Abstract

GERARD, SCOTT NEAL. Designing, Verifying, and Evolving Commitment-based Protocols for Business. (Under the direction of Professor Munindar P. Singh.)

Businesses today are facing increasing pressure to interoperate across multiple business partners. We address the design, verification, and evolution of business services in open environments. We approach the problem as an application of multiagent system concepts and techniques using *protocols* and *commitments*. We present an approach to the design and hierarchical composition of protocols that treats protocols as first-class entities, and explicitly addresses role-specific responsibilities and accountabilities. We propose a definition of protocol refinement between a putative subprotocol and a putative superprotocol. To address the evolution of protocol requirements, we describe an interaction architecture and a library of automated, interaction refactorings. We demonstrate these approaches via realistic applications from the insurance, manufacturing, healthcare, and software development domains.

© Copyright 2013 by Scott Neal Gerard

All Rights Reserved

Designing, Verifying, and Evolving
Commitment-based Protocols
for Business

by
Scott Neal Gerard

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2013

APPROVED BY:

Professor Jon Doyle	Professor Tao Xie
Professor Robert Handfield	Professor Munindar P. Singh Chair of Advisory Committee

Dedication

To my wife Nancy, without whose support and assistance, this would not have been possible.

Biography

I was born in Decatur, Illinois in 1956 and grew up in Council Bluffs, Iowa. Throughout my life, my parents, family, teachers, and friends helped shape my love of all things scientific and technological.

I received my Bachelor of Science in Physics and Mathematics in 1974 from Nebraska Wesleyan University in Lincoln, Nebraska, followed by a Masters of Science of Physics in 1978 from the University of Wisconsin in Madison, Wisconsin. After school, I joined IBM where I have worked for nearly 34 years, first in Rochester, Minnesota and now in Raleigh, North Carolina.

When my family moved to Raleigh, I decided to explore computer science more deeply and systematically than I had been able to do on my own. I entered the graduate program in the Department of Computer Science at North Carolina State University, and I have enjoyed (almost) every minute of this unusual, late-in-life endeavor.

My entire professional career, I have intuitively believed that distributed software component architectures, in some fashion, were the future. And while components would clearly be important, I also came to believe they were only half of the solution. The "software wires" that interconnect components were a crucial element, overlooked by most designers. The software wires were much more than hardware wires, and would communicate much more than mere bits of information. I believed they would communicate high-level concepts and even structure the conversation between components. Here, we generalize from components to multiagent systems, and from point-to-point software wires to multi-point protocols, but the basic idea and motivation are the same. I pursue this dissertation because I believe multiagent systems technology is a natural fit for our emerging interconnected business world. In its approach, this dissertation focuses on the "wires" (protocols). I still seek to make this dream a reality.

Acknowledgements

First, I thank Munindar, my advisor. His deep knowledge, insights, intuitions, and exacting standards have significantly improved my research directions, approaches, and ultimate goals. Under his careful direction, I have improved my English, my writing, and my LaTeX skills. He is a great advisor and teacher, and it is has been my privilege to know and work with him.

Second, I thank my loving and supportive wife, Nancy, for proof-reading texts outside her background, for making a number of interesting suggestions, for often listening to me ramble on about obscure topics, and for giving me the time and support to devote to this research. I thank my parents and family for their love, support, and encouragement. These loved ones have given me far more than I could ever list here.

Third, I thank (in no particular order) the excellent NCSU teaching faculty: Professors Dennis Bahler, Jon Doyle, Xiaohui (Helen) Gu, Nagiza Samatova, and William Stewart; members of my research group for their insightful and thought-provoking discussions: Pankaj Telang, Anup Kalia, Kartik Tadanki, Chung-Wei Hang, Nirmit Desai, and Amit Chopra; and others who helped in one way or another: Joe Bigus, Jennifer Bigus, James Carey, Jeffrey Gerard, Michael J. J. Moore, A. Carolyn Neal, and Eileen Townsend.

Finally, I thank Jesus Christ, my Lord, my Savior, my Teacher.

For the Lord gives wisdom; from his mouth come knowledge and understanding. (Proverbs 2:6)

Table of Contents

List of Tables . . . . **ix**

List of Figures . . . . **x**

List of Algorithms . . . . **xi**

Chapter 1 Overview **1**
1.1 Motivation 1
1.2 Problem Statement 2
1.2.1 General Problem 2
1.2.2 Specific Problem 3
1.2.3 Concrete Scenario 3
1.3 General Concepts 4
1.3.1 Protocols 4
1.3.2 Agent Autonomy and Commitments 4
1.3.3 Orchestration and Choreography 4
1.3.4 State Space 5
1.3.5 Verification 5
1.4 Our Approach 6
1.4.1 Commitments 6
1.4.2 Protocols 7
1.4.3 Refinement 8
1.4.4 Requirements Evolution 8
1.4.5 First Class Entities 8
1.4.6 Automated Tooling 10
1.5 Alternative Approaches 10
1.6 Complete Scenario 11
1.7 Who Can Benefit? 12
1.8 Summary of Contributions 12
1.9 Outline 12

Chapter 2 Background **13**
2.1 Running Examples 13
2.2 Commitments 15
2.3 Interpreted Systems 16
2.4 Interceptors 18

Chapter 3 Approach and Overview **19**
3.1 Terminology 19
3.2 Commitments 19
3.2.1 Serial Composition of Commitments 19
3.2.2 Scalar Serial Composition 21
3.2.3 Commitment Covering 21

3.3 Proton . . . 23
3.3.1 Proton Syntax . . . 23
3.3.2 Proton Semantics . . . 25
3.3.3 Proton Specification Examples . . . 26

Chapter 4 Composing Commitment Protocols . . . 28
4.1 Introduction . . . 28
4.1.1 Real-Life Scenario: AGFIL . . . 28
4.1.2 Contributions and Organization . . . 29
4.2 Technical Approach . . . 30
4.2.1 Protocol Composition . . . 30
4.2.2 Role Requirements . . . 31
4.2.3 Enactment Requirements . . . 32
4.2.4 Coupling Commitments . . . 33
4.2.5 Verification . . . 33
4.2.6 Composite Protocol Diagrams . . . 34
4.3 Methodology . . . 35
4.4 Evaluation . . . 36
4.4.1 AGFIL Evaluation . . . 37
4.4.2 Quote To Cash Evaluation . . . 39
4.4.3 ASPE Evaluation . . . 40
4.5 Results and Experience . . . 41
4.6 Discussion . . . 42
4.6.1 Relevant Literature . . . 42
4.6.2 Future Work . . . 45

Chapter 5 Formalizing and Verifying Protocol Refinements . . . 46
5.1 Introduction . . . 46
5.1.1 Proton: Approach and Contributions . . . 47
5.1.2 Contributions . . . 48
5.1.3 Organization . . . 48
5.1.4 Mapping Abstractions across Protocols . . . 49
5.2 Formalizing Protocols and their Refinement . . . 51
5.2.1 Protocol Enactment . . . 51
5.2.2 Protocol Refinement . . . 53
5.3 Verifying Protocol Refinement . . . 55
5.3.1 Intuition: Decomposition . . . 56
5.3.2 Intuition: Diffusion . . . 56
5.3.3 Accommodating Abstraction Mapping . . . 58
5.3.4 Verifying Refinement: Summary . . . 59
5.3.5 Generating CTL Formulas for Verification . . . 60
5.4 Tooling, Detailed Examples, and Experimental Results . . . 61
5.5 Correctness of Our Refinement Verification Method . . . 66
5.6 Discussion . . . 67
5.6.1 Relevant Literature . . . 68

5.6.2 Directions for Future Research ... 69

Chapter 6 Evolving Protocols and Agents in Multiagent Systems ... 70
6.1 Introduction ... 70
6.1.1 Problem: Requirements Evolution ... 71
6.1.2 Approach: Refactoring Interactions ... 71
6.2 Approach Illustrated ... 72
6.2.1 Applying Rule-Based Interceptors ... 72
6.3 Interceptors and Refactorings ... 74
6.3.1 Interceptor Chains ... 74
6.3.2 Interceptor Syntax and Semantics ... 75
6.3.3 Refactorings Formalized ... 76
6.3.4 Protocol Designer Independence ... 76
6.3.5 Agent Designer Independence ... 78
6.3.6 Designer Collaboration ... 78
6.4 Methodology and Application ... 79
6.4.1 Methodology for Protocol Evolution ... 79
6.4.2 Evolve Pay to PayViaCheck ... 79
6.4.3 Guard Propagation ... 82
6.5 Evaluation ... 83
6.6 Discussion ... 84
6.6.1 Comparison to Design Patterns ... 85
6.6.2 Comparison to Agent Designs ... 85
6.6.3 Comparison to Other Work ... 86
6.6.4 Comparison of Mechanistic Capabilities ... 87
6.6.5 Future Directions ... 87

Chapter 7 Case Study ... 89
7.1 Application Selection Desiderata ... 91
7.2 Software Development Protocol Description ... 91
7.3 SWDev Modifications for Positron ... 92
7.4 Design and Composition ... 93
7.5 Requirement Changes ... 96
7.5.1 Implement Requirement Changes by Applying Refactorings ... 96
7.5.2 Refactoring Summary ... 97
7.6 Results ... 97
7.7 Evaluation ... 100
7.7.1 Weaknesses Uncovered ... 100
7.7.2 Strengths Demonstrated ... 101

Chapter 8 Conclusions ... 103
8.1 Claims ... 103
8.2 Summary of Contributions ... 104
8.2.1 Protocol Composition ... 105
8.2.2 Refinement ... 105

8.2.3 Interaction Refactorings 106
8.2.4 Case Study 107
8.3 Future Work 107

References 109

Appendices 115
Appendix A Proton Source Code 116
Appendix B Refinement Theorems 121
Appendix C Rho Refactoring Library 130
C.1 Protocol Designer Independence Refactorings 130
C.2 Agent Designer Independence Refactorings 131
C.3 Designer Collaboration Refactorings 132
Appendix D SWDev Interceptor Chains 133

List of Tables

Table 4.1 Inputs and outputs for each step of the composition methodology. 36
Table 4.2 Positron statistics 42
Table 4.3 Approach comparison 43

Table 5.1 Information about each demonstrated Refinement 65

Table 6.1 Effort in evolving and running sample protocols. 83
Table 6.2 Comparison of Approaches 87

Table 7.1 Positron statistics 98
Table 7.2 Proton statistics 99

List of Figures

Figure 1.1 Process overview. 3
Figure 1.2 One suggestive message sequence diagram for the PayViaCheck protocol. 7
Figure 1.3 Inclusion (part-whole) relationships between first class entities 9

Figure 2.1 Suggestive sequence diagrams for selected protocols 14
Figure 2.2 State transition diagram for commitments 15

Figure 3.1 Proton input syntax in BNF. 24

Figure 4.1 Traditional model of a cross organizational insurance claim processing. 29
Figure 4.2 Selected states and transitions for AGFIL. 34
Figure 4.3 Composite protocol diagram for AGFIL 35
Figure 4.4 CPD for Quote To Cash. 39
Figure 4.5 Composite protocol diagram for ASPE. 41

Figure 5.1 Refinements demonstrated by Proton 48
Figure 5.2 Proton process flow 55
Figure 5.3 A schematic of some possible enactments of the *OrderPayShip* protocol. 56
Figure 5.4 Protocol refinement 59

Figure 6.1 Evolving *ReqResp* to *Order*. 73
Figure 6.2 Detailed enactment of Figure 6.1. 73
Figure 6.3 Interaction architecture 74
Figure 6.4 Evolution of *Pay* to *PayViaCheck* 80
Figure 6.5 Removing unused message (false guard). 82

Figure 7.1 Protocol transformations and verifications 89
Figure 7.2 SWDev as a composite protocol diagram. 94
Figure 7.3 Nested protocol structure for SWDev composite. 94

Figure 8.1 Refinements demonstrated by Proton 106

Figure B.1 The mapping between entities in π_P and π_Q. 121

List of Algorithms

Algorithm 3.1 Using (including) a constituent protocol. . . . 25
Algorithm 3.2 *Pay* Protocol 27
Algorithm 3.3 *PayViaMM* Protocol 27
Algorithm 4.1 Positron source for AGFIL. 38
Algorithm 5.1 Mapping M_1: *Pay* to *PayViaMM* 49
Algorithm 5.2 Alternative Mapping M_2: *Pay* to *PayViaMM* 50
Algorithm 5.3 Alternative Mapping M_3: *Pay* to *PayViaMM* 50
Algorithm 5.4 Nonrefining Mapping B_1: *Pay* to *PayViaMM* 50
Algorithm 5.5 Calculate **refines**(P, M, Q) 62
Algorithm 6.1 *Pay* Protocol 80
Algorithm 6.2 *PayViaCheck* Protocol 81
Algorithm A.1 Pay Protocol 116
Algorithm A.2 PayViaMM Protocol 117
Algorithm A.3 *PayViaCheck* Protocol 118
Algorithm A.4 PayViaCredit Protocol 119
Algorithm A.5 OrderPayShip Protocol 120

Chapter 1

Overview

Businesses interoperation motivates our work.

1.1 Motivation

There are a number of major trends in the modern business environment. Some of those trends pose problems.

Many businesses are focusing on their core competencies. First, some business are small and just getting started, without the time or resources to provide a complete solution on their own. Second, some businesses see the need to interoperate with partners, providing value to their end users that is bigger than the sum of its parts. These businesses are becoming increasingly integrated. Third, some businesses are large monolithic businesses that need to become more agile by downsizing and transferring peripheral, distracting, or low value functions to other organizations. These businesses are becoming more dis-integrated. In all of these cases, the level of integration is not the crucial issue. Rather the commonality is that multiple, autonomous, interdependent businesses and organizations must *interoperate* to form virtual enterprises. The difficulty of arranging inter-business interoperation is hard enough. Businesses must also initiate and terminate interoperations with increasing speed. Grefen and Eshuis [2009] describe the need for *Instant* Virtual Enterprises (IVE).

Supply chain management is just one specific example of the need for organizations to better interoperate, both internally and externally. Monczka et al. [2011, Chapter 4] describe the importance of internal and external integration in supply chain management, and integration is a central element in their two most evolved levels of supply management strategy [2011, p. 231].

Businesses are autonomous and without centralized control. No business will willing relinquish its ability to pursue its own self-interests. Any approach that depends upon a single, centralized planner or controller will never be widely adopted by businesses.

A different major business trend suggests solutions. National economies are becoming ever more

service oriented. According to Spohrer [2007] and Fitzsimmons and Fitzsimmons [2008], over eighty precent of the US economy is service-based and the size of the service economy is expected to grow ever larger. Businesses provide a growing number of on-line versions of their services (e-commerce). The growing number of on-line services can be the building blocks for cross-organizational interoperation.

These major business trends demonstrate the need for modern methodologies and tools so that multiple business partners, from different organizations, can efficiently and effectively interoperate and share each others' business services.

1.2 Problem Statement

We state both a general and a specific version of the problem.

1.2.1 General Problem

This leads us to the general problem:

> General Problem: In modern business environments, how can two or more business partners design and verify *business processes* for interoperation that satisfies each partner's *self-interests*.

Each business partner implements one or more software business agents that interoperate with other agents to enact a *business process*. Each partner's *self-interests* are partner- and protocol-specific.

We approach these important issues within a generic, multiagent systems framework, because multiagent systems support multiple, distinct partners interoperating without central control.

> General Approach: Address the General Problem above using a set of *autonomous, distributed*, software *agents*, in an *open, business* environment, enacting *business services* and interoperating via *business processes*.

Consider the key ideas in this statement. *Agents* are software entities that represent a specific business organization or person. Agents are *autonomous*, making their own decisions as to when and which messages to send; no organization has authority or control over another organization's agents. And there is no centralized controlling entity. Agents can be geographically *distributed*, communicating via messages. The agents exist in an *open* ecosystem, are designed by different organizations, and enter and leave the ecosystem whenever they choose.

We limit our approach to only those aspects of interoperation that can and should be automated. In modern businesses, many exceptional conditions occur that must be handled by human operators. We believe in automating business interoperation only where possible and appropriate.

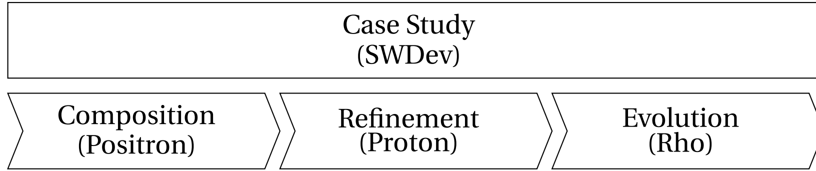


Figure 1.1: Process overview.

1.2.2 Specific Problem

In this dissertation, we address a subset of the broad spectrum of challenges contained in the General Problem. We restate a narrower, more specialized problem, focusing on aspects from the three functional areas shown in Figure 1.1: composition protocol *design*, protocol *verification*, and protocol *evolution*.

> Specific Problem: *Design*, *verify*, and *evolve* demonstrably correct *protocols*, for *business interoperation*, using socially visible *commitments*, as open *multiagent systems*.

We consider a protocol designer plus multiple, self-interested, agent designers, each representing a different organization. Each agent designer is motivated to collaborate with the other agent designers to solve a problem that none of them can solve on their own.

Business protocol and agent designers need to *design* a protocol for their interoperation, *verify* whether the protocol satisfies a set of properties, and *evolve* the protocol as requirements change over time. Rather than depending on agent-internal, mentalistic states (e.g., beliefs, desires, and intentions), we use commitments which are explicit and visible to other agents. We model the partners and their interaction as a multiagent system.

We describe our Positron functionality for protocol composition and single-protocol verification, Proton functionality for protocol refinement verification, and Rho library for interaction refactoring. Finally, we demonstrate our end-to-end approach on a realistic case study (SWDev).

1.2.3 Concrete Scenario

As a specific concrete example, consider the following, realistic example of multipartner interoperation. This software development example (SWDev) is further elaborated in Section 1.6 and is the case study for the end-to-end evaluation in Chapter 7.

> One business partner, CLIENT, conceives of a new software product that he cannot develop and deploy alone. He proposes the idea to a set of potential partners who might wish to collaborate on the project: PSP, a primary software provider; SP, a second software provider; TESTER, a software tester; and EXPERT, a deployer of software on hardware

platforms. These are the design-time business partners. Other partners might include additional, run-time-only, business partners that are not selected until run time.

A number of questions arise. How do the partners go about designing a protocol for their interoperation? Will the protocol enable them to successfully develop the software? Does the protocol satisfy each partner's self-interests (e.g. CLIENT getting functioning software or the other partners getting paid)? Can the partners understand the resulting protocol? How much technical sophistication is required to answer these questions? How much confidence do the partners have in the answers?

1.3 General Concepts

The following general concepts encapsulate the key background for understanding business protocols and facilitating communication.

1.3.1 Protocols

We describe a business process as one or more protocols, and we treat protocols as first class entities. Protocols specify the set of actions each agent can take at each state. In a well-designed protocol, these actions are compatible with correct enactments. On the one hand, a large set of choices gives agents more flexibility than just a single prescribed action at each state. On the other hand, a large set requires more extensive agent decision machinery to choose an action.

1.3.2 Agent Autonomy and Commitments

In our infrastructure, agents act autonomously and unilaterally. Agents cannot arrange to act simultaneously, thereby exposing agents to the risk of other agents not reciprocating. To partially mitigate this risk, agents can make business *commitments* to each other, enabling an agent to choose to act only after another agent has made an appropriate commitment. Commitments a major topic throughout this dissertation.

1.3.3 Orchestration and Choreography

There are two broad approaches to decision making in business processes: orchestration and choreography. In *orchestration*, a central controller directs all the other agents when and how they are to act. The metaphor suggests a conductor directing all the musicians in an orchestra.

In *choreography*, there is no central controller. Each agent chooses its own actions at each step (within limits enabled by the protocol). The metaphor suggests equal partners dancing together, each dancer choosing his or her own actions, within the limits enabled by the choreography.

Orchestration permits simpler agent design than the distributed designs required in choreography, but only the conductor is fully autonomous. In choreography, all agents are fully autonomous.

1.3.4 State Space

Newton's calculus interrelates two viewpoints on the evolution of continuous physical systems. Such a system is roughly similar to a strip of movie film containing multiple frames. The differential, or frame-by-frame, viewpoint describes how any one frame of the system evolves into its successor frame. The integral, or end-to-end, viewpoint describes how the first frame evolves into the last frame. Newton used differential equations to describe frame-to-frame evolution, and then he integrated the differential equations to create the integral viewpoint, stepping directly from the first frame all the way to the last frame.

Modern-day model checkers perform a similar function for discrete systems. Although the details are different, both differential and integral viewpoints exist for discrete state spaces. Instead of a single, linear strip of film, the state space of our business process is a graph of states. The graph branches as it progresses through time. From any given state, a protocol describes the possible state-to-state evolutions, and reflects the differential viewpoint. We use temporal logic formulas to describe properties that are validated over the entire state space, from beginning to end, which is the integral viewpoint.

Both differential and integral viewpoints are important to our work. We use the differential view to define the state-to-state evolution functions as a protocol of guarded messages. But many properties cannot be stated in purely differential terms, for example, commitment satisfaction. To verify commitments, we must use the integral viewpoint provided by temporal formulas. This is important because we describe agents' interests as commitments.

1.3.5 Verification

We use model checking to bridge between differential and integral viewpoints of the problem, and verify integral properties. Temporal logic approaches formally prove whether user specified temporal properties hold in the model's state space.

