541. Springer, 2008.

[IS12] Shams M. Imam and Vivek Sarkar. Integrating task parallelism with actors. *SIGPLAN Not.*, 47(10):753–772, 2012.

[IS14] Shams Imam and Vivek Sarkar. Savina - An Actor Benchmark Suite: Enabling Empirical Evaluation of Actor Libraries. In *AGERE*, pages 67–80. ACM, 2014.

[Jas] Jass Home Page. `http://modernjass.sourceforge.net/`.

[JGP16] Limin Jia, Hannah Gommerstadt, and Frank Pfenning. Monitors and blame assignment for higher-order session types. In *POPL*, pages 582–594. ACM, 2016.

[JP14] Svetlana Jakšić and Luca Padovani. Exception handling for copyless messaging. In *PPDP*, volume 84, pages 22–51, 2014.

[KCD$^+$09] Slim Kallel, Anis Charfi, Tom Dinkelaker, Mira Mezini, and Mohamed Jmaiel. Specifying and monitoring temporal properties in web services compositions. In *ECOWS*, pages 148–157. IEEE Computer Society, 2009.

[KMM07] Ingolf H. Krüger, Michael Meisinger, and Massimiliano Menarini. Runtime verification of interactions: from MSCs to aspects. RV, pages 63–74. Springer, 2007.

[KMM10] Ingolf H. Krüger, Michael Meisinger, and Massimiliano Menarini. Interaction-based runtime verification for systems of systems integration. *J. Log. Comput.*, 20(3):725–742, 2010.

[KY06] Pavel Krcal and Wang Yi. Communicating timed automata: The more synchronous, the more difficult to verify. In *CAV*, volume 4144 of *LNCS*, pages 243–257, 2006.

[Lar] Larva project. `http://www.cs.um.edu.mt/svrg/Tools/LARVA/`.

[LHJ05] Zheng Li, Jun Han, and Yan Jin. Pattern-based specification and validation of web services interaction properties. In *ICSOC*, volume 3826 of *LNCS*, pages 73–86. Springer, 2005.

[LJH06] Zheng Li, Yan Jin, and Jun Han. A runtime monitoring and validation framework for web service interactions. In *ASWEC*, pages 70–79. IEEE, 2006.

[LP12] Hugo A. López and Jorge A. Pérez. Time and exceptional behavior in multiparty structured interactions. In *WS-FM*, volume 7176 of *LNCS*, pages 48–63. Springer, 2012.

[LPT07a] Alessandro Lapadula, Rosario Pugliese, and Francesco Tiezzi. C-clock-ws: A timed service-oriented calculus. In *ICTAC*, volume 4711 of *LNCS*, pages 275–290. Springer, 2007.

[LPT07b] Alessandro Lapadula, Rosario Pugliese, and Francesco Tiezzi. A calculus for orchestration of web services. In *ESOP*, volume 4421 of *LNCS*, pages 33–47. Springer, 2007.

[LTY15] Julien Lange, Emilio Tuosto, and Nobuko Yoshida. From communicating machines to graphical choreographies. In *POPL*, pages 221–232. ACM, 2015.

[LZ02] Jeremy Y. Lee and John Zic. On modeling real-time mobile processes. *Aust. Comput. Sci. Commun.*, 24(1):139–147, January 2002.

[LZ05] Cosimo Laneve and Gianluigi Zavattaro. Foundations of web transactions. In *FOSSACS*, volume 3411 of *LNCS*, pages 282–298. Springer, 2005.

[MU00] Naftaly H. Minsky and Victoria Ungureanu. Law-governed interaction: a coordination and control mechanism for heterogeneous distributed systems. *ACM Trans. Softw. Eng. Methodol.*, 9:273–305, July 2000.

[MV11] Dimitris Mostrous and Vasco Thudichum Vasconcelos. Session Typing for a Featherweight Erlang. In *COORDINATION*, volume 6721 of *LNCS*, pages 95–109. Springer, 2011.

[NBY14] Rumyana Neykova, Laura Bocchi, and Nobuko Yoshida. Timed runtime monitoring for multiparty conversations. In *BEAT*, volume 162 of *EPTCS*, pages 19–26, 2014.

[NCY15] Nicholas Ng, Jose G.F. Coutinho, and Nobuko Yoshida. Protocols by default: Safe MPI code generation based on session types. In *CC 2015*, volume 9031 of *LNCS*, pages 212–232. Springer, 2015.

[Ney13] Rumyana Neykova. Session types go dynamic or how to verify your Python conversations. In *PLACES*, volume 137 of *EPTCS*, pages 95–102, 2013.

[NY14a] Rumyana Neykova and Nobuko Yoshida. Multiparty session actors. In *COORDINATION*, volume 8459 of *LNCS*, pages 131–146. Springer, 2014.

[NY14b] Rumyana Neykova and Nobuko Yoshida. Multiparty session actors. In *PLACES*, volume 155 of *EPTCS*, pages 32–37, 2014.

[NY14c] Nicholas Ng and Nobuko Yoshida. Pabble: Parameterised scribble for parallel programming. In *PDP 2014*, pages 707–714. IEEE, 2014.

[NY17a] Rumyana Neykova and Nobuko Yoshida. Let it Recover: Multiparty Protocol-Induced Recovery. In *CC*. ACM, 2017. To appear.

[NY17b] Rumyana Neykova and Nobuko Yoshida. Multiparty Session Actors. *LMCS*, 2017. To appear.

[NY17c] Rumyana Neykova and Nobuko Yoshida. Timed Runtime Monitoring for Multiparty Conversations. *FAOC*, 2017. To appear.

[NYH12] Nicholas Ng, Nobuko Yoshida, and Kohei Honda. Multiparty session C: Safe parallel programming with message optimisation. In *TOOLS'12*, LNCS, pages 202–218. Springer, 2012.

[NYH13] Rumyana Neykova, Nobuko Yoshida, and Raymond Hu. SPY: Local Verification of Global Protocols. In *RV*, volume 8174 of *LNCS*, pages 358–363. Springer, 2013.

[Nys09a] Jan Henry Nyström. *Analysing Fault Tolerance for ERLANG Applications*. PhD thesis, ACTA UNIVERSITATIS UPSALIENSI, 2009.

[Nys09b] Jan Henry Nyström. Automatic assessment of failure recovery in Erlang applications. In *ACM SIGPLAN Workshop on ERLANG*, ERLANG, pages 23–32. ACM, 2009.

[OOIa] Ocean Observatories Initiative. `http://www.oceanobservatories.org/`.

[OOIb] OOI. `https://confluence.oceanobservatories.org/display/CIDev/Identify+required+Scribble+extensions+for+advanced+scenarios+of+R3+COI`.

[OOIc] OOI codebase. `https://github.com/ooici/pyon`.

[OOId] OOI COI governance framework. `https://confluence.oceanobservatories.org/display/syseng/CIAD+COI+OV+Governance+Framework`.

[pyt] Timed Conversation API in Python. `http://www.doc.ic.ac.uk/~rn710/TimeApp.html`.

[RW95] C. M. F. Rubira and Zhixue Wu. Fault tolerance in concurrent object-oriented software through coordinated error recovery. In *FTCS*, pages 499–. IEEE Computer Society, 1995.

[RYC$^+$06] Shangping Ren, Yue Yu, Nianen Chen, Kevin Marth, Pierre-Etienne Poirot, and Limin Shen. Actors, roles and coordinators - a coordination model for open distributed and embedded systems. In *COORDINATION*, volume 4038 of *LNCS*, pages 247–265. Springer, 2006.

[Sal10] Gwen Salaün. Analysis and verification of service interaction protocols - a brief survey. In *TAV-WEB*, volume 35 of *EPTCS*, pages 75–86, 2010.

[Sav] Savina source. `https://github.com/shamsmahmood/savina`.

[Scra] Learn you some Erlang. `http://learnyousomeerlang.com/`.

[scrb] Scribble project home page. `http://www.scribble.org`.

[Se] Savara examples. `http://www.jboss.org/savara/downloads`.

[ses] SJ homepage. `http://www.doc.ic.ac.uk/~rhu/sessionj.html`.

[seta] Project page for monitoring. `http://www.doc.ic.ac.uk/~rn710/mon`.

[setb] Project page for session actors. `http://www.doc.ic.ac.uk/~rn710/sactor`.

[SG13] Neda Saeedloei and Gopal Gupta. Timed π-calculus. In *TGC*, volume 8358 of *LNCS*, pages 119–135. Springer, 2013.

[SLVW08] Martin Sulzmann, Edmund S. L. Lam, and Peter Van Weert. Actors with multi-headed message receive patterns. In *COORDINATION*, LNCS, pages 315–330. Springer, 2008.