In our case, we exploit this integral view of temporal logics, but we take a pragmatic approach. Exceptions can and do occur in business. Because agents are autonomous, they can violate their commitments at any time, so the possibility of business exceptions must be handled appropriately. Temporal logics are difficult for most business people and even most programmers to understand, so we construct high-level functions to express these enabled, but undesirable, situations. We do not attempt to fully verify every possible property. We verify a smaller number of temporal formulas, specifically commitment and agent interest checks.

1.4 Our Approach

Multiagent systems are multidesigner, multithreaded, distributed systems in open environments. Designing these types of systems are significantly more difficult than designing single-threaded, single-designer systems. Like all multithreaded systems, designing protocols between multiple, autonomous agents is hard work.

> We reduce protocol design effort and complexity by treating protocols as *first class entities*, employing *choreography*, *modularity*, *composition*, *loose coupling* and *flexible enactment*. Our protocols, described and processed by *Proton*, *Positron*, and a *model checker*, are based on a formal definition of *protocol refinement*, using *commitments* and explicit assignment of *responsibilities* and *accountabilities* to roles.

In an open environment, agents don't generally know their run-time partners at design time, but they must be designed to *some* interface. We propose *protocols* as a conceptual interface that is both clear and easy-to-understand. Protocols are *first class entities* that interconnect multiple abstract (agent) roles and are designed by protocol designers. Software agents are then designed to use a protocol and its abstract roles.

We use *choreography* rather than orchestration, because businesses retain their autonomy when using choreography. Real-world businesses will not give up their autonomy to a central controller, as is required for orchestration.

Modularity and *composition* are standard approaches to reducing design complexity and effort, but they require a clear definition and a means of combining multiple constituent protocols into a composite protocol. To enable wider reuse of protocols, agents are *loosely coupled* to each other, enabling *flexible enactments*, reducing strict sequencing of messages.

Responsibilities and *accountabilities* are explicitly assigned to roles. When we specify protocol properties that must hold, we identify which roles are responsible for achieving them. The debtors are explicitly responsible for satisfying a commitment's consequent. When we compose protocols together, we use coupling commitments to specify accountabilities that each role must ensure between the component protocols. Business partners who are debtors of coupling commitments are accountable for ensuring those commitments are satisfied.

1.4.1 Commitments

Business processes can be written in low-level programming terms like "if proposition A is true in the current state, then set proposition B to false in the next state". Whereas this is simple enough for small protocols, its complexity explodes for large protocols. For large, real world protocols, it requires highly specialized developers and architects, skilled in the necessary mental gymnastics required

when developing programs in general purpose imperative programming languages. Large protocols written in such low-level terms are incomprehensible to most business people.

Although we don't fully eliminate the need for low-level propositions, where ever possible, we prefer protocols designed with commitments, because they naturally describe business situations. *Commitments* are a high-level and formal concept, and they are essential to our approach to protocols. Commitments are written as

$$\mathsf{C}_{\{\text{debtors}\},\{\text{creditors}\}}(\textit{antecedent}, \textit{consequent}),$$

which represents a commitment (or promise or obligation) from a set of debtors, to a set of creditors, that if the antecedent condition becomes true, then the debtors commits to make the consequent condition true, at some point in the future.

Commitments can be formally manipulated as first class entities. The commitment covering relation describes when one commitment is "more general than" another. The serial composition operator constructs a result commitment from two input commitments, capturing the effect of a chain of commitments.

1.4.2 Protocols

Agents require some minimum level of common understanding to interoperate. We assume agents share (1) a common set of terms, or a mapping across their terms, (2) a protocol description written in Proton's source language, (3) a common understanding of the structure and meaning of commitments, and (4) a message communication mechanism.

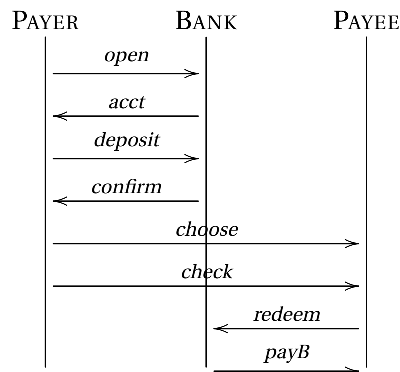


Figure 1.2: One suggestive message sequence diagram for the PayViaCheck protocol.

Protocol PayViaCheck is a small example, where role PAYER makes a payment to role PAYEE via a check. A suggestive message sequence diagram is shown in Figure 1.2, and the corresponding Proton source code is shown in Algorithm 6.2 on page 81. PAYER opens an account with BANK, receiving account information in response. PAYER then makes a deposit and receives a confirmation. PAYER commits to paying and tells PAYEE about its commitment (message choose). At some later point, PAYER sends a check to PAYEE who then redeems the check at BANK and receives payment.

1.4.3 Refinement

Given that protocols are first class entities, one can ask about relationships between protocols. We propose a definition of protocol refinement, where a subprotocol cannot execute any path that is not enabled by its superprotocol. We describe a computational test for refinement using temporal logic, and we prove the two are equivalent.

1.4.4 Requirements Evolution

Given the difficulty of constructing protocols, and given that realistic business processes undergo many requirement changes throughout their life, we must address how protocols can evolve. We propose an interaction architecture, and use interaction *refactorings* to incrementally evolve interactions in response to requirement changes.

1.4.5 First Class Entities

First class entities must have a clear definition and provide a collection of well-defined transformations, single-entity properties, or entity-to-entity relations. Figure 1.3 illustrates the inclusion (part-whole) relationships between the major first class entities we study. Next we describe the primary first class entities, shown in boxes.

Interactions and Interceptors

Interactions and interceptors are supra-protocol elements. Their identification and introduction were not foreseen from the start, but arose out of a need to introduce executable elements that are neither part of the protocol nor part of agent implementations. Others have noted the dichotomy between inter-agent concerns referred to as *protocols* and intra-agent concerns referred to as *policies*. Interceptors emerged as a third set of concerns, between protocols and policies, that are captured by our interaction architecture. Interceptors are always associated with exactly one agent, but exist outside the agent's implementation, in the middleware. Treating interactions and interceptors as first class entities involves:

- Interaction definition and architecture (Section 6.3.1).

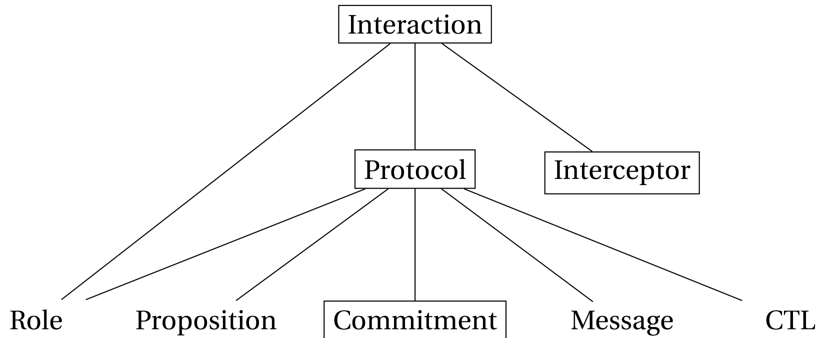


Figure 1.3: Inclusion (part-whole) relationships between first class entities. Boxed concepts received our primary focus.

- Interceptor syntax and semantics (Section 6.3.2).
- Definition of interaction refactoring (Section 6.3.3).
- Three types of interaction refactorings: *Protocol Designer Independence*, *Agent Designer Independence*, and *Designer Collaboration* (Section 6.3.3).
- A library of interaction refactorings (Section 6.3.3 and Appendix C).

Protocols

Protocols are the primary, first class entities we study. A full definition of a protocol also requires the important first class entities of roles, propositions, commitments, and messages. We use computational temporal formulas written in CTL as the language for verifying properties of protocol state spaces. Treatment of protocols as first class entities involves:

- Protocol definition as a set of roles, propositions, commitments, guarded messages, used constituent protocols, and verification conditions (Section 3.3).
- Protocol syntax (Section 3.3.1).
- Protocol semantics (Section 3.3.2).
- Expansion of a constituent protocol inside a composite protocol (Section 4.2.1).
- Refinement relation between a putative superprotocol and a putative subprotocol (Chapter 5).

Commitments

Commitments are another significant first class entity we study, particularly in the context of commitment-based protocols. Treatment of commitments as first class entities involves:

- Commitment definition (Section 2.2).
- Serial composition of two commitments (Section 3.2.1).
- Scalar serial composition of an expression and a commitment (Section 3.2.2).

- Commitment covering relation between two commitments (Section 3.2.3).

1.4.6 Automated Tooling

Throughout the dissertation, we rely upon automated tooling we implemented for this research.

The Proton program contains two distinct functionalities: *Proton* and *Positron*. The two names refer to distinct functionalities of the single program, developed in distinct phases to address distinct issues. Both read the same declarative source language statements for protocols (differential viewpoint), perform various protocol transformations, and construct models of the protocol plus temporal logic checks written in computational tree logic (CTL). The models and temporal logic formulas are read by the MCMAS model checking tool [Lomuscio et al., 2009], which performs the detailed verification (integral viewpoint). *Proton* creates MCMAS models to verify protocol refinement between two protocols. *Positron* creates MCMAS models to verify role and enactment properties of a single protocol.

The *Rho* refactoring library is a collection of callable refactoring routines. Each refactoring transforms one interaction into another interaction.

We name our tools after elementary particles in Physics, but little significance should be given to the names. We show similarities between the names and the concepts as capitalized and italicized characters: *PROTO*n for *PROTO*col, *POSIT*ron for com*POSIT*e, and *R*h*O* for *R*efact*O*ring.

The Rho elementary particle is a meson. A meson is composed of a quark and an antiquark, and mesons transmit the nuclear force that holds atomic nuclei together. A parallel mental image for refactorings is: a refactoring is typically composed of (one or more instances of) a send interceptor and a receive (“anti-send”) interceptor, and a refactoring bridges between (holds together) two adjacent interactions.

1.5 Alternative Approaches

Businesses can choose hand-crafted software for their interoperation. But, hand-crafted approaches typically (1) cannot be designed or reviewed by business users, because their only definition is their implementation in a programming language, (2) have long development cycles, (3) are multithreaded applications that are difficult to test and debug, and (4) are brittle to modify. Because of these serious difficulties, the hand-crafted approach does not scale to a wide population of users.

Business Process Model and Notation [OMG, 2011] is an industry standard, business-friendly notation for business processes. The industry has invested a lot of time and effort developing BPMN assets. However, BPMN emphasizes a step-by-step approach which overly limits agents’ ability to choose between alternative enactments, based on a particular situation. Furthermore, BPMN does not address verification.

The π-calculus is an advanced process calculus for describing processes. However, it requires specialized skills that business personnel lack, making it unsuitable for business process development. Business personnel must be able to read and understand their business processes. It is also an operational approach and omits the business meaning of the interactions.

1.6 Complete Scenario

The following, realistic example illustrates the end-to-end interoperation of the multiple elements of our approach. It is subject of our case study in Chapter 7.

> One business partner, CLIENT, conceives of a new software product that it cannot develop and deploy alone. It proposes the idea to a set of potential partners that might want to collaborate on the project: PSP, a primary software provider; SP, a second software provider; TESTER, a software tester; and EXPERT, a deployer of software on hardware platforms. These are the design-time business partners. Other partners might include additional, run-time-only, business partners that are not selected until run time.
>
> Business development personnel and technical protocol designers from each design-time business partner meet in a design session to design a mutually agreeable protocol for their interoperation. The business development personnel focus on business aspects, while the technical protocol designers focus on translating business requirements into protocols. Each partner wants to know the protocol satisfies its interests.

If any run-time-only partners exist (e.g., customers), the design-time partners must also clearly identify and describe their self-interests. We propose a methodology and business-friendly diagram to facilitate this design session (Chapter 4).

A natural question is how two similar protocols relate to each other. We propose a definition for *protocol refinement* where a subprotocol refines a superprotocol (Chapter 5).

After the protocol is designed, each business partner implements his own software agent. At each state, the agent chooses between the alternatives open to it by the protocol. Then all the partners deploy their agents into operation.

Over time, as the partners become more familiar and comfortable with each others' wants and needs, and as the business environment and requirements change, they may want to evolve their protocol. A business partner can modify its agent's operation at any time—unilaterally in some cases, jointly in other cases—using the refactorings described in Chapter 6.

1.7 Who Can Benefit?

This dissertation can be beneficial to many audiences: (1) start-up organizations that want to leverage existing partners to minimize their startup costs, to minimize their learning curve, or to exploit their partner's high quality services; (2) organizations that want to restructure; (3) organizations undergoing mergers and acquisitions; and (4) organizations that want to "rightsize" or refocus on their core competencies.

1.8 Summary of Contributions

We now summarize the main contributions of this dissertation here. Section 8.2 revisits these contributions in detail.

Define the protocol refinement relation between a putative superprotocol and a putative subprotocol, and implemented it using CTL formulas. Extend commitments to enable sets of debtors and creditors. Define two forms of serial composition of commitments and the commitment covering relation between two commitments.

Define protocol composition, based on role specific concepts and high-level verification functions. Describe *composite protocol diagrams* as a graphical notation, which conveys important features of the composite protocol to business and technical stakeholders. Implement a decision procedure and mechanical verification of protocols.

Define an architecture for evolving requirements, using *refactorings*. We identify three types of interaction refactorings, and a library of automated refactorings.

1.9 Outline

Chapter 2 describes preexisting work we rely upon. Chapter 3 lays out our framework for using protocols for business processes and other necessary machinery. Chapter 4 describes composition of protocols and our methodology for constructing them. Chapter 5 defines the relation of protocol refinement and proves results about it. Chapter 6 describes a set of refactorings for protocol requirements evolution. Chapter 7 applies our entire approach to a realistic software development case study. Chapter 8 covers our primary claims, states our final results, and discusses future directions.

Chapter 2

Background

Our approach builds upon the examples and concepts in this chapter.

2.1 Running Examples

We introduce a number of running examples of protocols in Figure 2.1 which are used throughout this dissertation. The diagrams are suggestive message sequence diagrams; other message sequences are possible. We understand a protocol semantically in terms of exactly the set of runs (i.e., computations) it allows.

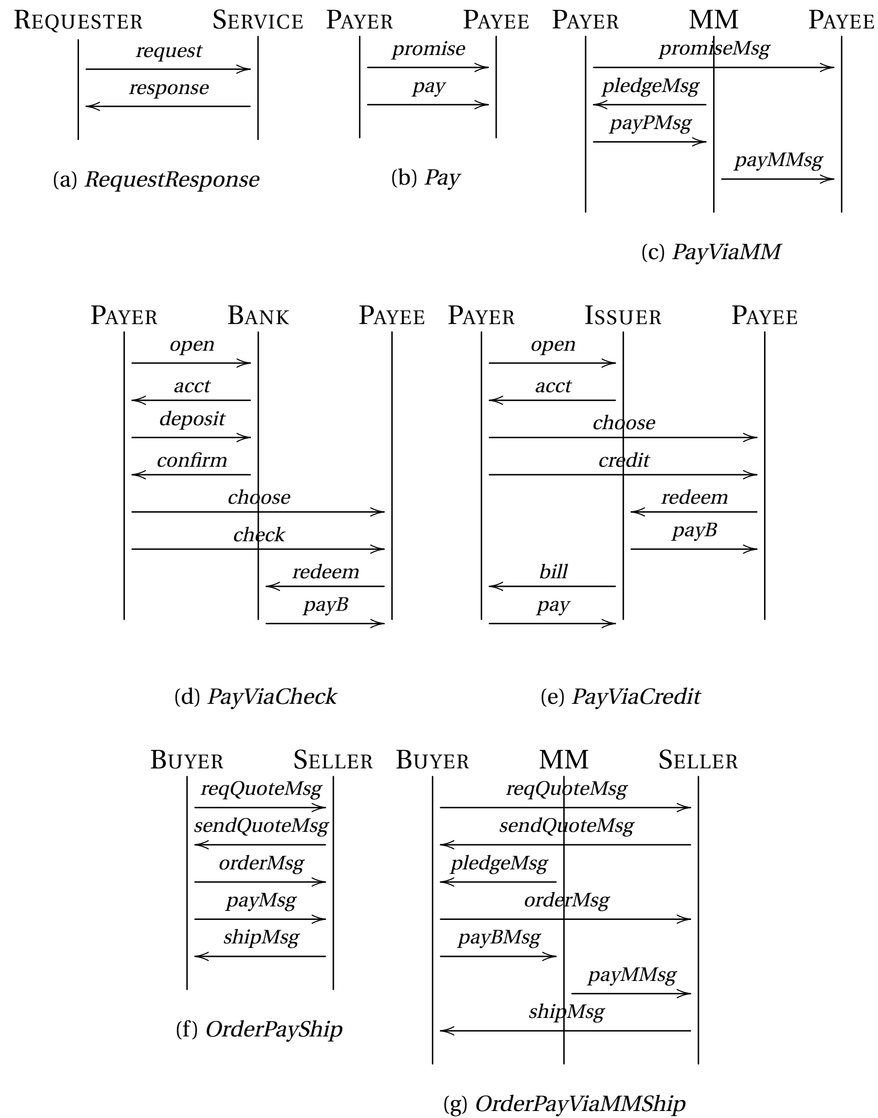
1. Protocol *RequestResponse*: A simple and common protocol. REQUESTER makes a request to SERVICE, who then responds.

2. Protocol *Pay*: Our simplest payment protocol. If PAYER chooses to do so, it may promise to pay PAYEE, creating a commitment. Once committed, PAYER pays at some future point.

3. Protocol PayViaMM: Similar to *Pay*, except PAYER first pays a middleman (MM), who in turn pays PAYEE. Both *Pay* (directly) and *PayViaMM* (indirectly) send a payment from PAYER to PAYEE.

4. Protocol *PayViaCheck*: Similar to *PayViaMM*, except PAYER must first open an account with BANK. At any time, PAYER can make confirmed deposits and can send PAYEE a check that it redeems for payment at BANK.

5. Protocol *PayViaCredit*: Similar to *PayViaCheck*, except PAYER must first open an account with a credit card ISSUER. At any time, PAYER can make confirmed deposits or can send PAYEE a credit that it may redeem for payment at ISSUER. ISSUER periodically sends bills to PAYER, who then pays.



(a) *RequestResponse*

(b) *Pay*

(c) *PayViaMM*

(d) *PayViaCheck*

(e) *PayViaCredit*

(f) *OrderPayShip*

(g) *OrderPayViaMMShip*

Figure 2.1: Suggestive sequence diagrams for selected protocols (in general, alternative sequences may occur).

6. Protocol *OrderPayShip*: BUYER requests a price quote for a good from SELLER. SELLER returns the price quote along with its implicit commitment to ship that good if BUYER pays for it. At this point, the BUYER may place an order for the good, along with its implicit commitment to pay for the good if SELLER ships it. Then, in either order, BUYER pays and SELLER ships.

7. Protocol *OrderPayViaMMShip*: Similar to *OrderPayShip*, except payments are made using *PayViaMM* rather than *Pay*.

2.2 Commitments

Commitments are a formal and concise method of describing how agent roles commit to performing future actions [Singh, 1999; Yolum and Singh, 2002]. Commitments state the debtor *should* make the consequent true whenever the antecedent is true, but agents are not *required* to do so. Debtors can break their commitments. Further, commitments do not require immediate fulfillment by debtors.

We extend previous commitment definitions in two ways. First, we allow both debtors and creditors to be sets of roles, enabling us to handle commitment chains with multiple debtors and multiple creditors. Second, we introduce a new TRANSFER commitment operation to unify and replace prior uses of DELEGATE and ASSIGN.

A commitment $\mathsf{C}_{\{\text{debtors}\},\{\text{creditors}\}}(\textit{antecedent}, \textit{consequent})$ means that the debtors commit to the creditors, that if the antecedent holds, they will bring about the consequent. In an active commitment whose antecedent is false, the debtors are conditionally committed to the creditors. When the antecedent becomes true, we say the commitment is *detached*, and the debtors become unconditionally committed to the creditors.

Consider the following example commitments drawn from the running examples. In Equation 2.1 for *RequestResponse*, SERVICE commits to REQUESTER to respond to requests. In Equation 2.2 for *Pay*, the PAYER conditionally commits to paying PAYEE. In Equation 2.3 for *PayViaMM*, PAYER and MM commit to paying PAYEE via *payP* (PAYER's payment) and *payM* (MM's payment).

$$\mathsf{C}_{\text{Service},\text{Requester}}(\textit{request}, \textit{response}) \tag{2.1}$$

$$\mathsf{C}_{\text{Payer},\text{Payee}}(\textit{promise}, \textit{pay}) \tag{2.2}$$

$$\mathsf{C}_{\{\text{Payer},\text{MM}\},\text{Payee}}(\textit{promise}, \textit{payP} \wedge \textit{payM}) \tag{2.3}$$

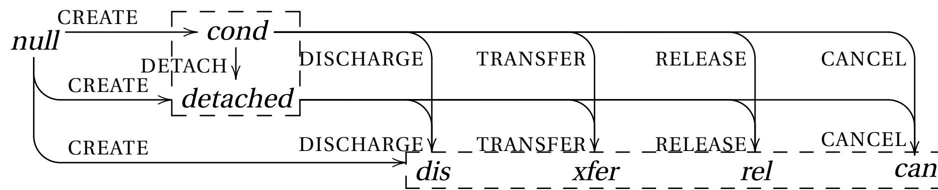


Figure 2.2: State transition diagram for commitments (states in lowercase; transitions in small caps).