[smt] The Simple Mail Transfer Protocol. `http://tools.ietf.org/html/rfc5321`.

[SPH10] Jan Schäfer and Arnd Poetzsch-Heffter. JCoBox: Generalizing active objects to concurrent components. In *ECOOP*, pages 275–299, Berlin, Heidelberg, 2010. Springer-Verlag.

[SS] Session Scala. `http://code.google.com/p/session-scala/`.

[STM14] Christophe Scholliers, Éric Tanter, and Wolfgang De Meuter. Parallel actor monitors: Disentangling task-level parallelism from data partitioning in the actor model. *Sci. Comput. Program.*, 80:52–64, 2014.

[TB14] Emina Torlak and Rastislav Bodík. A lightweight symbolic virtual machine for solver-aided host languages. In *PLDI*, volume 49, pages 530–541. ACM, 2014.

[TDJ13] Samira Tasharofi, Peter Dinges, and Ralph E. Johnson. Why do Scala developers mix the actor model with other concurrency models? In *ECOOP*, volume 7920 of *LNCS*, pages 302–326. Springer, 2013.

[Thi14] Peter Thiemann. Session types with gradual typing. In *TGC*, volume 8902 of *LNCS*, pages 144–158. Springer, 2014.

[Thi16] Peter Thiemann. A delta for hybrid type checking. In *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, volume 9600 of *LNCS*, pages 411–432. Springer, 2016.

[TIRL03] Ferda Tartanoglu, Valérie Issarny, Alexander B. Romanovsky, and Nicole Lévy. Coordinated forward error recovery for compositeweb services. In *SRDS*, pages 167–176. IEEE Computer Society, 2003.

[Tri99] Stavros Tripakis. Verifying progress in timed systems. In *Formal Methods for Real-Time and Probabilistic Systems*, volume 1601 of *LNCS*, pages 299–314. Springer, 1999.

[TY16] Bernardo Toninho and Nobuko Yoshida. Certifying data in multiparty session types. In *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, volume 9600 of *LNCS*, pages 433–458. Springer, 2016.

[UPP] UPPAAL tool website. `http://www.uppaal.org/`.

[VA01] Carlos Varela and Gul Agha. Programming dynamically reconfigurable open systems with SALSA. *SIGPLAN Not.*, 36(12):20–34, 2001.

[VCS08] Hugo Torres Vieira, Luís Caires, and João Costa Seco. The conversation calculus: A model of service-oriented computation. In *ESOP*, volume 4960 of *LNCS*, pages 269–283. Springer, 2008.

[VM14a] A. Vani Vathsala and Hrushikesha Mohanty. Interaction patterns based checkpointing of choreographed web services. In *PESOS*, pages 28–37. ACM, 2014.

[VM14b] A. Vani Vathsala and Hrushikesha Mohanty. A survey on checkpointing web services. In *PESOS*, pages 11–17. ACM, 2014.

[WIH11] Kenji Watahiki, Fuyuki Ishikawa, and Kunihiko Hiraishi. Formal verification of business processes with temporal and resource constraints. In *SMC*, pages 1173–1180. IEEE, 2011.

[XRR98] Jie Xu, Alexander Romanovsky, and Brian Randell. Coordinated exception handling in distributed object systems: From model to system implementation. In *ICDCS*, pages 12–21. IEEE Computer Society, 1998.

[YHE02] Wei Ye, John S. Heidemann, and Deborah Estrin. An energy-efficient MAC protocol for wireless sensor networks. In *INFOCOM*, volume 3, pages 1567–1576. IEEE, 2002.

[YHNN13] Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. The Scribble protocol language. In *TGC 2013*, volume 8358 of *LNCS*, pages 22–41. Springer, 2013.

[Z3C] Z3 SMT solver. `http://z3.codeplex.com/`.

[ZSM07] Wenxuan Zhang, Constantin Serban, and Naftaly Minsky. Establishing global properties of multi-agent systems via local laws. In *E4MAS*, LNCS, pages 170–183. Springer, 2007.

A. Proofs for Chapter 4

This appendix includes a full proof of Theorem 4.3.3.