Figure 2.2 shows the state transition diagram for a commitment. The states of a commitment are (i) *null*: where the commitment does not yet exist; (ii) *cond* (active and conditional): after CREATE with antecedent and consequent being false, and with no other operations; (iii) *detached* (active

and detached): after CREATE with antecedent true, consequent false, and with no other operations; (iv) *dis* (discharged): after CREATE and consequent being true; (v) *xfer* (transferred): after CREATE and TRANSFER; (vi) *rel* (released): after CREATE and RELEASE; and (vii) *can* (canceled): after CREATE and CANCEL. A commitment in states *cond* or *detached* is said to be *active*. A commitment in states *dis*, *xfer*, *rel*, or *can* is said to be *resolved*.

The commitment operations are (i) CREATE, performed only by debtors, creates an active commitment; (ii) DETACH, which occurs implicitly when the antecedent becomes true; (iii) DISCHARGE, which occurs implicitly when the consequent becomes true; (iv) TRANSFER, performed by either debtors or creditors, deactivates the current commitment and replaces it by another commitment; (v) RELEASE, performed only by creditors, deactivates the commitment, thus releasing the debtors; and (vi) CANCEL, performed only by debtors, which cancels, deactivates, and "breaks" the debtors' commitment.

As in previous work, in a correct enactment of a protocol, each detached commitment must eventually resolve. Debtors may act before they are required to do so and the consequent may become true before the antecedent. In general, there is no guarantee that autonomous debtors do not arbitrarily CANCEL. In practice, the creditors would assume the debtors are trustworthy or the setting would include an external mechanism (such as penalties) to ensure the debtors' compliance.

2.3 Interpreted Systems

We adopt Lomuscio and colleagues' [2006; 2009] formalization of a multiagent system as an *interpreted system*. Importantly, protocols involve roles, not agents. We presume no knowledge of the internals of an agent playing a role and consider all possible strategies that a role may follow in a protocol.

Each role is described by a set of possible local states, a set of local actions, a local strategy listing the legal actions in each local state, and a local progression function defining the progression of the role's local state based on the actions performed by all the roles. To clarify the terminology, our *role* and *strategy* respectively map to *agent* and *protocol* in work by Lomuscio and colleagues.

Definition 2.3.1 (Interpreted System)**.** *An interpreted system* $\mathscr{I}$ *is*

$$\mathscr{I} \quad = \quad \langle \Sigma, P, PV, L^i, Act^i, AP^i, t^i, G, G_0, F \rangle$$

$\Sigma = \{1, \ldots, n, e\}$ *is a set of three or more roles, including a distinguished role* e *that stands for the environment.* P *is a set of atomic propositions. Let* $i \in \Sigma$ *range over all roles and the environment* e*.* L^i *is a nonempty set of possible local states for each* i*.* Act^i *is a set of actions for each* i*.* $AP^i : L^i \times L^e \mapsto 2^{Act^i}$ *is the local strategy for each* i*. In local state* $l \in L^i$*,* $l^e \in L^e$*, role* i *can perform only the actions in* $AP^i(l, l^e)$*.* $G \subseteq L^1 \times \cdots \times L^n \times L^e$ *is the set of reachable global states. For any global state* $g \in G$*, we write* g^i *for the* i*-th component in* g*, i.e., the local state of role* i *in* g*.* $G_0 \subseteq G$ *is a nonempty set of initial global states.* $PV : P \mapsto 2^G$ *is the evaluation function for propositions. The set of joint actions is*

$Act = Act^1 \times \cdots \times Act^n \times Act^e$. $t^i : L^i \times L^e \times Act \mapsto L^i$ is the local progression function for role $i \in \Sigma \setminus e$, and $t^e : L^e \times Act \mapsto L^e$ is the progression function for the environment. All roles progress simultaneously. The global progression function is $T : G \times Act \mapsto G$ and is defined such that $T(g, a) = g'$ iff $\forall i \in \Sigma \setminus e$: $t^i(g^i, g^e, a^i) = g'^i$ and $t^e(g^e, a^e) = g'^e$. T must be serial ($\forall g \in G, \exists a \in Act, \exists g' \in G : T(g, a) = g'$). F is a set of Boolean fairness conditions, each of which must be true infinitely often on all legal execution paths. A path π in $\mathscr{I}$ is an infinite sequence of global states $\langle g_0, g_1, \ldots \rangle$ in G such that every pair of adjacent states is a legal transition, i.e., $\forall i \geq 0 : \exists a \in Act : T(g_i, a) = g_{i+1}$. The i-th state in path π is denoted π_i, and the set of all paths starting at $g \in G$ is denoted $\Pi(g)$.

Given an interpreted system $\mathscr{I}$, we associate with it a Kripke model $\mathscr{K} = (G, G_0, T, PV)$ where G is the set of possible worlds understood as the reachable states of $\mathscr{K}$, built from the set of initial states G_0 by iterating the global progression function T; and, the temporal relation $T \subseteq G \times Act \times G$ connects global states based on the joint actions. The labeling function PV is the propositional labeling function.

The grammar of the temporal language CTL is (*PropName* is an atomic proposition)

$$\phi \quad ::= \quad PropName \mid \neg\phi \mid \phi \vee \phi \mid \phi \wedge \phi \mid \mathbf{AG}\phi \mid \mathbf{AF}\phi \mid \mathbf{A}[\phi \mathbf{U} \phi] \mid \mathbf{EG}\phi \mid \mathbf{EF}\phi \mid \mathbf{E}[\phi \mathbf{U} \phi]$$

Our temporal formulae use the standard CTL temporal logic operators: **A** (for all paths), **E** (for some path), **G** (on all future states on a path), and **F** (eventually on the path). The following is the semantics for CTL, specified relative to the Kripke structure $\mathscr{K}$ at state g. Given a model $\mathscr{K}$ for an interpreted system, $\mathscr{I}$ satisfies a CTL formula ϕ if and only if $\mathscr{K}, g_0 \models \phi$, where $g_0 \in G_0$ is a starting state.

$\mathscr{K}, g \models p$	iff	$p \in g$
$\mathscr{K}, g \models \neg\phi$	iff	it is not the case $\mathscr{K}, g \models \phi$
$\mathscr{K}, g \models \phi \wedge \psi$	iff	$\mathscr{K}, g \models \phi$ and $\mathscr{K}, g \models \psi$
$\mathscr{K}, g \models \phi \vee \psi$	iff	$\mathscr{K}, g \models \phi$ or $\mathscr{K}, g \models \psi$
$\mathscr{K}, g \models \mathbf{AG}\phi$	iff	$\forall \pi \in \Pi(g), \forall i \geq 0, \mathscr{K}, \pi_i \models \phi$
$\mathscr{K}, g \models \mathbf{AF}\phi$	iff	$\forall \pi \in \Pi(g), \exists i \geq 0, \mathscr{K}, \pi_i \models \phi$
$\mathscr{K}, g \models \mathbf{A}[\phi \mathbf{U} \psi]$	iff	$\forall \pi \in \Pi(g), \exists k \geq 0, \mathscr{K}, \pi_k \models \psi$ and $\forall 0 \leq j < k, \mathscr{K}, \pi_j \models \phi$
$\mathscr{K}, g \models \mathbf{EG}\phi$	iff	$\exists \pi \in \Pi(g), \forall i \geq 0, \mathscr{K}, \pi_i \models \phi$
$\mathscr{K}, g \models \mathbf{EF}\phi$	iff	$\exists \pi \in \Pi(g), \exists i \geq 0, \mathscr{K}, \pi_i \models \phi$
$\mathscr{K}, g \models \mathbf{E}[\phi \mathbf{U} \psi]$	iff	$\exists \pi \in \Pi(g), \exists k \geq 0, \mathscr{K}, \pi_k \models \psi$ and $\forall 0 \leq j < k, \mathscr{K}, \pi_j \models \phi$

We write $a \models b$ if and only if for all models $\mathscr{K}$ and states g, $\mathscr{K}, g \models b$ holds whenever $\mathscr{K}, g \models a$ holds.

2.4 Interceptors

Our approach builds on the time-honored architectural abstraction of an *interceptor* [Vinoski, 2002], with servlet filter chains being one well-known example [Sun, 2009]. Extending this, we show how to construct interceptors modularly in a rule-based manner from logical specifications of refactorings. Each interceptor is expressed as a series of reaction rules that are potentially triggered by a message and which may refer to the interceptor's internal state. As traditionally, an interceptor mediates all message flow involving an agent. An *interceptor chain* is an ordered list of one or more interceptors. Incoming messages pass through an interceptor chain before arriving at the business logic component of an agent and outgoing messages likewise pass through the same interceptor chain in reverse order.

Chapter 3

Approach and Overview

In this chapter, we introduce a number of our contributions that are used through out, including new commitment operations, Proton's syntax and semantics, and the Proton source code for two example protocols.

3.1 Terminology

We adopt the following terminology. A *subprotocol* refines a *superprotocol*. In hyphenated form, *super-x* and *sub-x* refer to element x as it occurs in the superprotocol and subprotocol, respectively. For example, a *super-role* is a role defined in the superprotocol and a *sub-commitment* is a commitment defined in the subprotocol.

3.2 Commitments

Commitments are crucial element of our approach.

3.2.1 Serial Composition of Commitments

In *PayViaMM*, where the payer commits to a middleman who commits to the payee, the two commitments together effectively commit the payer to the payee. We introduce serial composition as a general way to chain commitments over intermediaries, computing a single, resultant commitment. The serial composition of commitments is a static construct, but the resultant commitment dynamically progresses through the states in Figure 2.2 as the protocol progresses.

Definition 3.2.1. *Let* C_A *and* C_B *be two commitments that satisfy the well-defineness condition* $\mathsf{C}_A.csq \models \mathsf{C}_B.ant$. *Then, the serial composition of* C_A *and* C_B *is the commitment* $\mathsf{C}_\oplus = \mathsf{C}_A \oplus \mathsf{C}_B$ *whose*

components are specified precisely as follows:

$$\mathsf{C}_{\oplus}.debt \;:=\; \mathsf{C}_A.debt \cup \mathsf{C}_B.debt \tag{3.1}$$

$$\mathsf{C}_{\oplus}.cred \;:=\; \mathsf{C}_A.cred \cup \mathsf{C}_B.cred \tag{3.2}$$

$$\mathsf{C}_{\oplus}.ant \;:=\; \mathsf{C}_A.ant \tag{3.3}$$

$$\mathsf{C}_{\oplus}.csq \;:=\; \mathsf{C}_A.csq \wedge \mathsf{C}_B.ant \wedge \mathsf{C}_B.csq \tag{3.4}$$

The state of $\mathsf{C}_{\oplus}$ *is defined based on the states of* C_A *and* C_B. $\mathsf{C}_{\oplus}$ *is created exactly when both* C_A *and* C_B *are created.* $\mathsf{C}_{\oplus}$ *is respectively transferred, released, or canceled when at least one of* C_A *and* C_B *is transferred, released, or canceled.*

Singh's [2008] formalization of commitments includes the similar idea of commitment chaining, but serial composition additionally captures the intuition of a coalition of roles. The above well-defineness condition $\mathsf{C}_A.csq \models \mathsf{C}_B.ant$ follows Singh's [2008] definition for chaining, although serial composition accommodates different roles. The second commitment becomes active ($\mathsf{C}_B.ant$) whenever the first commitment resolves ($\mathsf{C}_A.csq$), including the case where the first debtors perform without being required to do so ($\mathsf{C}_A.ant$ always false).

Informally, we say debtors are *responsible* for their commitments, and creditors are *beneficiaries* of their commitments. In a detached commitment, the debtors are responsible for eventually making the consequent true. Responsibility can be *several* (each debtor is responsible for just its portion), *joint* (each debtor is individually responsible for the entire commitment), or *joint and several* (the creditors hold one debtor fully responsible, who then pursues other debtors). We use *several* responsibility so that, in serial composition of commitments, a debtor is never compelled to assume additional responsibilities. The result of serial composition is useful for reasoning about multiple commitments, but the original commitment expression, with its individual commitments, must be retained to determine which role(s) failed to perform if the resultant consequent is not produced.

$\mathsf{C}_{\oplus}$ states the union of debtors is committed to the union of creditors to bring about the consequent $\mathsf{C}_A.csq \wedge \mathsf{C}_B.ant \wedge \mathsf{C}_B.csq$ when antecedent $\mathsf{C}_A.ant$ is true. Debtors are *severally* responsible for $\mathsf{C}_{\oplus}$, so that debtors are never compelled to assume additional responsibilities. Every debtor in $\mathsf{C}_A.debt$ is partially responsible for discharging C_A, and thus is partially responsible for discharging $\mathsf{C}_{\oplus}$. Also, every debtor in $\mathsf{C}_B.debt$ has some responsibility for $\mathsf{C}_{\oplus}$. Equation 3.1 captures this intuition for debtors. Equation 3.2 captures the analogous intuition for creditors.

If we order the states of a commitment as $null < can < rel < xfer < cond < detached < dis$, then the state of a serial composition is the minimum of its constituents' states: $\mathsf{C}_{\oplus}.state = min(\mathsf{C}_A.state, \mathsf{C}_B.state)$. That is, a serial composition progresses no further than its least constituent.

Because of Equations 3.3 and 3.4, serial composition is neither commutative nor associative. However, it creates commitments that are at least as strong as, and typically stronger than, their inputs.

$\mathsf{C}_A \oplus \mathsf{C}_B$ is typically stronger than C_A because, even though both have the same antecedent ($\mathsf{C}_A.ant$), in general, $\mathsf{C}_A \oplus \mathsf{C}_B$ has a stronger consequent ($\mathsf{C}_B.csq$ vs. $\mathsf{C}_A.csq \wedge \mathsf{C}_B.ant \wedge \mathsf{C}_B.csq$). The lemma below shows serial composition is not always stronger, because $\oplus$ is idempotent: a commitment can be usefully added to a commitment chain only once.

Lemma 3.2.2. *If C_k is any commitment in a commitment chain $\bigoplus_{1\le i\le n} \mathsf{C}_i$, then composing C_k again does not increase the strength.*

$$(\bigoplus_{1\le i\le n} \mathsf{C}_i) \oplus \mathsf{C}_k \quad = \quad \bigoplus_{1\le i\le n} \mathsf{C}_i$$

Proof. If $\bigoplus_i \mathsf{C}_i$ is well defined, then so is $(\bigoplus_i \mathsf{C}_i) \oplus \mathsf{C}_k$. By inspection, Equations (3.1–3.4) yield the same results for both sides. □

3.2.2 Scalar Serial Composition

Definition 3.2.3 (Scalar Serial Composition)**.** *Let S_A be a set of Boolean expressions (terms) and C_B be a commitment that satisfy the well-defineness condition $S_A \models \mathsf{C}_B.ant$. Then, the scalar serial composition of S_A and C_B is the commitment $\mathsf{C}_\oplus = S_A \oplus \mathsf{C}_B$ whose components are specified precisely as follows:*

$$\mathsf{C}_\oplus.debt \quad := \quad \mathsf{C}_B.debt \tag{3.5}$$

$$\mathsf{C}_\oplus.cred \quad := \quad \mathsf{C}_B.cred \tag{3.6}$$

$$\mathsf{C}_\oplus.ant \quad := \quad S_A \wedge \mathsf{C}_B.ant \tag{3.7}$$

$$\mathsf{C}_\oplus.csq \quad := \quad \mathsf{C}_B.csq \tag{3.8}$$

If all terms in S_A are true at some state of an interaction, and C_B has been created, then $\mathsf{C}_B.debt$ unconditionally commits to making $\mathsf{C}_B.csq$ true at some point in the future.

Scalar serial composition simply adds all the terms in S_A to C_B's antecedent. It enables us to efficiently encode statements of the form "if S_A is true, then $\mathsf{C}_B.csq$", and represent it as a commitment, which can then be serially composed with other commitments. It enables a set of propositions to be added to the front of a commitment chain.

3.2.3 Commitment Covering

Because commitments are crucial to our semantics of protocols, commitments are also crucial to refinement. And because we need to compare two protocols, we need a mechanism to compare two commitments. Specifically, each super-commitment must be *covered by*, or make at least the same commitment as, another relevant sub-commitment. The commitment comparison accommodates a mapping to account for the commitments being expressed at different levels of abstraction. Def-

inition 3.2.4 extends Chopra and Singh's [2009] notion of *commitment strength*. In addition to the logical relationships between antecedents and consequents, this definition incorporates the mapping of roles and propositions.

Definition 3.2.4 (Commitment Covering)**.** *A stronger commitment* C_S *covers (is stronger than) a weaker commitment* C_W *with respect to mapping* M*, written* $\mathsf{C}_W \leq_M \mathsf{C}_S$*, if and only if*

$$\forall d \in \mathsf{C}_W.debt \quad M(d) \cap \mathsf{C}_S.debt \neq \emptyset \tag{3.9}$$

$$\forall c \in \mathsf{C}_W.cred \quad M(c) \cap \mathsf{C}_S.cred \neq \emptyset \tag{3.10}$$

$$M(\mathsf{C}_W.ant) \models \mathsf{C}_S.ant \tag{3.11}$$

$$\mathsf{C}_S.csq \models M(\mathsf{C}_W.csq) \tag{3.12}$$

where $M(x)$ *maps (super-) element* x *in* C_W *to an expression of (sub-) elements in* C_S*.*

Every super-debtor is partially (severally) responsible for discharging the super-commitment. Each super-role is mapped to (implemented by) a set of sub-roles $M(d)$. We require each super-debtor's responsibilities be passed to one or more of its sub-debtors. Together these sub-debtors assume the super-debtor's responsibilities. Equation 3.9 captures the requirement that every super-debtor's responsibilities must pass to at least one of its sub-debtors, so that responsibilities are not lost. Similarly, each super-creditor is a partial beneficiary of the super-commitment. Equation 3.10 captures the requirement that every super-creditor's benefit pass to at least one of its sub-creditors.

In many situations, multiple sub-commitments must be combined to cover a single super-commitment. In those cases, a super-commitment is covered by the serial composition of multiple sub-commitments.

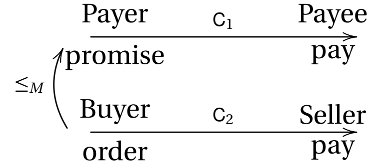
We visualize our explanations by diagramming a commitment as a labeled arrow.

$$\xrightarrow[\text{ant}^{(\wedge)} \qquad\qquad \text{csq}^{(\wedge)}]{\text{debt}^{(\cup)} \ \mathsf{C}\text{name}^{(\oplus)} \ \text{cred}^{(\cup)}}$$

The name of the commitment is written in the top center of the arrow. Debtors and creditors are above the arrow and the antecedent and consequent below it. When multiple terms appear in a position, they are implicitly combined using the operator in parentheses.

As a simple example, consider commitments $\mathsf{C}_1 = \mathsf{C}_{\text{Payer,Payee}}(\textit{promise}, \textit{pay})$, $\mathsf{C}_2 = \mathsf{C}_{\text{Buyer,Seller}}(\textit{order}, \textit{pay})$, and an abstraction mapping M that is defined on roles and propositions as follows: (i) $\textit{Payer} \mapsto \{\textit{Buyer}\}$,

(ii) $Payee \mapsto \{Seller\}$, (iii) $promise \mapsto order$, and (iv) $pay \mapsto pay$. The diagram shows C_2 covers C_1.



As another example, we obtain $\mathsf{C}(order, ship) \leq \mathsf{C}(order \vee freeCoupon, ship)$ by Equation 3.11, since the stronger commitment detaches when *order* or *freeCoupon* is true. And, likewise $\mathsf{C}(order, ship) \leq \mathsf{C}(order, ship \wedge expressDelivery)$ holds by Equation 3.12, since to discharge the stronger commitment requires *expressDelivery* in addition to *ship*.

3.3 Proton

This section covers the syntax and semantics of Proton and Positron source statements. It also contains source examples.

3.3.1 Proton Syntax

We describe protocols using Gerard and Singh's [2013] Proton syntax in Figure 3.1, where | separates alternatives, $\langle A\rangle^*$ is zero or more repetitions of A, $\langle A\rangle^+$ is one or more repetitions of A, and $\langle A\rangle^?$ is an optional occurrence of A.

The *Protocol* nonterminal describes the syntax for a protocol (as Listing 3.2 on page 27 exemplifies). A protocol declares roles, propositions, commitments, messages, and usages of component protocols.

Protocols contains sets of guarded statements of the form:

$$sender \rightarrow receiver : [guard]\ message\ \mathsf{means}\{actions\}$$

For example, PAYER → PAYEE : [*promise*] *pay* means {SET *(pay)*}. Each message m is sent from a sender ($m.snd$) to a receiver ($m.rcv$), has a guard ($m.guard$) which must be true before the message can be sent, a set of actions ($m.actions$), and means a conjunction of these actions ($m.actexp$). init is a pseudo-message for initialization. Actions are either propositions (being set true) or a commitment operation (being performed). Boolean negation (¬) is allowed in antecedent, consequent, and guard expressions to check state, but not in message meanings.

The sender *must not* send a *message* when its Boolean *guard* expression is false; it *may* send it when its *guard* is true. An agent implementation may have an internal, private guard that is different from the public guard specified in the protocol. An agent *satisfies* a protocol whenever each of its private guards is at least as restrictive as the corresponding public guard. This constraint ensures the

$$
\begin{array}{lcl}
\mathit{Protocol} & ::= & \langle \textsf{package}\ \mathit{packageName};\rangle^{?}\ \langle \textsf{import}\ \mathit{ProtoName};\rangle^{*}\ \textsf{protocol}\ \mathit{ProtoName}\ \langle \mathit{ArgList}\rangle^{*}\ \{ \\
 & & \textsf{role}\ \langle \mathit{RoleName};\rangle^{*}\ \textsf{prop}\ \langle \mathit{PropName};\rangle^{*}\ \textsf{commitment}\ \langle \mathit{Commit};\rangle^{*}\ \textsf{message}\ \langle \mathit{Message};\rangle^{*} \\
 & & \textsf{use}\ \langle \mathit{Usage};\rangle^{*}\} \\
\mathit{ArgList} & ::= & (\ \langle\langle \textsf{role}|\textsf{prop}|\textsf{commitment}\rangle\ \mathit{name}\rangle^{+}\ ) \\
\mathit{Usage} & ::= & \mathit{InstanceName} = \mathit{ProtoName}(\ \langle\langle \textsf{role}|\textsf{prop}|\textsf{commitment}\rangle\ \mathit{name} = \mathit{name}\rangle^{*}\ ) \\
\mathit{Commit} & ::= & \mathit{ComName} : \textsf{C}(\mathit{Debtor}, \mathit{Creditor}, \mathit{Ant}, \mathit{Csq}) \\
\mathit{Message} & ::= & \mathit{Snd} \rightarrow \mathit{Rcv} : \langle[\mathit{Guard}]\rangle^{?}\ \mathit{MsgName}\ \textsf{means}\ \{\mathit{Actions}\}\ | \\
 & & |\ \mathit{Snd} : \textsf{init}\ \mathit{MsgName}\ \textsf{means}\ \{\mathit{Actions}\} \\
\mathit{Debtor} & ::= & \mathit{RoleExp} \\
\mathit{Creditor} & ::= & \mathit{RoleExp} \\
\mathit{Ant} & ::= & \mathit{ActExp} \\
\mathit{Csq} & ::= & \mathit{ActExp} \\
\mathit{Snd} & ::= & \mathit{RoleName} \\
\mathit{Rcv} & ::= & \mathit{RoleName} \\
\mathit{Guard} & ::= & \mathit{MsgExp} \\
 & & \\
\mathit{Map} & ::= & \textsf{map}\ \mathit{MapName} : \mathit{ProtoName} \mapsto \mathit{ProtoName}\{\textsf{role}\langle \mathit{RoleMap};\rangle^{*} \\
 & & \textsf{prop}\langle \mathit{PropMap};\rangle^{*} \textsf{commitment}\ \langle \mathit{CommitMap};\rangle^{*}\} \\
\mathit{RoleMap} & ::= & \mathit{RoleName} \mapsto \mathit{RoleExp} \\
\mathit{PropMap} & ::= & \mathit{PropName} \mapsto \mathit{ActExp} \\
\mathit{CommitMap} & ::= & \mathit{ComName} \mapsto \mathit{CommitExp} \\
 & & \\
\mathit{RoleExp} & ::= & \mathit{RoleName}\ |\ \{\mathit{RoleName}\ \langle, \mathit{RoleName}\rangle^{*}\} \\
\mathit{MsgExp} & ::= & \mathit{MsgName}\ |\ \neg \mathit{MsgExp}\ |\ \mathit{MsgExp} \vee \mathit{MsgExp}\ |\ \mathit{MsgExp} \wedge \mathit{MsgExp}\ |\ \mathit{MsgExp} \rightarrow \mathit{MsgExp} \\
\mathit{ActExp} & ::= & \mathit{Action}\ |\ \neg \mathit{ActExp}\ |\ \mathit{ActExp} \vee \mathit{ActExp}\ |\ \mathit{ActExp} \wedge \mathit{ActExp}\ |\ \mathit{ActExp} \rightarrow \mathit{ActExp} \\
\mathit{CommitExp} & ::= & \mathit{ComName}\ |\ \mathit{CommitExp} \oplus \mathit{CommitExp} \\
\mathit{Actions} & ::= & \mathit{Action}\ \langle, \mathit{Action}\rangle^{*} \\
\mathit{Action} & ::= & \mathit{PropName}\ |\ \textsc{create}(\mathit{ComName})\ |\ \textsc{transfer}(\mathit{ComName})\ | \\
 & & \textsc{release}(\mathit{ComName})\ |\ \textsc{cancel}(\mathit{ComName})
\end{array}
$$

Figure 3.1: Proton input syntax in BNF.

agent does not violate the protocol's guard. Each message's meaning is a set of actions on propositions (SET and CLR) and commitments (CREATE, TRANSFER, RELEASE, and CANCEL).

The *Map* nonterminal describes the syntax for a mapping between two protocols (as Listing 5.1 exemplifies). A mapping maps individual roles, propositions, and commitments from the putative superprotocol to expressions in the putative subprotocol. *ProtoName*, *MapName*, *RoleName*, *PropName*, *ComName*, *MsgName*, and *Action* are names.

The serial composition operator, $\oplus$, chains two commitments together and is described in Section 3.2.1. We write $\bigoplus_i \mathsf{C}_i$ for a left-associated chain $((\mathsf{C}_1 \oplus \mathsf{C}_2) \oplus \ldots) \oplus \mathsf{C}_n$. In Section 3.2.3, we compare commitments between superprotocol and subprotocol, under an abstraction mapping M, using commitment covering ($\leq_M$).

Protocols may be parameterized with an ArgList of roles, propositions, and commitments, enabling a component protocol to be instantiated in composite protocols by use, with different values for those arguments. use functions like a macro expansion, expanding the contents (roles, propositions, commitments, and messages) of a component protocol into the composite protocol. For example, the protocol in Listing 3.1 instantiates an instances of constituent instances of *RequestResponse* using Proton's use statement.

Listing 3.1 Using (including) a constituent protocol.

```
1: protocol SomeComposite { (
2:     ...
3:     use
4:         A_B = agfil.RequestResponse(
5:             role REQUESTER = A, prop request=true,
6:             role SERVICE = B, prop response=true)
7:     ...
8: }
```

3.3.2 Proton Semantics

Proton's semantics is based on interpreted systems: it constructs an interpreted system from an input superprotocol, subprotocol, and mapping.

Each state g is a set of true propositions p_i. All propositions are false in the initial state g_0. Actions cause state transitions. For this paper, we use a simplified model of actions, assuming actions (i) always succeed, (ii) have definite outcomes (no uncertainty), and (iii) have no side-effects. The actions for role i, Act^i, are the propositional and commitment actions $Act^i = \{p_i\} \cup \{a(\mathsf{C}_j) | a \in Act_{\mathsf{C}}\} \cup \{nop\}$ where $\mathscr{A}$ is the set of protocol propositions, $\mathscr{C}$ is the set of protocol commitments, for all $p_i \in \mathscr{A}$,

$Act_{\mathsf{C}} = \{\textsc{create}, \textsc{transfer}, \textsc{release}, \textsc{cancel}\}$, for all $\mathsf{C}_j \in \mathscr{C}$, and *nop* represents no-operation.

Operationally, messaging is point-to-point and synchronous. All protocol state is stored in the "environment" (effectively, a distinguished agent), and is globally accessible by all roles (a current simplification). At each time step, the environment schedules one role to execute next (interleaved execution). When scheduled, the role's agent (i) determines which of its messages are currently enabled, by accessing the protocol's global state and evaluating each message's guard expression, (ii) chooses an enabled message to send or chooses "no-operation", (iii) performs all actions in message's meaning, in any order, and (iv) updates the protocol's state.

In every global state g of the interpreted system, each commitment C_i has a state, $\mathsf{C}_i.state \in Stat_{\mathsf{C}}$, where $Stat_{\mathsf{C}} = \{null, cond, detached, dis, xfer, rel, can\}$, whose value can be any of the states in Figure 2.2. We define propositions for the expressions $\mathsf{C}.state = x$ and $\mathsf{C}.state \neq x$ in each state g. For each commitment C_i, we evaluate the occurrence of the commitment operations using the four propositions:

$$\begin{aligned}
\textsc{create}(\mathsf{C}_i) &\triangleq \mathsf{C}_i.state \neq null \\
\textsc{transfer}(\mathsf{C}_i) &\triangleq \mathsf{C}_i.state = xfer \\
\textsc{release}(\mathsf{C}_i) &\triangleq \mathsf{C}_i.state = rel \\
\textsc{cancel}(\mathsf{C}_i) &\triangleq \mathsf{C}_i.state = can
\end{aligned}$$

As Section 5.1.1 mentions and Definition 5.2.6 formalizes, refinement depends upon a mapping between protocols to account for their different levels of abstraction. Specifically, we must map each (i) super-role to a set of sub-roles, (ii) super-proposition to a Boolean expression of sub-propositions, and (iii) super-commitment to an expression of sub-commitments. Proton combines two protocols and a mapping to (i) construct an interpreted system model $\mathscr{I}$ from the subprotocol's propositions, commitments, and guarded actions, as specified in Definition 5.2.2 and (ii) generate appropriate CTL formulas as specified in Section 5.3.5. The refinement in consideration holds if and only if the constructed model satisfies all the CTL formulas that Proton generates.

3.3.3 Proton Specification Examples

Listing 3.2 shows the Proton specification of protocol *Pay*. Lines 2–5 declare roles Payer and Payee, propositions *promise* and *pay*, and the commitment. Both *promiseMsg* and *payMsg* messages are sent by Payer to Payee. A message may be sent only if its guard (the expression between [and]) is true. The guard for *payMsg* in Line 8 is *promiseMsg*. If no guard is explicitly specified, as is the case for *promiseMsg* in Line 7, it is implicitly true. A message's meaning is expressed as a set of actions after means and between { and }.

The Proton specification for *PayViaMM* is shown in Listing 3.3. Middleman commits to Payer to pass along any payment it receives (Line 11). Payer will not pay Middleman without this commitment (Line 12). Since *payMMsg* has an implicit guard of true (Line 14), Middleman is allowed to pay early.

Listing 3.2 *Pay* Protocol

```
1: protocol Pay {
2:     role Payer; Payee;
3:     prop promise; pay;
4:     commitment
5:         C_pay : C(Payer, Payee, promise, pay);
6:     message
7:         Payer → Payee : promiseMsg means {promise, CREATE(C_pay)};
8:         Payer → Payee : [promiseMsg] payMsg means {pay};
9: }
```

Listing 3.3 *PayViaMM* Protocol

```
1: protocol PayViaMM {
2:     role Payer; MM; Payee;
3:     prop promise;
4:         payP;     //payment from Payer to MM
5:         payM;     //payment from MM to Payee
6:     commitment
7:         C_payP : C(Payer, Payee, promise, payP);
8:         C_payM : C(MM, Payer, payP, payM);
9:     message
10:        Payer → Payee : promiseMsg means {promise, CREATE(C_payP)};
11:        MM → Payer : pledgeMsg means {CREATE(C_payM)};
12:        Payer → MM : [promiseMsg ∧ pledgeMsg]
13:                        payPMsg means {payP};
14:        MM → Payee : payMMsg means {payM};
15: }
```

Chapter 4

Composing Commitment Protocols

Pankaj Telang and Anup Kalia contributed to the material in this chapter ([Gerard et al., 2013]).

4.1 Introduction

We adopt an interaction-oriented stance on multiagent systems, e.g., as applied in cross-organizational service engagements. We consider *(commitment) protocols*, which specify the interactions between two or more roles in terms of how their messages relate to their *commitments* [Singh, 1999], obtaining well-recognized benefits in dealing with the autonomy and heterogeneity of business partners [Baldoni et al., 2010b; Yolum and Singh, 2002].

Composition is a key construct in software engineering. We address two role-specific aspects of composition: *role requirements* which capture the benefits a role receives from the composite and *role accountability* which captures the commitments a role must make to other roles.

4.1.1 Real-Life Scenario: AGFIL

We illustrate our approach using the real-life, automobile insurance claims processing case for AGF Irish Life Holding (AGFIL) from Browne and Kellett [1999]. As Figure 4.1 illustrates, this case involves four parties along with PolicyHolder and Adjuster (not shown). AGFIL underwrites automobile insurance policies and covers losses incurred by PolicyHolder. Europ Assist (CallCenter) provides a 24-hour help-line service for receiving claims. Approved Repairers provide repair services. Lee Consulting Services (Coordinator) coordinates with AGFIL, repairers, and adjusters to handle a claim.

The traditional model of the AGFIL scenario describes the workflows of each partner along with how they relate to one another. Such a description, even if supported by standards such as BPMN [OMG, 2010], tightly couples the inner workings of the partners. Newer approaches deemphasize the inner workings and instead capture the interactions between the business partners more explicitly via

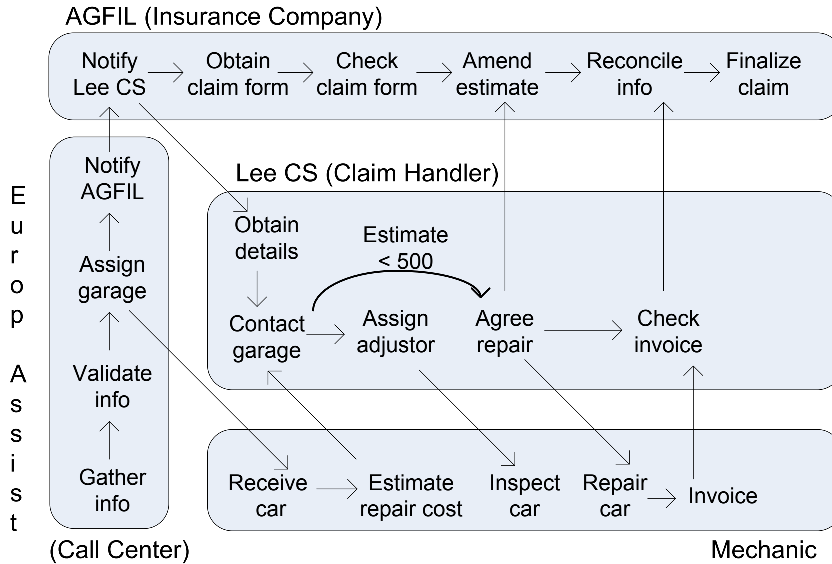


Figure 4.1: Traditional model of a cross organizational insurance claim processing.

a formal notation [HL7, 2007; RosettaNet, 2009; WS-CDL, 2005]. These approaches express constraints on the ordering and occurrence of the messages exchanged by the business partners.

In contrast, a commitment protocol emphasizes the social state of an interaction, expressed here in terms of commitments. A protocol describes the roles involved, the messages they exchange, and any preconditions on and effects of the messages on the social state. An agent adopts a role and enacts the specified protocol by autonomously choosing (in accordance with its internal policies) how to interact.

4.1.2 Contributions and Organization

Although protocols offer significant benefits over traditional approaches, protocols are not fully viable for the following reasons. One, specifying in one shot an adequate protocol for a complex scenario is nontrivial. Two, implementing agents who can play roles in such a comprehensive protocol is difficult because the differing details of the protocols complicate reusing parts of agent implementations. Our contribution shows how complex protocols can be constructed by composing existing protocols. Previous relevant research falls into these categories: (a) commitments but not composition [Gerard and Singh, 2013]; (b) composition but no commitments [Miller and McGinnis, 2008; Singh, 2011]; and (c) composition and commitments. The last category can be categorized as (c1) purely abstract description without a specification language or tools [Mallya and Singh, 2007]; (c2) composition of commitment-based protocols based on regulative constraints [Marengo, 2013]; and (c3) our approach to composition of commitment-based protocols based on role responsibilities and accountabilities.

Our approach, *Positron*, extends Proton Gerard and Singh's [2013] Proton approach to provide a clear syntax and semantics for composite protocols. Where Proton checks protocol refinement, Positron composes protocols. Positron (a) recursively expands nested constituent protocols; (b) introduces *composite protocol diagrams* as a graphical notation, conveying important features of the composite protocol to business and technical stakeholders; (c) introduces *role requirements* and *role accountabilities*; (d) incorporates a methodology for composing commitment protocols; and (e) implements a decision procedure and mechanical verification of protocols with respect to role requirements, role accountabilities, and enactments, compiling formulas to temporal logic, and employing MCMAS [Lomuscio et al., 2009], a leading model checker, to verify if the composite protocol satisfies those formulas.

We describe relevant background, the major elements of our technical approach, and our methodology for constructing composite protocols. We evaluate our approach by modeling protocols from the insurance, manufacturing, and healthcare domains, that other researchers have studied, and summarize our results and experiences. We conclude with a discussion of the relevant literature and future work.

4.2 Technical Approach

Positron provides a formal language in which to express composite protocols based on existing constituent protocols (Section 3.3). Recall that Proton provides a language for capturing roles, propositions, and messages with guards on when they can be sent, and their effects on the commitments of roles [Gerard and Singh, 2013]. Positron augments the Proton language by adding constructs to define a composite protocol *using* a set of parameterized constituent protocols and defines a protocol composition methodology.

Further, while it accepts and verifies any CTL expression, Positron introduces five constructs for common verification patterns when composing protocols: *Req* function for role requirements, *coupling commitments* for role accountabilities, and three *path expressions* for good and bad enactments.

4.2.1 Protocol Composition

Positron supports nested composition of protocols. A composite protocol P can use (or include) a set of constituent protocol instances with a use statement

$$uses(P) = \{q : Q(\overline{x} = \overline{p})\}$$

where q is a constituent instance name, Q is a protocol type, $\overline{p}$ is a set of arguments passed by P, and $\overline{x}$ is a matching set a parameters accepted by constituent Q. Arguments and parameters are named and

have a type of either role or proposition. The argument and parameter sets must contain matching names and types.

Positron expands P to produce a new, flatter protocol P'. Expansion replaces each parameter identifier in Q with its corresponding argument, and replaces each non-parameter identifier a new, unique name. Unique names are constructed by prepending the unique instance name q to each element name in Q.

Definition 4.2.1. *Given a set of arguments $\overline{p}$, a parameterized constituent protocol instance Q that accepts a set of parameters $\overline{x}$, and the sets $\overline{p}$ and $\overline{x}$ agree in both name and type. Define $Q^{\overline{x}}_{\overline{p}}$ as Q in which (1) every parameter identifier in $\overline{x}$ is replaced with its corresponding argument in $\overline{p}$, and (2) every non-parameter identifier in Q is made unique by prepending q to its name.*

Protocol expansion of a composite P containing multiple constituent instances $\{q : Q(\overline{x} = \overline{p})\}$ is the union of P and $Q^{\overline{x}}_{\overline{p}}$, and removing the expanded use statement. The definition expands any single constituent.

Definition 4.2.2. *Given a composite protocol P that uses multiple constituent protocol instances $\{q : Q(\overline{x} = \overline{p})\}$, where P passes a set of arguments $\overline{p}$, Q accepts a set of parameters $\overline{x}$, and the sets $\overline{p}$ and $\overline{x}$ agree in name and type. Then protocol $P' = expand(P,\ q : Q(\overline{x} = \overline{p}))$ is the expanded version of P and Q, and is defined as*

$$
\begin{aligned}
roles(P') &:= roles(P) \cup role(Q^{\overline{x}}_{\overline{p}}) &(4.1)\\
parms(P') &:= parms(P) &(4.2)\\
props(P') &:= props(P) \cup props(Q^{\overline{x}}_{\overline{p}}) &(4.3)\\
commitments(P') &:= commitments(P) \cup commitments(Q^{\overline{x}}_{\overline{p}}) &(4.4)\\
messages(P') &:= messages(P) \cup messages(Q^{\overline{x}}_{\overline{p}}) &(4.5)\\
checks(P') &:= checks(P) \cup checks(Q^{\overline{x}}_{\overline{p}}) &(4.6)\\
uses(P') &:= (uses(P) - q) \cup uses(Q^{\overline{x}}_{\overline{p}}) &(4.7)
\end{aligned}
$$

where $roles(Q)$, $parms(Q)$, $props(Q)$, $commitments(Q)$, $messages(Q)$, $checks(Q)$, and $uses(Q)$ refer to the corresponding element sets of protocol Q.

Repeated application of the definition expands all constituent instances.

4.2.2 Role Requirements

Role requirements are the requirements that an agent playing a role places on the composite protocol. A designer specifies a role requirement in the Positron language, which Positron compiles into a CTL formula. As an example, in the AGFIL scenario, POLICYHOLDER expect his car will be repaired if he

has an accident. Positron could compile this requirement into the CTL specification: $\mathbf{AG}(accident \rightarrow \mathbf{AF}\, repair)$.

However, such a requirement ignores *business exceptions*: a commitment may fail because its debtor either chooses not to, or is prevented by circumstances from, discharging it. In verifying a role requirement, we cannot assume commitments are never canceled. Rather, we state role R's requirement as: if R fulfills all its commitments and p holds at any state, then always eventually, either q holds or a role other than R cancels one of its commitments. If R's requirement fails because R cancels a commitment, that is not a fault of the protocol, but of R. In CTL, where *r.anyCancel* is true if and only if role r cancels any of its commitments, this is

$$Req(R,p,q) \quad ::= \quad \mathbf{AG}(p \rightarrow \mathbf{AF}(q \vee \bigvee_{r \neq R} r.anyCancel))$$

In AGFIL, one of POLICYHOLDER's role requirements is captured as: if INSURER offers coverage, I paid the premium, and I have an accident, then my car will be repaired: $Req(\text{PH}, coverage \wedge premium \wedge accident, repair)$

4.2.3 Enactment Requirements

Although capturing all possible enactments is not feasible, designers often know of specific good and bad enactments. We use the specified *enactments* as bases for verifying a composite protocol to assist designers in refining the protocol specification (e.g., its constituent protocols and coupling commitments) or the requirements. Our notion of enactments resembles scenarios from scenario-based requirements engineering [Filippidou, 1998]. In essence, each enactment corresponds to a unit test in software engineering.