Theorem 4.3.3 (Session fidelity) *If Δ corresponds to $G_1,\dots,G_n$ and $\Delta,\varepsilon \to^* \Delta',\Sigma'$, there exists $\Delta',\Sigma' \to^* \Delta'',\varepsilon$ such that Δ'' corresponds to $G_1'',\dots,G_n''$ which is a derivative of $G_1,\dots,G_n$.*

Proof. We prove that if there is an intermediate correspondence between Δ,Σ and $G_1,\dots,G_n$ and if $\Delta,\varepsilon \to \Delta',\Sigma'$, then there is an intermediate correspondence Δ'' and $G_1'',\dots,G_n''$ which is a derivative of $G_1,\dots,G_n$.

We use Ω,Θ alongside Δ to denote session environment. According to the derivative definition above, we extend the notion of evaluation contexts to global types.

Case (`Out`) is trivial from the first rule of intermediate correspondence.

Case (`EOut`)**.** We have $\Delta = \Theta, s[\mathtt{A}] : E^{\mathtt{c}_0}[\{|T|\}^{\mathtt{c}} \triangleright \langle \mathtt{A}?l\rangle;T']$. Correspondence gives

- $\Delta = \Theta_0, s[\mathtt{A}] : E^{\mathtt{c}_0}[\{|T|\}^{\mathtt{c}} \triangleright \langle \mathtt{A}?l\rangle;T'], \prod_{1\le i\le n} s[\mathtt{A}_i] : E_i{}^{\mathtt{c}_i}[\{|T_i|\}^{\mathtt{c}} \triangleright \langle \mathtt{A}?l\rangle;T_i']$ and
- $\Sigma = \Sigma_1,\Sigma_0$ with $\Theta_0;\Sigma_0$

corresponding to $G_2,\dots,G_n$ and $(\Delta-\Theta');\Sigma_1$ corresponding to G_1. We know that $\Sigma' = \Sigma_0, \prod_{1\le i\le n} s[\mathtt{A}_i] : h_i.\mathtt{c}^{\mathtt{I}}[\mathtt{A},\mathtt{A}_i]\langle l\rangle$. Concluding is easy using the second rule of intermediate correspondence with $k=0$.

Case (`In`)**.** We assume $\Delta = \Theta, s[\mathtt{A}'] : E^{\mathtt{c}}[\mathtt{A}?\{l_i.T_i\}]$ and $\Sigma = \Sigma_0, s[\mathtt{A}'] : h.\mathtt{c}[\mathtt{A},\mathtt{A}']\langle l_j\rangle$. We know there exists $G_1,\dots G_n$ and Δ_0 such that $\Delta_0 = \Theta_1,\dots,\Theta_n$ with $\Theta_i = \bigcup_{\mathtt{A}\in G_i} G_i \uparrow^{\mathtt{A}}$.

Without loss of generality we have $\Theta_1 = s[\mathtt{A}'] : E^{\mathtt{c}}[\mathtt{A}?\{l_i.T_i\}], \Theta_1'$. By the rules of projection, it means $G_1 = \mathtt{A}\to\mathtt{A}' : \{l_j.G_j\}_{j\in J}$, implying $\Theta_1' = s[\mathtt{A}'] : E^{\mathtt{c}'}[\mathtt{A}?\{l_i.T_i\}], \Theta_1''$. So we have

$$\Delta' = \Omega, s[\mathtt{A}] : E^{\mathtt{c}}[T_j], s[\mathtt{A}'] : E^{\mathtt{c}'}[\mathtt{A}?\{l_i.T_i\}], \Theta_1' \text{ and } \Sigma' = \Sigma_0, s[\mathtt{A}'] : h.\mathtt{c}[\mathtt{A},\mathtt{A}']\langle l_j\rangle.$$

We apply use $(\mathtt{In})$ to conclude, using the projection rule on G_j.

Case $(\mathtt{EIn}_1)$**.** We pose $\mathtt{A} = \mathtt{A}_{k+1}$. We have $\Sigma = \Sigma', \mathtt{c}^{\mathtt{I}}[\mathtt{A}_0, \mathtt{A}_{k+1}]\langle l\rangle.h$ and $\varphi(\Sigma', \mathtt{c})$. Then let us define $\Delta = \Theta, s[\mathtt{A}_{k+1}] : E^{\mathtt{c}_0}[\{\!|T_{k+1}|\!\}^{\mathtt{c}} \triangleright \langle \mathtt{A}_0?l\rangle; T']$ and $\Delta' = \Theta, s[\mathtt{A}_{k+1}]E^{\mathtt{c}_0}[\{\!|\,\|T\|\,|\!\}^{\mathtt{c}} \triangleright \langle \mathtt{A}_0?l\rangle; T']$. Without loss of generality we suppose $\Delta; \Sigma$ corresponds to G. We deduce that