We use model checking to verify enactments. We introduce three recursive functions to simplify enactment specification. Given an enactment list L, which is an ordered list of Boolean expressions over states and messages, where $head(L)$ is the first element in list L, and $tail(L)$ is L without the first element, let

$$EXPath(L) \quad ::= \quad \begin{cases} head(L) \wedge \mathbf{EX}(EXPath(tail(L))) & \text{if } |L| > 1 \\ \mathbf{EX}(L) & \text{if } |L| = 1 \end{cases}$$

$$EFPath(L) \quad ::= \quad \begin{cases} \mathbf{EF}(head(L) \wedge EFPath(tail(L))) & \text{if } |L| > 1 \\ \mathbf{EF}(L) & \text{if } |L| = 1 \end{cases}$$

$$EUPath(r,L) \quad ::= \quad \begin{cases} \mathbf{E}(\neg r \;\mathbf{U}\; (head(L) \wedge EUPath(r, tail(L)))) & \text{if } |L| > 1 \\ \mathbf{E}(\neg r \;\mathbf{U}\; L) & \text{if } |L| = 1 \end{cases}$$

EXPath specifies a list of states, beginning at a start state, that must appear consecutively without

skipping over other states. In protocols with many constituents, *EXPath* can be too strong a constraint, since it precludes interleaving of constituent protocols.

EFPath specifies a list of states that must appear in order, but allows other states to be interleaved. This is a weaker constraint than *EXPath*.

EUPath specifies a list of states that must appear in order, and constrains which states can be interleaved in the path. Expression r identifies which states must *not* be interleaved in the path. An *EUPath* constraint is stronger than *EFPath* and weaker than *EXPath*.

Two enactments from AGFIL are

$$EFPath(accident, deliverReq, deliverCar, \ldots, repair)$$
$$\neg EFPath(repair, accident)$$

4.2.4 Coupling Commitments

The constituent protocols occurring in a (nontrivial) composite protocol must be interrelated. In a multiagent system, some role must be accountable for ensuring constituent protocols are properly interrelated. We capture the *role accountability* implied by such an interrelationship via a *coupling commitment*. A coupling commitment's debtor is the accountable role, and its creditors are (in general) the union of all roles connected by the interrelated constituent protocols, minus the debtor. Like any commitment, debtor commits to discharge consequent if antecedent becomes true.

$$\mathsf{C}_{\text{accountable role},\{\text{interrelated roles}\}}(antecedent, consequent)$$

Two coupling commitments from AGFIL are

$$\mathsf{C}_{CC,\{PH,\ Re\}}(deliverReq, notify\textsc{Re})$$
$$\mathsf{C}_{PH,\{CC,Re\}}(deliverReq \land approval, deliverCar)$$

4.2.5 Verification

Positron reads designer written source code for the composite and constituent protocols and generates a single MCMAS input file. MCMAS reads the input, builds the model, and reports whether each CTL formula holds in the model.

Figure 4.2 shows a portion of the state space Positron generates for verification from AGFIL's constituent protocols and coupling commitments. The start state is denoted by the unlabeled line in the top left. Solid black lines are valid transitions (messages); dashed red lines are invalid transitions. Since the message guard for *coverage* is *premium*, *coverage* can occur only after *premium*, making s_1 an invalid start state. The other states are also invalid start states.

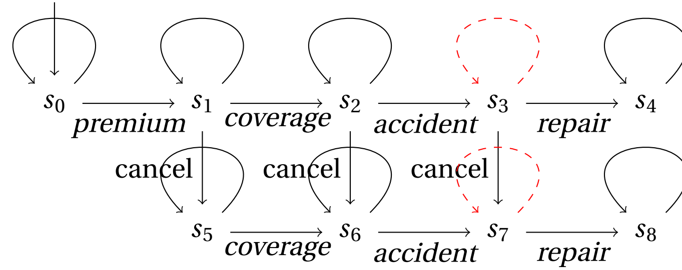


Figure 4.2: Selected states and transitions for AGFIL.

Notice that the top row (*premium*, *coverage*, *accident*, and *repair*) begins a good enactment. Positron can verify the existence of this path using *EFPath* requirement. Further, Positron can ensure that the model is free of specific bad enactments, for example, that s_1 must not be a start state.

To capture the only compliance requirement for commitments, Positron generates a model checking fairness constraint for each commitment: a commitment must not remain unconditional and unresolved forever. Red dashed loops are invalid because they violate a fairness constraint.

A composite protocol may fail to satisfy role or enactment requirements for different reasons.

Ver_{RR}: If a role requirement formula based on the Req function fails, then coupling commitments are missing; add coupling commitments that require agents to act.

Ver_G: If a good enactment formula fails, then either (Ver_{GG}) some message guards are too strong; weaken guards to validate additional good transitions. Or (Ver_{GC}) some commitment become detached, but can never resolve; weaken guards to validate transitions that satisfy the commitment's consequent.

Ver_B: If a bad enactment formula fails, then some message guards are too weak; strengthen guards to invalidate existing incorrect transitions.

4.2.6 Composite Protocol Diagrams

We propose *composite protocol diagrams* (CPDs) as a notation that displays the essence of a composite protocol both to business analysts and technical designers, who collaborate in its construction. We use CPDs in this paper to help visualize a large amount of protocol information, but we defer evaluation of this visual notation to future work. CPD diagrams focus designers' attention by summarizing high-level business relationships between the roles as reusable constituent protocols, hiding the details of each constituent.

Figure 4.3 shows the CPD for composite protocol AGFIL. It contains constituent protocol instance PH-IN of type *Exchange*. PH $\xrightarrow{R1}$ PH-IN shows role PH enacts role *R1* in constituent protocol PH-IN. The two unlabeled circular arcs centered on POLICYHOLDER represent coupling commitments, implicitly

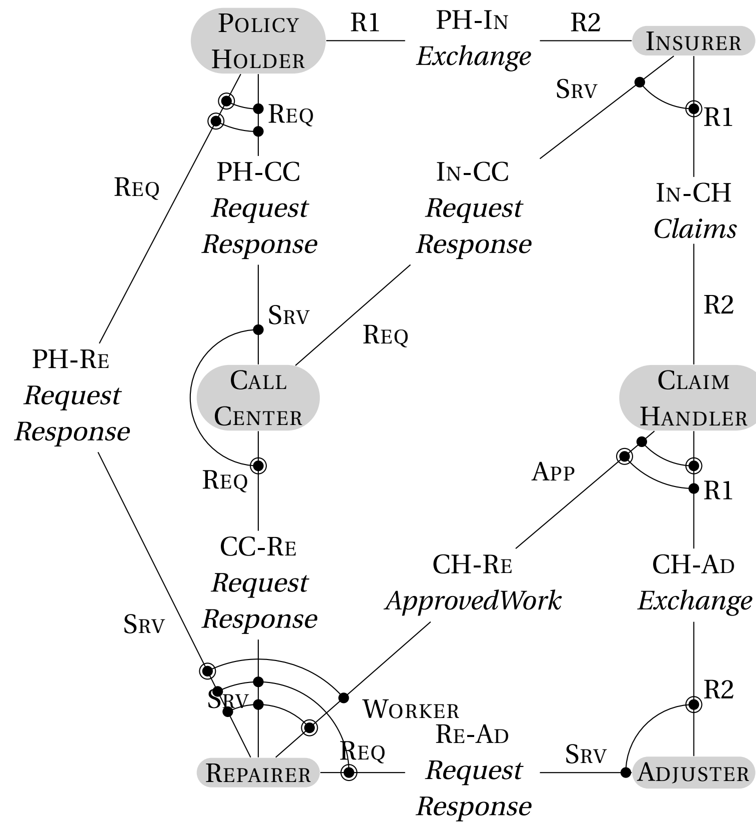


Figure 4.3: Composite protocol diagram for AGFIL. A CPD for a *composite* protocol displays its contained *constituent* protocol instances (unshaded labeled nodes). *Composite roles* (shaded labeled nodes) and constituents are connected by *constituent roles* (straight labeled edges). Unlabeled circular arcs represent coupling commitments with input constituents (dot) and output constituents (target symbol).

named PH_1 (inner) and PH_2 (outer). (role names Req and R1 label straight edges, not arcs.)

4.3 Methodology

This section describes, and Table 4.1 summarizes, our iterative methodology to develop a CPD such as that of Figure 4.3.

CM1 (Roles): Identify all roles with a business function, i.e., that potentially send messages or enter into commitments with others.

CM2 (Constituent Selection): Identify all business relationships among different subsets of roles and identify constituent protocols that realize such relationships. Examine message flows since they suggest the presence and nature of the constituent protocols. If a suitable protocol is known, use

Table 4.1: Inputs and outputs for each step of the composition methodology.

Step	Name	Inputs	Outputs
CM1	Roles	Background and requirements	Roles in the composite
CM2	Constituent Selection	Role relationships and protocol library to be composed	Constituent protocols
CM3	Role Requirements	Role business needs	Role requirements
CM4	Enactments	Background knowledge of requirements	Good and bad enactments
CM5	Coupling Commitments	Enactments complete CPD	Coupling commitments;
CM6	Positron	All artifacts	Positron source code
CM7	Verification	Positron source code	Model checker results

it; else apply the methodology recursively. The constituent name is the protocol name in the `use` statement of the Positron source.

CM3 (Role Requirements): Specify each role's requirements as Req functions.

CM4 (Enactments): Incrementally specify good and bad enactments. The enactments can be developed by a role-playing process similar to that described by Parunak [1996]. They should cover representative enactments that support or invalidate each of the role requirements identified in the previous step. Manually tracing enactments throughout a model is tedious and error prone. Explicitly specifying enactments enables the model checker to trace the enactments mechanically, after each incremental composite change.

CM5 (Coupling Commitments): Examine each good enactment from beginning to end, and assume roles do only what is *minimally* required to discharge their commitments. When adjacent steps of a good enactment are messages from the same constituent protocol, verify that the constituent accurately enacts those steps, or select a different constituent. When adjacent steps of an enactment are messages from different protocols (where some role receives a message in one constituent and then sends a message in another constituent), add a coupling commitment for that role.

CM6 (Positron): Given the previous methodology artifacts, write the Positron source code, including roles, constituent protocols, role requirements, coupling commitments and enactments.

CM7 (Verification): Run Positron and MCMAS to verify all formulas for role and enactment requirements.

4.4 Evaluation

To demonstrate the broad applicability of Positron, we evaluate our contributions by modeling protocols from the insurance, manufacturing, and healthcare domains.

4.4.1 AGFIL Evaluation

We extend the AGFIL protocol by adding (a) POLICYHOLDER and accident reporting; (b) ADJUSTER and the redirection of two messages between CLAIMHANDLER and REPAIRER through ADJUSTER; (c) payments from REPAIRER to CLAIMHANDLER to INSURER; (d) a protocol for premiums and coverage between POLICYHOLDER and INSURER; and (e) REPAIRER returning the car.

At the top of Figure 4.3, POLICYHOLDER (enacting role R1) purchases insurance from INSURER (enacting role R2) using an instance of constituent protocol *Exchange* named PH-IN. POLICYHOLDER reports accidents to CALLCENTER using an instance of constituent protocol *RequestResponse* named PH-CC. CALLCENTER notifies INSURER (IN-CC), and assigns and notifies REPAIRER (CC-RE). INSURER passes the claim to CLAIMHANDLER (IN-CH). POLICYHOLDER and REPAIRER exchange the damaged and later repaired car (PH-RE). CLAIMHANDLER, REPAIRER, and ADJUSTER inspect the car and approve repairs (CH-AD, RE-AD and CH-RE).

CM1 (Roles): Figure 4.1 refers to specific agents (companies), not roles. Declare six roles: role INSURER (abbreviated IN) for agent AGFIL, CALLCENTER (CC) for EUROP ASSIST, and CLAIMHANDLER (CH) for LEE. The other roles are POLICYHOLDER (PH), REPAIRER (RE), and ADJUSTER (AD). At the end of this step, the CPD diagram in Figure 4.3 shows only the six shaded role nodes.

CM2 (Constituent Selection): Assume protocols *RequestResponse* and *Exchange* (where two roles swap items) already exist. Designers recursively create *Claims* for IN-CH and *ApprovedWork* for CH-RE. Designers add the constituent protocol nodes and edges to Figure 4.3, completing all CPD nodes and edges.

CM3 (Role Requirements): POLICYHOLDER requires: (1) if he has coverage, pays his premium, and has an accident, his car is repaired; (2) if he delivers his car to REPAIRER, his car is returned. INSURER requires: if a claim is filed, the claim is finalized. All roles except POLICYHOLDER require payment if they perform their tasks. All these are described as Req functions. Role requirements can *also* be specified directly in CTL, e.g., INSURER requires no car repairs without an inspection: $AG(\neg(repair \wedge \neg inspectCH))$.

CM4 (Enactments): An important good enactment is that of reporting an accident and getting car repaired: (a) POLICYHOLDER reports an accident to CALLCENTER (PH-CC); (b) CALLCENTER assigns and notifies REPAIRER to repair the car (CC-RE); (c) CALLCENTER asks POLICYHOLDER to deliver his car to a specific REPAIRER (PH-CC); (d) POLICYHOLDER delivers car to REPAIRER (PH-RE); Remaining steps are omitted. Performing repairs before an accident is reported is a bad enactment: (e) car repaired; (f) accident reported.

CM5 (Coupling Commitments): Between messages (a) and (b) of the accident-reporting enactment (see previous step), if POLICYHOLDER reports an accident, CALLCENTER assigns and notifies REPAIRER: $C_{CC,\{PH,RE\}}(accident, notifyRE)$. Between messages (c) and (d), if CALLCENTER asks POLICYHOLDER to deliver his car to REPAIRER, he does so.

$\mathrm{PH}_1 = \mathsf{C}_{\mathrm{PH},\{\mathrm{CC},\mathrm{Re}\}}(\mathit{deliverReq} \wedge \mathit{approval}, \mathit{deliverCar})$

Adding arcs, the complete AGFIL CPD is Figure 4.3.

CM6 (Positron): Listing 4.1 shows some lines from the Positron source file for AGFIL. Lines 2-3 declares all roles and propositions. Line 4 instantiates an instance of *Claims* named IN-CH. Lines 10 and 11 are two coupling commitments. Line 14 lists one of POLICYHOLDER's role requirements. Line 15 lists an INSURER requirement as explicit CTL. Line 17 verifies the good, accident-reporting enactment that must exist in the composite, and Line 18 verifies a bad enactment that must not exist.

Listing 4.1 Positron source for AGFIL.

```
1:  protocol AGFIL {
2:    role PH; In; CC; CH; Re; Ad;
3:    prop accident; deliver; repair; paid; ...
4:    use IN-CH : Claims(
5:        role R1 = IN, role R2=CH,
6:        prop pre1 = true, prop pre2=reportCH,
7:        prop act1 = payCH, prop act2=repairIn);
8:      ...
9:    commitment
10:     CC1 : C_{CC,{PH, Re}}(deliverReq, notifyRE);
11:     PH1 : C_{PH,{CC,Re}}(deliverReq ∧ approval, deliver);
12:     ...
13:   formula
14:     Req(IN, coverage ∧ premium ∧ accident, repair);
15:     AG(¬(repair ∧ ¬inspectCH));
16:     ...
17:     EFPath(accident, deliverReq, ..., repair);
18:     ¬EFPath(repair, accident);
19:     ...
20: }
```

The conversion of Positron to MCMAS maps role to role; proposition to boolean; commitment to enum and fairness condition; message to action; and formula expansion to formula. It also defines proposition and commitment evolutions, maps high-level Positron expressions to low-level MCMAS expressions, tracks each agent's last action and anyCancel values, and generates a large number of proposition and commitment state evaluation statements.

CM7 (Verification): Running Positron and MCMAS successfully verified nine CTL formula: eight role and one enactment requirements. Removing any single coupling commitment caused one or more formulas to fail.

4.4.2 Quote To Cash Evaluation

We further evaluate our methodology on Quote To Cash (QTC), an important business process that supports manufacturing supply chains [Oracle, 2009]. Figure 4.4 shows its CPD.

Figure 4.4: CPD for Quote To Cash.

CM1 (Roles): Identify the six roles: CUSTOMER, RESELLER, DISTRIBUTOR, SELLER, SHIPPER1, and SHIPPER2.

CM2 (Constituent Selection): A CUSTOMER orders goods and services from RESELLER using constituent protocol *CommercialTran* (*ComTran*). RESELLER fulfills the order by *Outsourcing* to DISTRIBUTOR. DISTRIBUTOR orders good from SELLER using *CommercialTran*. SELLER arranges shipping with SHIPPER2, and DISTRIBUTOR arranges shipping with SHIPPER1, using additional instances of *Outsourcing*. SELLER provides a customer support contract to CUSTOMER though *StandingService*.

CM3 (Role Requirements): If CUSTOMER pays, he receives goods and services. If RESELLER pays

DISTRIBUTOR, he receives shipment. Except for CUSTOMER whenever a role performs its task, it gets paid.

CM4 (Enactments): Two good enactments are identified beginning with CUSTOMER placing an order and ending with fulfillment: one if DISTRIBUTOR has goods in stock, one if it restocks from SELLER. A bad enactment is fulfilling an order before it is verified.

CM5 (Coupling Commitments): CUSTOMER couples CU-RE and CU-RE-DI: if CUSTOMER receives a shipment, he pays RESELLER.

$\mathrm{CU}_1 = \mathsf{C}_{\mathrm{CU},\{\mathrm{DI},\mathrm{RE},\mathrm{S1}\}}(\mathit{ship}\mathrm{S1}, \mathit{pay}\mathrm{RE})$

RESELLER couples CU-RE and CU-RE-DI: whenever RESELLER receives an order, he orders from DISTRIBUTOR.

$\mathrm{RE}_1 = \mathsf{C}_{\mathrm{RE},\{\mathrm{DI},\mathrm{S1}\}}(\mathit{order}\mathrm{CU}, \mathit{order}\mathrm{DI})$

CM6 (Positron): The Positron source for Quote To Cash is omitted, but is similar in form to Listing 4.1. From the CPD and its annotations, we can produce Positron source code so that its properties can be mechanically verified.

CM7 (Verification): Positron and the model checker successfully verified 16 CTL formulas: 13 role and three enactment requirements. Removing any single coupling commitment caused one or more formulas to fail.

This CPD summarizes a complex protocol, which can otherwise be represented using a large number of sequence diagrams [Telang and Singh, 2012].

4.4.3 ASPE Evaluation

We also consider the healthcare process for breast cancer diagnosis, as described by an HHS committee [ASPE, 2010]. The resulting *ASPE* protocol contains five roles (for convenience, we associate feminine pronouns with PATIENT, RADIOLOGIST, and REGISTRAR and masculine pronouns with PHYSICIAN and PATHOLOGIST).

The process begins when PATIENT visits a primary care physician (PHYSICIAN), who detects a suspicious mass in her breast. He sends the patient to RADIOLOGIST for a mammography. If RADIOLOGIST notices suspicious calcifications, she sends a report to PHYSICIAN recommending a biopsy. PHYSICIAN requests the RADIOLOGIST perform a biopsy, who then collects a tissue specimen from PATIENT, and sends it to PATHOLOGIST. PATHOLOGIST analyzes the specimen, and performs ancillary studies. If necessary, PATHOLOGIST and RADIOLOGIST confer to reconcile their results and produce a consensus report. PHYSICIAN reviews the integrated report with PATIENT to create a treatment plan. PATHOLOGIST forwards his report to REGISTRAR who adds PATIENT to a state-wide cancer registry.

There are only two coupling commitments in ASPE. PHYSICIAN has no coupling commitments because it is his choice whether PATIENT needs mammogram and biopsy exams from RADIOLOGIST.

Figure 4.5: Composite protocol diagram for ASPE.

4.5 Results and Experience

To demonstrate the broad applicability of Positron, our methodology was successfully able to create composite protocols for scenarios from three different business domains: AGFIL from insurance [Browne and Kellett, 1999]; Quote To Cash, an important business process for manufacturing supply chains [Oracle, 2009]; and ASPE, a healthcare process for breast cancer diagnosis [ASPE, 2010]. Positron successfully verified all role and enactment requirements. Table 4.2 shows statistics and timings for these three protocols.

Model Verification: We encountered and fixed a number of verification failures. Good enactment failures (Ver_{GG} and Ver_{GC}) were generally easier to fix, since they only require that some path exist. Bad enactment failures (Ver_{B}) require the impossibility of a particular enactment.

Model Validation: Identifying requirements was straightforward, but some initial specifications were incorrect because preconditions were missed. For example, POLICYHOLDER's role requirement on Line 14 initially failed (Ver_{RR}) because coupling commitment PH_1 on Line 11 did not include *approval*; REPAIRER will not repair a car just because it is delivered to him. Repairs must also be approved by CALLCENTER. And, an accident is insufficient to get POLICYHOLDER's car is repaired; POLICYHOLDER

Table 4.2: Positron statistics. (M is 10^6 and G is 10^9.)

Composite Metric	**AGFIL**	**QTC**	**ASPE**
Constituent instances	11	6	12
Roles	6	6	5
Propositions	22	37	18
Commitments (all)	24	43	12
Coupling commitments	9	21	2
Messages	22	55	20
CTL Formulas	9	17	14
Role requirements	8	13	7
Enactment requirements	1	4	7
Positron statements	94	164	81
State space size	120M	381G	1.47M
Positron processing time	1.98s	3.16s	1.68s
MCMAS processing time	4.29s	1274s	5.78s
Total time	6.27s	1278s	7.46s

must also have a policy and pay the premium.

Positron generates model checking fairness conditions to ensure all unconditional commitments eventually resolve. Initially, AGFIL's good enactment on Line 17 mysteriously failed because, even though the model allowed all the transitions, the good enactment had unresolvable commitments (invalid by commitment fairness conditions). We corrected the model so all commitments could resolve.

4.6 Discussion

Positron gains an advantage over traditional approaches by focusing on high-level business relationships realized as constituent protocols, and by focusing on commitments rather than control flow. Because role accountabilities are stated as commitments, if a requirement fails, we can trace the failure back to a specific role.

CPDs summarize relevant details about a composite protocol and we expect they will prove valuable, because they bring together both technical and business descriptions of protocols, helping bridge the Business-IT Divide [Smith and Fingar, 2002].

4.6.1 Relevant Literature

Table 4.3 compares Positron with other work. Some papers propose a protocol specification language, and some propose an accompanying protocol specification methodology. Some papers address

Table 4.3: Approach comparison. Column abbreviations and citations are Po=Positron; Pr=Proton [Gerard and Singh, 2013]; DA=Desai et al. [2009], DO=Desai et al. [2005]; Dv=Desai et al. [2007]; DM=Desai et al. [2007]; T=Telang and Singh [2012]; Y=Yolum [2007]; Mi=Miller and McBurney [2011]; G=Günay et al.[2012]; C=Cheong and Winikoff [2009]; Mc=McGinnis and Robertson [2005]; L=Lomuscio et al. [2012]; and Ma=Marengo [2013]. Check marks show the significant topics addressed by each paper. The cell contents of the verification rows indicates whether the paper discusses (D) or mechanizes (M) verification.

Significant Topics	**Po**	**Pr**	**DA**	**DO**	**Dv**	**DM**	**T**	**Y**	**Mi**	**G**	**C**	**Mc**	**L**	**Ma**
Protocol specification	✓	✓	✓	✓	–	✓	✓	✓	✓	✓	✓	✓	–	✓
Methodology	✓	–	✓	✓	–	–	✓	–	–	✓	✓	–	–	–
Single protocol	–	✓	✓	✓	✓	–	–	✓	✓	✓	–	✓	✓	✓
Protocol patterns	–	–	–	–	–	–	✓	–	–	–	–	–	–	–
Protocol composition	✓	–	✓	✓	✓	✓	–	–	✓	–	✓	–	✓	✓
Requirements verification														
Business	M	–	–	–	M	–	M	–	–	D	–	–	M	M
Protocol-to-protocol	–	M	–	–	–	–	M	–	–	–	–	–	M	–
Protocol	M	–	–	–	M	–	–	M	M	D	D	–	–	M
Inter-constituent	M	–	–	–	–	M	–	–	–	–	–	–	–	–
Role-specific	M	–	–	–	–	–	–	–	–	D	–	–	M	–
Enactments	M	–	–	–	–	–	M	–	–	–	D	–	M	M
Other	M	M	–	–	M	M	M	M	M	D	–	–	M	M

single protocols in isolation; some address common patterns within protocols; some address the composition of multiple protocols to create new composite protocols. Of those papers that address verification, some address business level requirements; some address verification properties between two protocols or models (such as protocol refinement); some address protocol-wide properties; some verify properties that must hold between the constituents of a composite protocol; some formulate role-specific properties; some formulate good or bad enactment properties; and some address other verification topics not addressed above.

Desai et al. [2009] propose OWL-P [2005; 2007] and MAD-P [2007] for specifying and verifying commitment protocols and their compositions. They employ axioms to specify a composition. These approaches suffer from a key drawback: axiom violations are not assigned to any particular role. In contrast, Positron employs coupling commitments with clear role accountability for the effects of one constituent protocol on others. Further, Amoeba is purely manual, whereas Positron incorporates mechanical verification. Adopting Amoeba's event ordering idea would add flexibility to our approach, but more granular parameterizations of constituents provides the same functionality.

Telang and Singh [2012] (T&S) describe a methodology for business modeling that captures the commitments to be created among the parties by melding selected business patterns. In contrast, a protocol in Positron additionally specifies the messages and guards, and the protocols are first class entities that retain their identity in the composite protocol, yielding improved modularity and modifiability. Most significantly, T&S's approach verifies if one implementation is sound with respect to the model. In contrast, Positron verifies if the model itself is sound.

Yolum [2007] proposes generic correctness properties of commitment protocols for design-time verification, but does not address composite protocols. She considers generic properties, whereas we consider role-specific business requirements. It would be interesting to formulate Yolum's generic correctness properties in Positron to help improve protocol designs.

Miller and McBurney [2011] (M&M) propose the $\mathscr{RASA}$ language based on propositional dynamic logic (PDL) to specify and compose protocols. $\mathscr{RASA}$'s preconditions, actions and postconditions correspond to Positron's guards, messages and meanings. Positron additionally incorporates role requirements, coupling commitments, and good and bad enactment paths, making Positron practically viable (intentionally omitted from $\mathscr{RASA}$). Theses are important in naturally describing business protocols, as we demonstrated above. Whereas M&M describe a custom reasoner, we rely on CTL semantics as realized in MCMAS.

Günay et al. [2012] treat protocols as sets of commitments and propose automatically generating such sets from an agent's beliefs, goals, and capabilities. In contrast, we offer a semiautomatic approach where a tool helps designers compose existing protocols. Automatic generation is attractive but may not be feasible for complex settings, although a hybrid approach of developing atomic protocols mechanically and composite protocols with human assistance might be viable.

Cheong and Winikoff [2009] describe the Hermes system for goal-oriented interaction. They focus

on interaction-level goals, where we focus on role-level requirements and commitments. Their action sequence diagrams capture only good enactments.

McGinnis and Robertson [2005] propose an approach in which an agent sends a protocol specification to other agents at runtime, as a way to accomplish dynamic, runtime, protocol adaptation. They remark that their approach lacks a way to prevent the agents from making an undesirable change to a protocol. If their protocols were augmented with commitments, Positron can help address this gap. For example, an agent may not remove a message from a protocol that brings about the consequent of a detached commitment. While they describe rules for dynamically changing protocols, they do not address formal verification of interaction properties.

Lomuscio et al. [2012] semiautomatically compile and verify contract-regulated service compositions. They use temporal-epistemic logic to check whether agents comply with their contracts using MCMAS (the same tool we employ). A crucial difference is that whereas Lomuscio et al. consider service compositions, we consider protocol compositions. Since a protocol has a footprint distributed across two or more roles, dealing with their compositions is inherently more subtle. In Table 4.3, this is classified as a kind of composition, and includes verification between actual behaviors and contractually correct behaviors (two "protocols").

Marengo [2013] considers a related problem where protocols are composed (grafted). Marengo uses regulative specifications (constraints) using Linear Temporal Logic (LTL). We propose a methodology and use role responsibilities, role accountabilities, and path enactments using CTL. These idea sets are complementary and worth further study.

BPMN [OMG, 2010] is an industry standard notation for business processes. Unlike Positron, BPMN is only semiformal, and does not lend itself to formal verification. BPMN's emphasis on control flow results in rigid processes. Positron minimally constrains the participants by specifying the process in terms of commitments. Protocols are building blocks in Positron—a process is composed of protocols.

4.6.2 Future Work

The foregoing opens up useful directions for future work. At the theoretical level, treating the goals of the participants [Günay et al., 2012] is natural. At the practical level, generating enactments via tooling would be valuable. At the empirical level, evaluating the effectiveness of Positron (the approach and the tool) with professional developers on cross-organizational business processes would be necessary to promote the adoption of Positron by industry.

Chapter 5

Formalizing and Verifying Protocol Refinements

Software engineering using protocols presupposes a formalization of protocols and a notion of the *refinement* of one protocol by another. Refinement for protocols is both intuitively obvious (e.g., *PayViaCheck* is clearly a kind of *Pay*) and technically nontrivial (e.g., compared to *Pay*, *PayViaCheck* involves different participants exchanging different messages). This chapter formalizes protocols and their refinement.

5.1 Introduction

We focus our attention on business service engagements as realized over the Internet. In current practice, such an engagement is defined rigidly and purely in operational terms. Consequently, the software components of the business partners involved are tightly coupled with each other, and depend closely on the engagement specification. Even small changes in one partner's components must be propagated to others, even when such changes are inconsequential to the business being conducted. Conversely, if the model leaves the engagements unstructured, humans must carry out the necessary interactions manually, with concomitant loss in productivity. We motivate protocols as providing a happy middle ground between rigid automation and flexible manual execution.

Specifically, in contrast with traditional approaches, we model each partner as an autonomous *agent*. The agents participate in a *(business) protocol* to realize a service engagement. A protocol describes a pattern of communication between agents. Based on the foregoing, we formulate the following key requirements on a suitable formalization of protocols. First, a protocol is *public*, meaning that it pertains to the messages sent and received by participating agents, not how those agents are implemented. Thus, the semantics of a protocol should depend solely on the communications of the agents enacting it, not on their internal policies. Second, the semantics should capture the business

meanings of the messages, thereby avoiding operational constraints, and thus enabling the agents to deal better with exceptions and opportunities [Yolum and Singh, 2002]. Third, the semantics should be modular: an agent who enacts a protocol correctly may concurrently enact additional protocols. Fourth, designing engagements using protocols presupposes that we support engineering methodologies such as those based on stepwise refinement. We address the above criteria for protocols with an emphasis on their refinement.

We understand a protocol semantically in terms of exactly the set of runs (i.e., computations) that it allows. Following Mallya and Singh [2007], we posit that a putative subprotocol *refines* a putative superprotocol if and only if each run allowed by the subprotocol is also allowed by the superprotocol. In general, a subprotocol would include additional roles and actions: in determining refinement, we disregard those that do not feature in the superprotocol. Doing so facilitates modularity, enhanceability, and reuse of protocols.

Consider a simple protocol *Pay* consisting of two actions where a payer first commits to paying a payee, and later pays. Now consider a protocol *PayViaMM* where the payer first pays a middleman, who in turn pays the payee. Both *Pay* and *PayViaMM* send a payment from the payer to the payee. Even though *PayViaMM* involves an additional role (middleman) and *PayViaMM* uses different messages (two payment messages instead of one), we expect *PayViaMM* refines *Pay*, because *PayViaMM* makes a payment as *Pay* specifies. Similarly, we expect *PayViaCheck* and *PayViaCredit* also refine *Pay*. We imagine a service engagement design exercise where protocol designers begin by identifying the need for payment as *Pay*, then refine it to *PayViaMM*, and then to *PayViaCheck*. The designers may build or find an existing repository of protocols (analogous to taxonomies of business processes [Malone et al., 2003]). The question we address is how can protocols in such a repository be expressed so that their refinements can be rigorously verified.

5.1.1 Proton: Approach and Contributions

We formulate refinement in technical terms and show how to compute it via a tool called *Proton*. We specify a protocol declaratively in terms of (i) its roles, (ii) the guarded messages the roles exchange, and (iii) the meaning of each message as a set of actions on the public state of the roles, sometimes termed the *social state* [Baldoni et al., 2010a]. Commitments between roles are central to our approach [Singh, 1999]. Section 2.2 provides additional details. For now, suffice it to say that a state of a protocol is determined by what atomic propositions hold therein (some propositions specify the states of commitments).

We define the semantics of a protocol precisely in terms of the runs (i.e., sequences of actions) it allows. Informally, a *subprotocol* refines a *superprotocol* if and only if the latter allows all the runs the former allows. However, refinement is nontrivial because the protocols may involve different roles and messages, the messages may have different meanings, and the meanings may be at different

levels of abstraction. Hence, we define refinement only with respect to a mapping of meanings from the superprotocol to the subprotocol. For example, the payment in *Pay* maps to two payments in *PayViaMM*.

Our approach for verifying refinement takes three inputs: formal descriptions of a putative superprotocol and subprotocol, and a mapping between them. We reduce the protocol descriptions to their canonical forms, taking into account the mapping provided. We generate input to an existing model checker consisting of (i) a specification of a temporal logic model and (ii) temporal formulas whose truth in the model verifies refinement.

5.1.2 Contributions

Our main contributions are as follows. One, we offer the *first* approach that computes the refinement for protocols based on static analysis of protocol specifications. Two, we formulate a notion of the serial composition of commitments, which can have broader applications than this paper, e.g., in the treatment of commitments in coalitions.

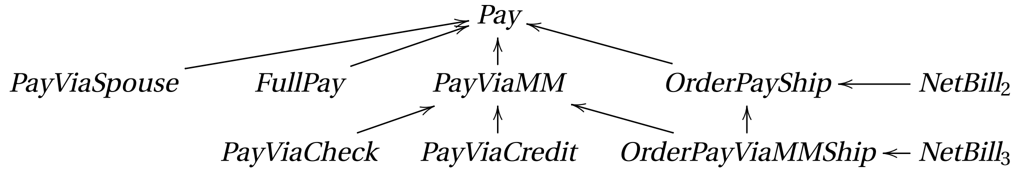


Figure 5.1: Refinements demonstrated by Proton (arrows point from subprotocols to superprotocols).

Further, we have implemented our approach in the Proton tool that overlays the well-known model checker MCMAS (http://www-lai.doc.ic.ac.uk/mcmas/). Figure 5.1 summarizes some protocol refinements that Proton verifies (under the obvious mappings) based on the above and other examples known from the literature.

5.1.3 Organization

Section 2.2 presented our syntax and semantics and briefly reviews commitments. Section 2.1 introduced our running examples for payment and order protocols. Section 5.2 formalizes our definitions of protocols, mappings between protocols, and protocol refinement. Section 5.3 describes how Proton generates input for the MCMAS model checker and the CTL formulas that must be satisfied for protocol refinement to hold. Section 5.4 pulls the previous sections together, illustrating how protocol *PayViaMM* refines, or fails to refine, protocol *Pay* under various mappings. Section 5.5 shows that

the algorithmic implementation in Section 5.3 is correct with respect to the theoretical framework of Section 5.2. Section 5.6 describes the related literature and important future directions.

5.1.4 Mapping Abstractions across Protocols

Since superprotocols represent higher-level abstractions than subprotocols, comparing protocols must address differences in abstraction level. To this end, we map elements (roles, propositions, and commitments) of a putative superprotocol to elements of a putative subprotocol. We map every super-element to an expression of sub-elements, but a subprotocol may contain sub-elements that do not correspond with any super-element.

Listing 5.1 Mapping M_1: *Pay* to *PayViaMM*

```
 1: map M1: Pay ↦ PayViaMM {
 2:     role
 3:         Payer ↦ {Payer};
 4:         Payee ↦ {Payee};
 5:     prop
 6:         promise ↦ promise;
 7:         pay ↦ payP ∧ payM;
 8:     commitment
 9:         C_pay ↦ C_payP ⊕ C_payM;      //requires C_pay ≤_M1 C_payP ⊕ C_payM
10: }
```

Consider mapping M_1 in Listing 5.1 from *Pay* to *PayViaMM*. Each super-role is mapped to a set of sub-roles. Line 3 maps the Payer super-role in *Pay* to the Payer sub-role in *PayViaMM*. Each super-proposition in *Pay* is mapped to a Boolean expression of sub-propositions in *PayViaMM*. Line 7 maps *pay* to the conjunction of *payP* and *payM*. Notice that *payP* and *payM* are messages sent by different roles in *PayViaMM*; thus even the simple Line 7 demonstrates the generality of our mapping approach. Line 9 maps super-commitment C_{pay} to the serial composition of sub-commitments C_{payP} and C_{payM}.

There can be multiple mappings between some protocol pairs. The Middleman role does not appear in mapping M_1. We can construct alternative mappings that group the Middleman into coalitions with different super-roles. Mapping M_2 in Listing 5.2 and Mapping M_3 in Listing 5.3 are each the same as M_1 except for their role mappings: M_2 groups the Middleman into a coalition with Payer and M_3 into a coalition with Payee. *PayViaMM* refines *Pay* under all three mappings M_1, M_2, and M_3.

We require each commitment to be explicit mapped. A commitment mapping must not violate the

Listing 5.2 Alternative Mapping M_2: *Pay* to *PayViaMM*

```
1: map M₂: Pay ↦ PayViaMM {
2:     role
3:         Payer ↦ {Payer, MM};
4:         Payee ↦ {Payee};
5:     ...
6: }
```

Listing 5.3 Alternative Mapping M_3: *Pay* to *PayViaMM*

```
1: map M₃: Pay ↦ PayViaMM {
2:     role
3:         Payer ↦ {Payer};
4:         Payee ↦ {Payee, MM};
5:     ...
6: }
```

role and proposition mappings, but that is not always sufficient to uniquely determine the commitment mapping. It is possible that a super-commitment can be mapped to multiple serial compositions that meet all constraints. In a hypothetical *PayViaTwoMM* protocol, where the payment can be made through either Middleman_1 or Middleman_2, the super-commitment $\mathsf{C}_1 = \mathsf{C}_{Payer,\text{Payee}}(promise, pay)$ can be mapped to either of two serial compositions (the protocol designer chooses between them based on other factors).

$$\mathsf{C}_1 \mapsto \mathsf{C}_{Payer,MM_1}(promise, payMM_1) \oplus \mathsf{C}_{MM_1,Payee}(payMM_1, pay)$$
$$\mathsf{C}_1 \mapsto \mathsf{C}_{Payer,MM_2}(promise, payMM_2) \oplus \mathsf{C}_{MM_2,Payee}(payMM_2, pay)$$

Mapping B_1 in Listing 5.4 shows a possible mapping between *Pay* and *PayViaMM*, similar to M_1

Listing 5.4 Nonrefining Mapping B_1: *Pay* to *PayViaMM*

```
1: map B₁: Pay ↦ PayViaMM {
2:     role
3:         Payer ↦ {Payer};
4:         Payee ↦ {Payee};
5:     prop
6:         promise ↦ promise;
7:         pay ↦ payP ∧ payM;
8:     commitment
9:         C_pay ↦ C_payM ⊕ C_payP;        //wrong order
10: }
```

except the serial composition in Line 9 combines the commitments in the wrong order. In Section 5.4, we show *PayViaMM* does *not* refine *Pay* under mapping B_1.

5.2 Formalizing Protocols and their Refinement

We assume a set of atomic propositions that describe the state of the world and states of relevant commitments. We define actions as atomic propositions (being made true) and commitment operations (being performed). Messages set propositions true, but not false.

Definition 5.2.1. *A* protocol *is a septuple* $\langle \mathscr{R}, \mathscr{M}, \mathscr{C}, \mathscr{A}, \mathscr{S}, \mathscr{S}^0, \mathscr{G} \rangle$ *corresponding respectively to (i) a set* $\mathscr{R}$ *of roles; (ii) a set* $\mathscr{M}$ *of message names; (iii) a set* $\mathscr{C}$ *of commitments; (iv) a set* $\mathscr{A}$ *of Boolean propositions and commitment states; (v) a set* $\mathscr{S}$ *of states,* $\mathscr{S} \subseteq 2^{\mathscr{M}}$ *such that if* $s \in \mathscr{S}$*,* $gs \in \mathscr{G}$*, and* $s \in gs.guard$ *then* $s \cup gs.msg \in \mathscr{S}$*; (vi) a set* $\mathscr{S}^0 \subseteq \mathscr{S}$ *of initial states; and (vii) a set* $\mathscr{G}$ *of guarded statements of the form* $\langle snd, rcv, guard, msg, actions \rangle$ *with* $snd, rcv \in \mathscr{R}$*,* $guard \subseteq \mathscr{S}$*,* $msg \in \mathscr{M}$*, and* $actions = \{a_i \in \mathscr{A}\} \cup \{Act_{\mathsf{C}}(\mathsf{C}_j) \in \mathscr{C})\} \cup \{nop\}$*. In addition, we impose the* no overlap *constraint:* $\forall gs_1, gs_2 \in \mathscr{G}$*, if* $gs_1.actions \cap gs_2.actions \neq \emptyset$ *then* $gs_1.guard \cap gs_2.guard = \emptyset$*.*

Each message corresponds to an atomic proposition recording whether the message has been sent. Each global state $s \in \mathscr{S}$ is a set of (the atomic propositions corresponding to) the messages that have been sent in that state (Item v in the definition). Each guarded statement $gs \in \mathscr{G}$ has a guard *gs.guard* which is a set of states, and a meaning *gs.actexp*—a conjunctive expression of actions. A message *msg* can be sent by the sender (*gs.snd*) to the receiver (*gs.rcv*) in state s only if $s \in gs.guard$. When m is sent, the action expression *gs.actexp* becomes true in the next state. The actions corresponding to different messages may be interleaved. The *no overlap* constraint ensures that if two or more super-actions contain the same sub-action, and both super-actions are enabled in a state, then the occurrence of the common sub-action in a sub-run is unambiguous as to which super-action it corresponds to, which recall is key to our notion of refinement.

5.2.1 Protocol Enactment

We introduce a *run*, a possible computation through our model, as a basis for our semantics. A run, notated π, is an alternating sequence of states and actions $\langle s_0, a_1, s_1, a_2, s_2, \ldots \rangle$ such that s_{i+1} results from performing a_{i+1} in s_i. The length of π is written $|\pi|$.