- $G = F\{\!|G_0|\!\}^{\mathtt{c}}\langle l \;\mathtt{by}\; \mathtt{A}\rangle; G'_{\_}$;
  $\Delta = s[\mathtt{A}_0] : E^{-}[\{\!|T|\!\}^{\mathtt{c}} \triangleright \langle \mathtt{A}'?l\rangle; \_], \prod_{1\leq i\leq k} s[\mathtt{A}_i] : E_i^{-}[\{\!|\,\|T_i\|\,|\!\}^{\mathtt{c}} \triangleright \langle \mathtt{A}'?l\rangle; \_], \prod_{k+1\leq j\leq n} s[\mathtt{A}_j] : E_i^{-}[\{\!|T_j|\!\}^{\mathtt{c}} \triangleright \langle \mathtt{A}'?l\rangle; \_]$; and
- $\Sigma = \Sigma_0, \prod_{k+1\leq j\leq n} s[\mathtt{A}_j] : \mathtt{c}^{\mathtt{I}}[\mathtt{A}, \mathtt{A}_j]\langle l\rangle.h_j$.

Thus we have

- $\Delta' = s[\mathtt{A}_0] : E^{-}[\{\!|T|\!\}^{\mathtt{c}} \triangleright \langle \mathtt{A}'?l\rangle; \_], \prod_{1\leq i\leq k+1} s[\mathtt{A}_i] : E_i^{-}[\{\!|\,\|T_i\|\,|\!\}^{\mathtt{c}} \triangleright \langle \mathtt{A}'?l\rangle; \_], \prod_{k+2\leq j\leq n} s[\mathtt{A}_j] : E_i^{-}[\{\!|T_j|\!\}^{\mathtt{c}} \triangleright \langle \mathtt{A}'?l\rangle; \_]$ and
- $\Sigma' = \Sigma_0, \prod_{k+2\leq j\leq n} s[\mathtt{A}_j] : \mathtt{c}^{\mathtt{I}}[\mathtt{A}, \mathtt{A}_j]\langle l\rangle.h_j$.

We conclude using the second definition of intermediate correspondence with $k = k+1$.

Case $(\mathtt{EIn}_2)$**.** We pose $\mathtt{A}_n = \mathtt{A}$, we have $\Sigma = \Sigma', \mathtt{c}^{\mathtt{I}}[\mathtt{A}_0, \mathtt{A}_n]\langle l\rangle.h$ and $\neg\varphi(\Sigma', \mathtt{c})$. Then $\Delta = \Theta, s[\mathtt{A}_n] : E^{\mathtt{c}_0}[\{\!|T_n|\!\}^{\mathtt{c}} \triangleright \langle \mathtt{A}_0?l\rangle; T']$ and $\Delta' = \Theta, s[\mathtt{A}_{k+1}]E^{\mathtt{c}_0}[\{\!|\,\|T\|\,|\!\}^{\mathtt{c}} \triangleright \langle \mathtt{A}_0?l\rangle; T']$. Without loss of generality we suppose $\Delta; \Sigma$ corresponds to G. We deduce that

- $G = F\{\!|G_0|\!\}^{\mathtt{c}}\langle l \;\mathtt{by}\; \mathtt{A}\rangle; G'_{\_}$,
- $\Delta = s[\mathtt{A}_0] : E^{-}[\{\!|T|\!\}^{\mathtt{c}} \triangleright \langle \mathtt{A}'?l\rangle; \_], \prod_{1\leq i\leq n-1} s[\mathtt{A}_i] : E_i^{-}[\{\!|\,\|T_i\|\,|\!\}^{\mathtt{c}} \triangleright \langle \mathtt{A}'?l\rangle; \_], s[\mathtt{A}_n] : E_n^{-}[\{\!|T_n|\!\}^{\mathtt{c}} \triangleright \langle \mathtt{A}'?l\rangle; \_]$ and
- $\Sigma = \Sigma_0, s[\mathtt{A}_n] : \mathtt{c}^{\mathtt{I}}[\mathtt{A}, \mathtt{A}_n]\langle l\rangle.h$.

From the semantics, we also have

- $\Delta' = s[\mathtt{A}_0] : E^{-}[\{\!|\mathtt{Eend}|\!\}^{\mathtt{c}} \triangleright \langle \mathtt{A}'?l\rangle; \_], \prod_{1\leq i\leq n} s[\mathtt{A}_i] : E_i^{-}[\{\!|\mathtt{Eend}|\!\}^{\mathtt{c}} \triangleright \langle \mathtt{A}'?l\rangle; \_]$ and
- $\Sigma = \Sigma_0, s[\mathtt{A}_n] : h$.

We use the hypothesis and the intermediate correspondence rule to prove that $\Delta'; \Sigma'$ corresponds to
$F\{\!|\mathtt{Eend}|\!\}^{\mathtt{c}}\langle l \;\mathtt{by}\; \mathtt{A}\rangle; G'_{\_}$ which is a derivative of G. We conclude.

Case $(\mathtt{Disc}')$ is easy using the first definition of the intermediate correspondence.

Case $(\mathtt{EDisc}_1)$ is similar to $(\mathtt{EIn}_1)$.

Case $(\mathtt{EDisc}_2)$ is similar to $(\mathtt{EIn}_2)$.

We then prove the following progress property: if Δ, Σ is in intermediate correspondence with $G_1, \ldots, G_n$, and $\Sigma \neq \varepsilon$ then there exist Δ', Σ' with Σ' strictly smaller than Σ. We prove it as follows:

- if Σ contains $\mathtt{c}[\mathtt{A}, \mathtt{A}']\langle l \rangle$, we use the weak projection definitions to prove that Δ contains either $s[\mathtt{A}'] : E^{\mathtt{c}}[\mathtt{A}?\{l_i.T_i\}]$ or $s[\mathtt{A}'] : E^{\mathtt{c}'}[\{\|\|T\|\|\}^{]} \triangleright \langle \mathtt{A}?l \rangle;\_$ with T containing $\mathtt{A}?\{l_i.T_i\}$. We conclude by applying $(\mathtt{In})$ or $(\mathtt{Disc}')$.
- otherwise Σ contains $\mathtt{c}^{\mathtt{I}}[\mathtt{A}, \mathtt{A}']\langle l \rangle$ and we use the intermediate correspondence definition to discuss whether $\mathtt{c}$ is inside an interrupted scope or not and then whether $\varphi(\Sigma_0,)$ or not, the we conclude by applying $(\mathtt{EIn1})$, $(\mathtt{EIn2})$, $(\mathtt{EDisc1})$ or $(\mathtt{EDisc2})$.

By using these properties, we conclude the proof.

B. Implementations from Chapter 6

Here we present the corresponding Python code for some of the example.

Listing B.1: Fibonacci in Session Actors

```
@protocol(c, Fibonacci, P, Ch)
class Fibonacci(Actor):
  def __init__(self, first = False):
    self.ready = false
    self.current = 0
    self.first = first

  @role(c, P)
  def request(self, c, n):
      if n<=2:
        c.P.send.response(1)
      else:
        fib1 = c.subprotocol("Fibonacci",
          P=self, Ch=new Fibonacci())
        fib1.C.send.request(n-1)

        fib2 = c.subprotocol("Fibonacci",
          P=self, Ch=new Fibonacci())
        fib2.C.send.request(n-2)

  @role(c, Ch)
  def response(self, c, n):
    if self.ready:
      if self.is_first:
        print 'The result is', n + self.current
      else:
        c.P.send.response(n + self.current)
    else:
      self.ready = True
      self.current = n
```

We give in Listing B.1 the implementation of the Fibonacci protocol. The `first` argument in the initialisation of a `Fibonacci` actor specifies if this is the initial actor to receive the number to be computed. As prescribed by the protocol, two message handlers, `request` and `response`, are implemented. The latter is the message received by the parent role `P`, while the former is received from the child role `Ch`. Line 13–17 illustrate a new method on the `ActorRole` class for creating subprotocols. Note that the `response` method should be called twice (once for each subprotocol). When the first response message is received the payload is saved temporary in the actor field `self.current` (line 30). When the second response is received, its value is added to `self.current` and the sum is returned to the parent actor (line 27).