We can now express two key intuitions. First, the semantics of a protocol is simply the set of runs it allows. Underlying each run is a coarser *message enactment*: a sequence of states and messages where each message's guard is true in the state where the message occurs. Second, a protocol refines another if and only if the runs of the first are also runs of the second, with the proviso that the putative subprotocol may involve roles and actions that are absent in the putative superprotocol. To capture

the above, we need to relate protocols to models. Our approach generates a model from the putative subprotocol and then verifies (using suitable mapping) whether the putative superprotocol relates correctly with the subprotocol in that model, that is, whether the runs of the two protocols relate as explained above. Definition 5.2.2 specifies such a model.

Definition 5.2.2 (Proton Model)**.** *Let $P = \langle \mathscr{R}, \mathscr{M}, \mathscr{C}, \mathscr{A}, \mathscr{S}, \mathscr{S}^0, \mathscr{G} \rangle$ be a protocol. Then, the* Proton model for *P is $\mathscr{I} = \langle \Sigma, P, PV, L^i, Act^i, AP^i, t^i, G, G_0, F \rangle$ where (i) $\Sigma = \mathscr{R} \cup \{e\}$, with e being the environment, (ii) $P = \mathscr{A}$, (iii) $\forall p \in \mathscr{A}\ : PV(p) = \{s | p \in s\}$, (iv) $\forall i \in \mathscr{R} : L^i = \{l\}$ and $L^e = \prod_{m_i \in \mathscr{M}} m_i \times \prod_{\mathsf{C}_i \in \mathscr{C}} \mathsf{C}_i.state$, (v) $\forall i \in \mathscr{R} : Act^i = \{m | m.snd = i\} \cup \{nop\}$, and $Act^e = \{sched = r | r \in \mathscr{R}\}$. (vi) $\forall i \in \mathscr{R}, \forall s \in \mathscr{S}\ : AP^i(s) = \{m | sched = i \wedge m.snd = i \wedge s \in m.guard\}$, (vii) $\forall i \in \mathscr{R}\ :\ t^i(l) = l$, and $t^e = \prod_{m_i \in \mathscr{M}} t^{m_i} \times \prod_{\mathsf{C}_i \in \mathscr{C}} t^{\mathsf{C}_i}$, (viii) G is the set of all states reachable from G_0 by transition function T in $\mathscr{I}$, (ix) $G_0 = \mathscr{S}^0$, and (x) $F = \{\mathsf{C}_i.state \neq detached | \mathsf{C}_i \in \mathscr{C}\}$.*

Where $\times$ is binary cross-product and $\prod$ is set cross-product. The protocol's state is the cross-product of the state of each message and commitment. Since both messages and commitments involve multiple roles, each role has just a single state l and all state is in the environment Item (iv). Proton supports interleaved rather than concurrent actions with the environment scheduling one role at each step (Item v). Every role can perform the *nop* action (no-operation) at every step (Item v). t^{m_i} is the transition function that tracks the past occurrence of message m_i, and t^{C_i} is the transition function that tracks the commitment state of commitment C_i as defined by Figure 2.2 (Item vii).

Through a slight abuse of notation, for simplicity, we treat guards and actions as expressions in the following.

Definition 5.2.3 (Enactment)**.** *Let $P = \langle \mathscr{R}, \mathscr{M}, \mathscr{C}, \mathscr{A}, \mathscr{S}, \mathscr{S}^0, \mathscr{G} \rangle$ be a protocol and $\mathscr{I}$ be its Proton model. Then, an alternating sequence of states and messages $\langle h_0, m_1, h_1, m_2, h_2, \ldots \rangle$ is a* message enactment of *P if and only if $h_0 \in \mathscr{S}^0$ and ($\forall j \geq 0 : h_j \in \mathscr{S}, m_{j+1} \in \mathscr{M}\ : \mathscr{I}, h_j \models m_{j+1}.guard$ and $\mathscr{I}, h_{j+1} \models m_{j+1}.actexp$).*

A message enactment yields one or more runs with different interleavings of each message's actions. We define a function μ that maps each index in the message enactment to the index in the run where the corresponding message expression $m_j.actexp$ becomes true. Each message expression occurs in the same order in every run, and becomes true precisely at the state where its execution completes.

Definition 5.2.4 (Run)**.** *Let $P = \langle \mathscr{R}, \mathscr{M}, \mathscr{C}, \mathscr{A}, \mathscr{S}, \mathscr{S}^0, \mathscr{G} \rangle$ be a protocol and $\mathscr{I}$ be its Proton model. Then, an alternating sequence of states and actions $\langle s_0, a_1, s_1, a_2, s_2, \ldots \rangle$ is a* run of *P if and only if $s_0 \in \mathscr{S}^0$ and ($\forall j \geq 0 : s_j \in \mathscr{S}, a_{j+1} \in \mathscr{A}\ : \mathscr{I}, s_j \models a_{j+1}.guard$ and $\mathscr{I}, s_{j+1} \models a_{j+1}.actexp$).*

We say a run is *well defined* to emphasize that it satisfies the guard and action expression conditions above: that it is more than just an alternating sequence of states and actions. The empty run $\langle \emptyset \rangle$ is always well defined, since no agent is required to perform any action.

Definition 5.2.5 (Generated Runs). *Let $P = \langle \mathscr{R}, \mathscr{M}, \mathscr{C}, \mathscr{A}, \mathscr{S}, \mathscr{S}^0, \mathscr{G} \rangle$ be a Proton protocol and $\mathscr{I}_P$ its model. Then, a run $\pi = \langle s_0, a_1, s_1, \ldots \rangle$ is* generated by P *if and only if there exists a message enactment $\langle h_0, m_1, h_1, \ldots \rangle$, and there exists a strictly increasing function on the natural numbers $\mu : \mathbb{N} \mapsto \mathbb{N}$ such that $(\forall j \geq 0 : \mathscr{I}, s_{\mu(j)} \models m_{j+1}.guard$ and $\mathscr{I}, s_{\mu(j+1)} \models m_{j+1}.actexp)$.*

We write **runs**(P) for the set of all runs generated by protocol P in $\mathscr{I}_P$.

5.2.2 Protocol Refinement

Definition 5.2.6 (Mapping). *M maps protocol $P = \langle \mathscr{R}_P, \mathscr{M}_P, \mathscr{C}_P, \mathscr{A}_P, \mathscr{S}_P, \mathscr{S}_P^0, \mathscr{G}_P \rangle$ to protocol $Q = \langle \mathscr{R}_Q, \mathscr{M}_Q, \mathscr{C}_Q, \mathscr{A}_Q, \mathscr{S}_Q, \mathscr{S}_Q^0, \mathscr{G}_Q \rangle$ if and only if $M = \langle M_R, M_P, M_C \rangle$, where*

$$
\begin{aligned}
M_R &= \{\langle r, R \rangle \mid r \in \mathscr{R}_P, R \subseteq \mathscr{R}_Q\} \\
M_P &= \{\langle p, e \rangle \mid p \in \mathscr{A}_P, e \subseteq 2^{\mathscr{A}_Q}\} \\
M_C &= \{\langle \mathsf{C}, \oplus_i \mathsf{C}_i \rangle \mid \mathsf{C} \in \mathscr{C}_P, \mathsf{C}_i \in \mathscr{C}_Q, \mathsf{C} \leq_M \oplus_i \mathsf{C}_i\}
\end{aligned}
$$

When mapping actions on propositional expressions of propositions, we apply the following mappings

$$\textsc{isSet}(p \wedge q) \mapsto \textsc{isSet}(p) \wedge \textsc{isSet}(q) \tag{5.1}$$

$$\textsc{isSet}(p \vee q) \mapsto \textsc{isSet}(p) \vee \textsc{isSet}(q) \tag{5.2}$$

$$\textsc{isSet}(\neg p) \mapsto \neg\textsc{isSet}(p) \tag{5.3}$$

$$\textsc{set}(p \wedge q) \mapsto \textsc{set}(p) \wedge \textsc{set}(q) \tag{5.4}$$

$$\textsc{set}(p \vee q) \mapsto \textsc{set}(p) \vee \textsc{set}(q) \tag{5.5}$$

$$\textsc{set}(\neg p) \mapsto \neg\textsc{set}(p) \tag{5.6}$$

When mapping actions on commitment expressions, we apply the following mappings

$$\textsc{isActive}(\mathsf{C}_A \oplus \mathsf{C}_B) \mapsto \textsc{isActive}(\mathsf{C}_A) \wedge \textsc{isActive}(\mathsf{C}_B) \tag{5.7}$$

$$\textsc{create}(\mathsf{C}_A \oplus \mathsf{C}_B) \mapsto \textsc{create}(\mathsf{C}_A) \wedge \textsc{create}(\mathsf{C}_B) \tag{5.8}$$

$$\textsc{transfer}(\mathsf{C}_A \oplus \mathsf{C}_B) \mapsto \textsc{transfer}(\mathsf{C}_A) \vee \textsc{transfer}(\mathsf{C}_B) \tag{5.9}$$

$$\textsc{release}(\mathsf{C}_A \oplus \mathsf{C}_B) \mapsto \textsc{release}(\mathsf{C}_A) \vee \textsc{release}(\mathsf{C}_B) \tag{5.10}$$

$$\textsc{cancel}(\mathsf{C}_A \oplus \mathsf{C}_B) \mapsto \textsc{cancel}(\mathsf{C}_A) \vee \textsc{cancel}(\mathsf{C}_B) \tag{5.11}$$

$$\textsc{reset}(\mathsf{C}_A \oplus \mathsf{C}_B) \mapsto \textsc{reset}(\mathsf{C}_A) \vee \textsc{reset}(\mathsf{C}_B) \tag{5.12}$$

Informally, a run π_Q *embeds* a run π_P if all of π_P lies within π_Q. In effect, π_Q does everything that π_P does, and possibly more: as Mallya and Singh [2007] propose, a protocol Q refines a protocol P if

and only if every run of Q embeds some run of P. This captures the intuition that any computation (run) allowed by Q is allowed by P as well.

Consider the mapping from *Pay* to *OrderPayShip*. In protocol *Pay*, *promiseMsg* means {CREATE(C_{pay})} and *payMsg* means {*pay*}. In protocol *OrderPayShip*, *orderMsg* means {CREATE(C_{pay})} and *payMsg* means {*pay*}. Therefore, *promiseMsg* and *payMsg* in *Pay* mean the same, respectively, as *orderMsg* and *payMsg* in *OrderPayShip*.

$$\begin{aligned} \textit{Pay} &\mapsto \textit{OrderPayShip} \\ \textit{promiseMsg} &\mapsto \textit{orderMsg} \\ \textit{payMsg} &\mapsto \textit{payMsg} \end{aligned}$$

Pay has two message enactments: ⟨⟩ (the empty enactment) and ⟨promiseMsg, payMsg⟩. *OrderPayShip* has five message enactments, which embed *Pay*'s runs as follows.

⟨⟩	: ⟨⟩
⟨⟩	: ⟨*reqQuoteMsg*⟩
⟨⟩	: ⟨*reqQuoteMsg*, *sendQuoteMsg*⟩
⟨promiseMsg, payMsg⟩	: ⟨*reqQuoteMsg*, *sendQuoteMsg*, orderMsg, payMsg, *shipMsg*⟩
⟨promiseMsg, payMsg⟩	: ⟨*reqQuoteMsg*, *sendQuoteMsg*, orderMsg, *shipMsg*, payMsg⟩

We define a *mapped run* where each sub-state s is enriched to a state $M(s)$ by including values for all super-propositions and super-commitments. We now compare enriched sub-states $M(s)$ in mapped sub-runs with super-states in super-runs. Below, we write $exp\langle\langle x \mapsto y\rangle\rangle$ to mean the expression resulting from the uniform substitution of symbol x by expression y in *exp*.

Definition 5.2.7 (Mapped Run). *Let* $\pi = \langle s_0, a_1, s_1, \ldots\rangle$ *be a run and* $M = \langle M_R, M_P, M_C\rangle$ *be a protocol mapping. Then the* M-map of π, $M(\pi) = \langle M(s_0), a_1, M(s_1), \ldots\rangle$ *is a run where for all* s, $M(s) \supseteq s$ *and* $M(s)$ *is the minimal set for which the following conditions hold:*

- *(Propositions) if* $\langle m, E\rangle \in M_P$ *and* $s \models E$, *then* $m \in M(s)$.
- *(Commitments) if* $\langle \mathsf{C}, \bigoplus_i \mathsf{C}_i\rangle \in M_C$ *and* $\forall i : s \models \mathsf{C}_i.state$, *then* $\mathsf{C}.state \in M(s)$ *where* $\mathsf{C}.state = min(\mathsf{C}_i.state)$.

Continuing with the above discussion, we map each sub-run and verify that it embeds some super-run. The following definition captures the intuition that the embedding sub-run steps through each of the states of the embedded super-run, but may potentially include additional states. We ignore the transitions in each run.

To simplify the notation, we also introduce a projected mapping function $\widehat{M}(q) = M(q) \cap \mathscr{A}_P$ that is the set of just the propositions and states in a (super-)protocol P.

Definition 5.2.8 (Embedding)**.** *Let P and Q be two protocols. A run $\pi_Q = \langle q_0, \cdot, q_1, \ldots \rangle \in$ **runs**(Q)* embeds *a run $\pi_P = \langle p_0, \cdot, p_1, \ldots \rangle \in$ **runs**(P), written **emb**(π_Q, π_P), if and only if there exists a strictly increasing function on natural numbers $\tau : \mathbb{N} \mapsto \mathbb{N}$ such that $(\forall i : 0 \leq i \leq |\pi_P| : p_i = \widehat{M}(q_{\tau(i)})$ and $(\forall j : \tau(i) \leq j < \tau(i+1) : \widehat{M}(q_{\tau(i)}) = \widehat{M}(q_j))$, where $\widehat{M}(q) = M(q_j) \cap \mathscr{A}_P$.*

Function τ maps from indices of π_P to indices of π_Q, and the conditions ensure every time $\widehat{M}(q_j)$ changes, the new value matches the next p_i.

Now we can define refinement in purely semantic terms that capture our intuition that each mapped sub-run must embed some super-run. Notice that this definition implicitly uses Proton models $\mathscr{I}_P$ and $\mathscr{I}_Q$ respectively for P and Q.

Definition 5.2.9 (Refinement)**.** *Let P and Q be two protocols, and M a mapping from P to Q. Then Q refines P under M if and only if $(\forall \pi_Q \in$ **runs**$(Q) : (\exists \pi_P \in$ **runs**$(P) :$ **emb**$(M(\pi_Q), \pi_P)))$.*

5.3 Verifying Protocol Refinement

Figure 5.2 shows the high-level process flow for verifying protocol refinement. The Proton preprocessor reads the subprotocol, superprotocol, and mapping specifications and constructs (as Section 5.4 details) the input for the MCMAS model checker in the Interpreted Systems Programming Language (ISPL) [Lomuscio et al., 2009].

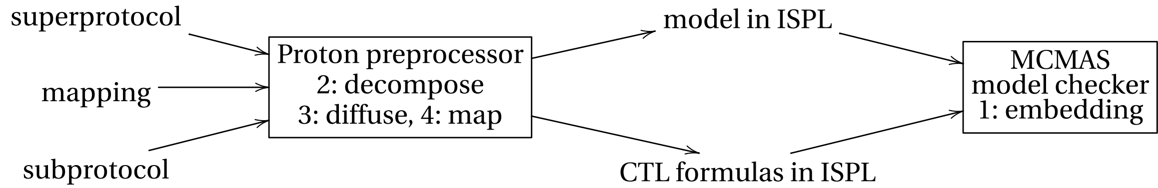


Figure 5.2: Proton process flow. Activities are shown in boxes and specification documents are not. Transformations and comparisons are numbered 1 to 4 in sequence.

The input to MCMAS is a set of guarded statements for each role. MCMAS internally generates a state transition system such as that shown in Figure 5.3. The system starts in initial state s_0. Action *requestQuote* transitions to state s_1, action *sendQuote* transitions to state s_2, and so on. There is an edge for every action enabled in a state.

The Proton preprocessor generates an interpreted system model for the subprotocol. There is one ISPL agent definition for each sub-role, and the state of all sub-elements (propositions and

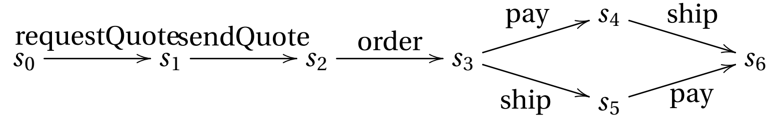


Figure 5.3: A schematic of some possible enactments of the *OrderPayShip* protocol.

commitments) are expressed as model state variables. The model checker simulates the subprotocol's actions. Because each super-element is mapped to an expression of sub-elements, the state of every super-element can be inferred from the subprotocol's state. As Section 5.3.5 shows, protocol refinement conditions are expressed as CTL formulas. If all these CTL formulas are true, the subprotocol refines the superprotocol.

5.3.1 Intuition: Decomposition

A message can mean multiple things. To better understand and characterize a message, we decompose each *message* into its meaning as a set of primitive, well-defined *actions*. The meaning of a message is then the conjunction of all its constituent actions.

An action is either a Boolean proposition or a commitment operation. A propositional action sets the value of the proposition to true. We do not support setting propositions to false. Commitment actions are the operations CREATE, TRANSFER, RELEASE, and CANCEL that change the state of a commitment.

We replace all message terms with a conjunction of their actions throughout a protocol, *decomposing* protocol messages, converting from a "protocol of guarded messages" to a "protocol of guarded actions." Each *msg* term is replaced by a conjunction of its actions in both the guard and the action expression of every guarded statement.

$$\mathbf{means}([guard]\ msg\ means\{act_i\}) \quad \Rightarrow \quad [guard\langle\langle msg \mapsto \bigwedge_i act_i \rangle\rangle]\ \bigwedge_i act_i$$

5.3.2 Intuition: Diffusion

The result of decomposition is a set of guarded action expressions, but the model checker executes actions, not action expressions. Therefore, each guarded action expression must be converted to an equivalent set of guarded actions. However, computing the equivalent guarded actions is nontrivial. For example, consider the guarded action expression

$$[guard]\ payP \wedge payM$$

A naïve approach would be to apply the guard to each action separately: [*guard*] *payP* and [*guard*] *payM*. Doing so would be overly restrictive because neither payP nor payM can occur before the guard becomes true. Greater flexibility is needed.

Given a guarded conjunction of actions [*guard*] $a_1 \wedge a_2 \wedge \cdots \wedge a_n$, the action expression becomes true when the *last* (in time) of the a_is becomes true. Given a guarded disjunction of actions [*guard*] $a_1 \vee a_2 \vee \cdots \vee a_n$, the action expression becomes true when the *first* (in time) of the a_is becomes true. We minimally constrain when the a_i become true so the overall expression becomes true at exactly the same point, relative to the other actions, in both the super-runs and the sub-runs. For conjunctions, all the a_i, except the *last*, can move to any *earlier* point in time. For disjunctions, all the a_i, except the *first*, can move to any *later* point in time. The a_i can even move all the way to the run's beginning (conjunction) or end (disjunction). Decomposition (Section 5.3.1) generates guarded action expressions with conjunctions; abstraction mappings (Section 5.3.3) can generate guarded action expressions with both conjunctions and disjunctions.

The recursive *diffusion* function **dif** transforms a guarded action expression to a set of guarded actions.

$$\mathbf{dif}([guard]\ \bigvee_i exp_i) \Rightarrow \{\mathbf{dif}([guard]\ exp_i)\} \tag{5.13}$$

$$\mathbf{dif}([guard]\ \bigwedge_i exp_i) \Rightarrow \{\mathbf{dif}([guard \vee \bigvee_{j \neq i} \neg exp_j]\ exp_i)\} \tag{5.14}$$

$$\mathbf{dif}([guard]\ act) \Rightarrow [guard]\ act \tag{5.15}$$

where *guard* and exp_i are Boolean expressions of actions. Diffusion transforms the guarded expression example above to

$$[guard \vee \neg payM]\ payP$$
$$[guard \vee \neg payP]\ payM$$

Both actions can still occur after the guard becomes true, so it allows at least all of the runs allowed in the naïve approach. Additionally, the first action to fire can fire at any time. If *payP* fires before *payM*, then ¬*payM* is true when *payP* fires, so *payP* can fire at any time. Diffusion thus covers possibilities that the naïve approach omits.

Diffusion can generate multiple guarded statements for the same action, but MCMAS requires a single guarded statement for each action. Therefore, we introduce the *collection* function **col** that converts a set of guarded statements back to canonical form, where there is a single guarded statement for each action. Consider each individual, input action. Here, **col** collects potentially multiple guarded statements for that action in its input, and generates a single guarded statement for that action in its output. The output guard for the action is the disjunction of all the input guards. If an action

appears in only one guarded statement in the input, that guarded statement appears unmodified in the output.

$$\mathbf{col}(\{[guard_i]\ act_i\}) \quad \Rightarrow \quad \{[\bigvee_j guard_j]\ act_i \quad | act_j = act_i\}$$

Consider a partial protocol containing these two guarded statements. Since the messages overlap on action *ship*, condition *freeCoupon* ensures the no overlap constraint of Definition 5.2.1.

$$[orderMsg \wedge \neg freeCoupon]\ paidShipMsg \ \text{means}\ \{bill, ship\};$$
$$[orderMsg \wedge freeCoupon]\ freeShipMsg \ \text{means}\ \{ship\};$$

Diffusion transforms those to the following three guarded statements.

$$[(orderMsg \wedge \neg freeCoupon) \vee \neg ship]\ bill$$
$$[(orderMsg \wedge \neg freeCoupon) \vee \neg bill]\ ship$$
$$[orderMsg \wedge freeCoupon]\ ship$$

Collection merges the two statements for *ship*, giving these two, final guarded statements.