Listing B.2: Counter in Session Actors

```
@protocol(cc, Counter, C, P)
class Counter(Actor):
  def __init__(self):
    self.current = 0

  @role(cc)
  def val(self, n):
      self.current+=n

  @role(cc, P)
  def retrieve_message(self, cc):
    cc.P.send.res(self.current)

@protocol(cc, CounterProtocol, C, P)
class Producer(Actor):
  @role(cc, self)
  def join(n, cc):
    for i in range(1, n):
      cc.C.send.val(1)
    cc.C.send.retrieve_message()

  @role(cc, C)
  def res(self, n)
    print 'The result is', n
```

Counter We give in Listing B.2 the implementation of the Counter protocol. We have two Actor classes, `Counter` and `Producer`. The interactions start with the `join` method of the `Producer` class. The body defines consecutive sending of *n* messages (invoking the message handler `val`) of the `Counter` actor (role `C` in the protocol CounterProtocol). In the `val` method the counter actor accumulates the number of received messages as part of its internal state (`self.current`). After the iteration is completed, the `Producer` sends `retrieve_message` to the `Counter`, asking for the result (line 43). The annotation on `retrieve_message` specifies that the receiver of the message is the role instance `cc` and the sender is the role `P` (Producer). In the body of `retrieve_message`, the `Counter` sends to `P` (`c.P.res`) the final result (`self.current`).

Listing B.3: DiningPhilosphers in Session Actors

```
@protocol(c, DiningPhilosophers, Ph, A)
class Philosopher(Actor):
  def __init__(self, no):
    self.no = no

  def join(self, c):
    c.A.send.req(self.no)

  def yes(self):
    self.become(self.eat)

  @role(c, self)
  def eat(self):
    if full:
      c.A.send.done()

  def no(self):
    c.A.send.req(self.no)

@protocol(c, DiningPhilosophers, A, Ph)
class Arbitrator(Actor):

  @role(c, Ph)
  def req(self, c, sender=ph_no)
    if self.free_forks():
      c.Ph[ph_no].send.yes()
    else:
      c.Ph[ph_no].send.no()
  @role(c, Ph)
  def done(self, c):
    print 'The dinner is over'
```

DiningPhilosopher We give in Listing B.3 the implementation of the Dining-Philosophers protocol. For sake of space and clarity, the implementation shows only the interactions for the `Arbiter` and `Philospher`, abstracting from the resource management done by the `Arbiter` (in `free_forks()`). For a `Philospher` the protocol starts from the method `join`. Using its role instance, the actor does the first request to the `Arbiter` to eat (`c.A.req(self.no)`). To distinguish between `Philosphers`, each one is given a number when initialised and when a `Philosopher` sends a message to an `Arbiter` (`c.A.req`) it also sends its number. The `Arbiter` uses this information to identify `Philosopher` that is waiting for the reply(`c.Ph[ph_no]`). Note that in line 18 and in line 20 the `Arbiter` does not know which `Philospher` sends the message (the annotation only specifies that the sender is a role of type `Ph` (`Philosopher`)). When a `Philosopher` receives a `yes` reply it sends a message to itself to eat (`self.become(self.eat())`). If `no` is returned (line 14), the `Philosopher` repeats its request to the `Arbiter` (`c.A.req`).

Listing B.4: CigaretteSmoker in Session Actors

```
@protocol(c, CigaretteSmoker, A, S)
class Arbiter(Actor):
  def __init__(self, n):
    self.n = n

  def join(self, c):
    # sends a message to a random smokers
    radom = generate_random()
    c.S[random].send.start_smoking()

  @role(c, S)
  def started_smoking(self):
    if enough:
      for i in range(1, n):
        c.S[i].send.exit()
    else:
      radom = geenrate_random()
      c.S[random].send.start_smoking()

@protocol(c, CigaretteSmoker, S, A)
class Smoker(Actor):
  @role(c, A)
  def start_smoking(self):
    c.A.send.started_smoking()

  @role(c, A)
  def exit(self):
    print 'I am done.'
```

CigaretteSmoker We give in Listing B.4 the implementation of the CigaretteSmoker protocol. Similar to the DiningPhilospher, the protocol should be initialised by giving the number of the Smokers *n*. The `Arbiter` initiates the first interaction through its `join` method, where it sends a message `start_smoke` to a random smoker (it chooses random smoker from the list of smokers– `S[random]`). The `Smoker` actor has a handler for `start_smoke`. The handler is annotated with a `@role`, specifying this is a handler for the role smoker `c` and the caller for that handler will be the role `A`. Then the `Smoker` notifies the `Arbiter` so the `Arbiter` can proceed and give resources to other `Smokers`. When the `Arbiter` decides to end the smoking it sends `exit` message to everyone.