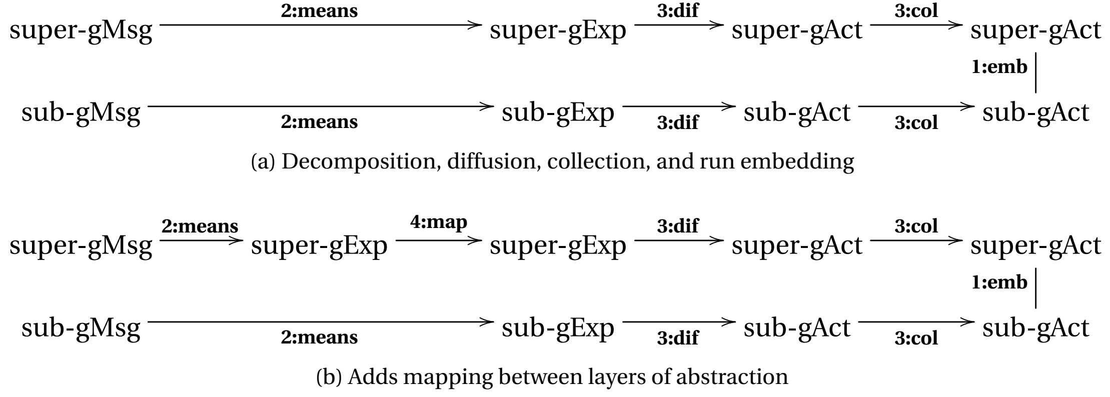
$$[(orderMsg \wedge \neg freeCoupon) \vee \neg ship]\ bill$$
$$[(orderMsg \vee \neg bill]\ ship$$

Figure 5.4a schematically shows how the foregoing developments of decomposition, diffusion, collection, and embedding combine together to check whether one protocol refines another. However, by themselves, they do not address the fact that different protocols can be written at different layers of abstraction.

5.3.3 Accommodating Abstraction Mapping

Since superprotocols represent higher-level abstractions than subprotocols, comparing protocols must address differences in levels of abstraction. There is often no one-to-one correspondence between super-elements and sub-elements. Protocol elements (roles, propositions, and commitments) must be mapped between the two protocols to compare them.

An important type of abstraction difference is the introduction of intermediaries or middlemen in lower-level abstractions. Whereas two super-roles may communicate directly with each other using a single message in a high-level protocol, there is a natural tendency for message communication to pass through multiple, intermediary sub-roles as that protocol is refined to lower-level abstractions. Protocol refinement must allow super-elements to span intermediaries. One super-proposition could



(a) Decomposition, diffusion, collection, and run embedding

(b) Adds mapping between layers of abstraction

Figure 5.4: Protocol refinement defined as transformations (horizontal lines) and comparisons (vertical lines) between protocols of guarded messages (gMsg), guarded action expressions (gExp), and guarded actions (gAct). In each subfigure, the top row refers to a superprotocol and the bottom row to a subprotocol.

map to an expression of multiple sub-propositions, each controlled by different sub-roles (intermediaries), and one super-commitment could be fulfilled through multiple sub-commitments and their intermediate sub-debtors.

We say one protocol refines another protocol *under a given mapping*, because mapping functions are an essential element for protocol refinement, and must be an explicit input. A subprotocol might refine a superprotocol under one mapping, but not under a different mapping. Our approach does not determine whether it is impossible for one protocol to refine another protocol under any possible mapping.

A mapping expresses how terms in a putative superprotocol map to expressions in a putative subprotocol. The *mapping* function **map** converts guarded action expressions written with high-level abstractions x_i in a putative superprotocol, to expressions e_i of low-level terms in a putative subprotocol. (Below, $\langle\langle x_i \mapsto e_i \rangle\rangle$ is a mapping assertion.)

$$\textbf{map}([\mathit{guard}]\ \mathit{exp}) \quad \Rightarrow \quad [\mathit{guard}\langle\langle x_i \mapsto e_i \rangle\rangle]\ \mathit{exp}\langle\langle x_i \mapsto e_i \rangle\rangle$$

5.3.4 Verifying Refinement: Summary

Figure 5.4b schematically shows the transformations and comparison required to demonstrate protocol refinement. In both subfigures, horizontal lines show the transformations of a single protocol: decomposition (**means**), mapping (**map**), diffusion (**dif**), and collection (**col**). Vertical lines show the comparison between two protocols: run embedding (**emb**). The nodes in the figure show how guarded messages (gMsg) are transformed first to guarded action expressions (gExp), and then to guarded actions (gAct). In each subfigure, the top row refers to a superprotocol and the bottom row refers to a

subprotocol.

5.3.5 Generating CTL Formulas for Verification

MCMAS checks whether an interpreted system model satisfies specified CTL formulas. In this section, we describe how Proton expresses conditions for commitment resolution, overlapping messages, serial composition, and commitment covering as CTL formulas. All such formulas must be satisfied for protocol refinement to hold.

Verify Run Embedding by Checking Guards

Protocol comparison is fundamentally based on run embedding. Run embedding means, at every state, if the subprotocol can perform an action then the superprotocol must also be able to perform that action. That is, when an action's sub-guard is true, its super-guard must also be true. Since run embedding ignores actions not in the superprotocol, Proton generates CTL formulas for all actions that result from mapping all super-actions ($\forall a \in M(\mathscr{A}_{super})$):

$$\mathbf{AG}(a.\textit{sub-guard} \rightarrow a.\textit{super-guard}) \tag{5.16}$$

Verify that Messages Do Not Overlap

So that every action a in a sub-run can be uniquely associated with a message, we verify the no overlap constraint of Definition 5.2.1. For every pair of guarded statements gs_1 and gs_2 that share a common action meaning a,

$$\mathbf{AG}(\neg(gs_1.guard \wedge gs_2.guard)) \tag{5.17}$$

Verify that Detached Commitments Eventually Resolve

We require each detached commitment must eventually resolve in every correct protocol enactment. We employ model checker fairness constraints (expressions that must be true infinitely often on any run) to eliminate sub-runs in which the sub-roles fail to act properly and resolve their detached commitments. Doing so restricts our verification to correct enactments of the given protocols, thus avoiding false negatives due to incorrect enactments.

$$\textbf{Fairness } \mathsf{C}_{sub}.state \neq detached \tag{5.18}$$

The states of super-commitments can be inferred from the states of sub-commitments.

Verify Commitment Covering

The truth or falsity of a statement in an unreachable state has no bearing on the enactment of a protocol, so we can replace $a \models b$ statements by the CTL formula $\mathbf{AG}(a \rightarrow b)$. Doing so enables us to use the model checker to verify commitment covering, which would otherwise need to be handled separately, as indeed it was in a previous version of Proton.

Verifying one commitment covers another under map M, $\mathsf{C}_W \leq_M \mathsf{C}_S$, is done in two parts. First, the preprocessor verifies the debtor and creditor conditions (Equations 3.9–3.10). Second, the model checker verifies the antecedent and consequent conditions (Equations 3.11–3.12) hold in all (reachable) states with the CTL formulas

$$\mathbf{AG}(M(\mathsf{C}_W.ant) \rightarrow \mathsf{C}_S.ant) \tag{5.19}$$

$$\mathbf{AG}(\mathsf{C}_S.csq \rightarrow M(\mathsf{C}_W.csq)) \tag{5.20}$$

Verify Serial Compositions are Well Defined

For serial compositions $\mathsf{C}_\oplus = \mathsf{C}_A \oplus \mathsf{C}_B$, the model checker verifies the well-definedness condition holds in all (reachable) states on all paths with the CTL formula

$$\mathbf{AG}(\mathsf{C}_A.csq \rightarrow \mathsf{C}_B.ant) \tag{5.21}$$

5.4 Tooling, Detailed Examples, and Experimental Results

In this section, we pull together the many elements: commitments, serial composition of commitments, and commitment covering; the example payment and order protocols, and various mappings between them; and the formal definitions. We concretely demonstrate how *PayViaMM* refines, or fails to refine, *Pay* under various mappings.

Proton verifies protocol refinement using the process flow as shown in Figure 5.2 and the pseudocode for **refines**(super, map, sub) shown in Listing 5.5. The inputs P and Q are protocols, which in our syntax are in terms of guarded messages. The first lines of the algorithm transform these into protocols expressed in terms of guarded actions. Proton generates an interpreted system model from the guarded actions of the subprotocol. There is one MCMAS agent definition for each sub-role, and the state of the sub-elements (propositions and commitments) are MCMAS state variables or MCMAS evaluation expressions. The MCMAS model checker then simulates the subprotocol's actions. Because each super-element is mapped to an expression of sub-elements, the superprotocol's state can be inferred from the subprotocol's state. Refinement requires the model of the subprotocol to satisfy a

Listing 5.5 Calculate **refines**(P, M, Q)

1: $Q_{gMsg} = Q$ ▷ Input Q is a protocol of guarded messages
2: $P_{gMsg} = P$ ▷ Input P is a protocol of guarded messages
3: $Q_{gAct} = \mathbf{col}(\mathbf{dif}(\mathbf{means}(Q_{gMsg})))$ ▷ protocol of guarded sub-actions
4: $P_{gAct} = \mathbf{col}(\mathbf{dif}(\mathbf{map}_M(\mathbf{means}(P_{gMsg}))))$ ▷ protocol of guarded super-actions
5: model = genModel(Q_{gAct}) ▷ generate ISPL model
6: **for all** $act_P \in P_{gAct}.actions$ **do** ▷ For all super-actions
7: genCTL($\mathbf{AG}(act_P.sub\text{-}guard \rightarrow act_P.super\text{-}guard)$)
8: **end for**
9: **for all** $\mathsf{C}_Q \in Q_{gAct}.\mathscr{C}$ **do** ▷ For all sub-commitments
10: genFairness($\mathsf{C}_Q.state \neq detached$)
11: **end for**
12: **for all** $\mathsf{C}_P \in P_{gAct}.\mathscr{C}$ **do** ▷ For all super-commitments
13: $\mathsf{C}_Q = \mathsf{C}_P.coveringCommitment$
14: genCTL($\mathbf{AG}(M(\mathsf{C}_P.ant) \rightarrow \mathsf{C}_Q.ant)$)
15: genCTL($\mathbf{AG}(\mathsf{C}_Q.csq \rightarrow M(\mathsf{C}_P.csq))$)
16: **end for**
17: **for all** $\mathsf{C}_A \oplus \mathsf{C}_B$ **do** ▷ For all serial compositions
18: genCTL($\mathbf{AG}(\mathsf{C}_A.csq \rightarrow \mathsf{C}_B.ant)$)
19: **end for**
20: **for all** overlapping guarded statement pairs gs_1 and gs_2 in P and Q **do**
21: genCTL($\mathbf{AG}(\neg(gs_1.guard \wedge gs_2.guard))$)
22: **end for**
23: ctl = all generated CTL formulas
24: **return** MCMAS(model, ctl) ▷ Are all CTL formulas satisfiable?

set of CTL formulas. If all formulas are true, the subprotocol refines the superprotocol.

$$\mathsf{C}_{pay} \quad \mapsto \quad \mathsf{C}_{payP} \oplus \mathsf{C}_{payM} \qquad \text{if } \mathsf{C}_{pay} \leq_{M_1} \mathsf{C}_{payP} \oplus \mathsf{C}_{payM}$$

Payer $\xrightarrow{\mathsf{C}_{pay}}$ Payee
promise → pay
$\geq_{M_1}$
{Payer, MM} $\xrightarrow{\mathsf{C}_{payP}\oplus\mathsf{C}_{payM}}$ {Payee, Payer}
promise → payP ∧ payM
$\oplus$
Payer $\xrightarrow{\mathsf{C}_{payP}}$ Payee MM $\xrightarrow{\mathsf{C}_{payM}}$ Payer
promise → payP payP → payM

We now check whether *PayViaMM* refines *Pay* under map M_1. Using the commitment diagrams from Section 3.2.3, this diagram demonstrates commitment C_{pay} from *Pay* is covered by the serial composition of C_{payP} and C_{payM} from *PayViaMM* under mapping M_1 in Listing 5.1 on page 49. The bottom-left arrow states that if promise becomes true, Payer commits to making payP true. The bottom-right arrow states that if payP becomes true, MiddleMan commits to making payM true. In the serial composition (middle arrow), if promise becomes true, then Payer and MiddleMan (severally) commit to making both payP and payM true. The well-definedness condition ensures that the discharge of the first commitment entails the antecedent of the second commitment, thus detaching it. The Payee and Payer are creditors of the input commitments, and are thus creditors of the serial composition. The serial composition covers the commitment in *Pay* (top arrow).

Proton generates the following six CTL formulas to verify *PayViaMM* refines *Pay* under M_1. Equation 5.16, which verifies whether sub-guards imply super-guards, for actions *promise*, pay_P, and pay_M, generates Equations 5.22–5.24, respectively. Equation 5.25 verifies that $\mathsf{C}_{\mathrm{payP}} \oplus \mathsf{C}_{\mathrm{payM}}$ is valid. Equations 3.11–3.12, which verify the antecedent and consequent conditions of $\mathsf{C}_{\mathrm{payP}} \oplus \mathsf{C}_{\mathrm{payM}}$ covers $\mathsf{C}_{\mathrm{pay}}$, generates Equation 5.26–5.27, respectively (the debtor and creditor conditions in Equations 3.9–3.10 are checked directly by the Proton preprocessor, not by MCMAS).

$$\mathbf{AG}(\top \rightarrow \top) \tag{5.22}$$

$$\mathbf{AG}(promise \wedge \textsc{create}(\mathsf{C}_{\mathrm{payP}}) \wedge \textsc{create}(\mathsf{C}_{\mathrm{payM}}) \rightarrow (promise \wedge \textsc{create}(\mathsf{C}_{\mathrm{payP}}) \wedge \textsc{create}(\mathsf{C}_{\mathrm{payM}})) \vee \neg pay_M) \tag{5.23}$$

$$\mathbf{AG}(\top \rightarrow (promise \wedge \textsc{create}(\mathsf{C}_{\mathrm{payP}}) \wedge \textsc{create}(\mathsf{C}_{\mathrm{payM}})) \vee \neg pay_P) \tag{5.24}$$

$$\mathbf{AG}(pay_P \rightarrow pay_P) \tag{5.25}$$

$$\mathbf{AG}(promise \rightarrow promise) \tag{5.26}$$

$$\mathbf{AG}(pay_P \wedge pay_M \rightarrow pay_P \wedge pay_M) \tag{5.27}$$

All of the above formulas are obviously true, except Equation 5.24, which can be rewritten $\mathbf{AG}(pay_P \rightarrow$

$promise \wedge \textsc{create}(\mathsf{C}_{\mathrm{payP}}) \wedge \textsc{create}(\mathsf{C}_{\mathrm{payM}})$). It is true because the progression of the model, controlled by message guards, ensures pay_P becomes true only after *promiseMsg*. MCMAS verifies each generated CTL formula holds in the model, so *PayViaMM* refines *Pay* under map M_1.

Proton generates exactly the same input to MCMAS when checking whether *PayViaMM* refines *Pay* under map M_2 or map M_3, because the subprotocol models are derived from exactly the same *PayViaMM* protocol, the superprotocol *Pay* contains exactly the same propositional and commitment super-elements, and exactly the same CTL conditions must be checked. Therefore, *PayViaMM* refines *Pay* under both maps M_2 and M_3.

$$\mathbf{AG}(\top \rightarrow \top) \tag{5.28}$$

$$\mathbf{AG}(promise \wedge \textsc{create}(\mathsf{C}_{\mathrm{payP}}) \wedge \textsc{create}(\mathsf{C}_{\mathrm{payM}}) \rightarrow (promise \wedge \textsc{create}(\mathsf{C}_{\mathrm{payP}}) \wedge \textsc{create}(\mathsf{C}_{\mathrm{payM}})) \vee \neg pay_M) \tag{5.29}$$

$$\mathbf{AG}(\top \rightarrow (promise \wedge \textsc{create}(\mathsf{C}_{\mathrm{payP}}) \wedge \textsc{create}(\mathsf{C}_{\mathrm{payM}})) \vee \neg pay_P) \tag{5.30}$$

$$\mathbf{AG}(pay_M \rightarrow promise) \tag{5.31}$$

$$\mathbf{AG}(promise \rightarrow pay_P) \tag{5.32}$$

$$\mathbf{AG}(promise \wedge pay_P \wedge pay_M \rightarrow pay_P \wedge pay_M) \tag{5.33}$$

Proton correctly reports failures. Proton generates these CTL formulas when checking whether *PayViaMM* refines *Pay* under mapping B_1 in Listing 5.4 on page 50. Recall that B_1 maps the super-commitment to a serial composition in the wrong order. Equations 5.28–5.30 are the same as Equations 5.22–5.24, and all hold in the model. Equations 5.33 obviously holds. But Equation 5.31 does not hold because pay_M has a true guard in Listing 3.3 on page 27, so the Middleman can send pay_M at any time, even before *promise*. That leaves Equation 5.32, which comes from the antecedent of $\mathsf{C}_{\mathrm{pay}}$ which is *promise* and the antecedent of $\mathsf{C}_{\mathrm{payM}} \oplus \mathsf{C}_{\mathrm{payP}}$ which is pay_P. In the states between the Payer promising and the Payer actually paying, the formula does not hold, meaning $\mathsf{C}_{\mathrm{pay}}$ can become detached without $\mathsf{C}_{\mathrm{payM}} \oplus \mathsf{C}_{\mathrm{payP}}$ also becoming detached. The result is $\mathsf{C}_{\mathrm{pay}}$ is not covered by $\mathsf{C}_{\mathrm{payM}} \oplus \mathsf{C}_{\mathrm{payP}}$. MCMAS correctly reports these two formulas are false in the model, and *PayViaMM* does *not* refine *Pay* under mapping B_1.

As in Mallya and Singh's [2007] approach, an interesting consequence of our treatment of refinement is that aggregation functions like refinement. For example, consider a protocol *OrderPayShip*. Because all enactments of *OrderPayShip* necessarily include an enactment of *Pay*, *OrderPayShip* is naturally a refinement of *Pay*. The foregoing coheres with the notion of subtype in object-oriented programming.

Proton verified all the refinements shown in Figure 5.1 and Table 5.1. The first three columns are the superprotocol, subprotocol, and mapping, respectively. *OrderPayShip* is identical to the first

Table 5.1: Information about each demonstrated Refinement. The columns are: the name of the superprotocol; the name of the subprotocol; the name of the map; whether subprotocol refines superprotocol under map; $\oplus$ is the number of serial compositions; *covers* is the number of commitment covering checks; *formulas* is the number of CTL formulas verified by the model checker; and *time* is the elapsed time (in seconds) for refinement verification.

Superprotocol	**Subprotocol**	**Map**	**refines**	$\oplus$	**cover**	**formulas**	**time**
Pay	*PayViaSpouse*	M_1	*Yes*	0	1	4	0.7
Pay	*FullPay*	M_1	*Yes*	0	1	4	0.8
Pay	*PayViaMM*	M_1	*Yes*	1	1	6	0.7
Pay	*PayViaMM*	M_2	*Yes*	1	1	6	0.7
Pay	*PayViaMM*	M_3	*Yes*	1	1	6	0.6
Pay	*PayViaMM*	B_1	*No*	1	1	6	0.7
Pay	*PayViaCheck*	M_1	*Yes*	2	1	7	1.1
Pay	*PayViaCredit*	M_1	*Yes*	2	1	9	3.6
Pay	*OrderPayShip*	M_1	*Yes*	1	1	5	1.0
Pay	*OrderPayViaMMShip*	M_1	*Yes*	2	1	7	1.2
PayViaMM	*PayViaCheck*	M_1	*Yes*	3	2	12	1.7
PayViaMM	*PayViaCredit*	M_1	*Yes*	3	2	12	1.7
PayViaMM	*OrderPayViaMMShip*	M_1	*Yes*	0	2	7	1.2
OrderPayShip	*OrderPayViaMMShip*	M_1	*Yes*	1	2	11	1.4
OrderPayShip	$NetBill_2$	M_1	*Yes*	0	2	9	1.2
OrderPayViaMMShip	$NetBill_3$	M_1	*Yes*	0	3	12	1.8

NetBill scenario described by Mallya and Singh [2007]. NetBill_2 and NetBill_3 are scenarios 2 and 3 in the same paper. *PayBySpouse* is a new, simple, payment protocol where one person promises and then his or her spouse pays. *FullPay* is similar to *Pay*, but exercises all the commitment operations: CREATE, TRANSFER, RELEASE, and CANCEL.

Each superprotocol-subprotocol pair has a unique set of mapping names, so M_1 for *Pay* and *PayViaMM* is a different mapping than M_1 for *Pay* and *PayViaCheck*. Mappings M_1, M_2, M_3, and B_1 for *Pay* and *PayViaMM* are shown, respectively, in Listings 5.1, 5.2, 5.3, and 5.4 on pages 50–50. The *refines* column is whether subprotocol refines superprotocol under map; the $\oplus$ column is the number of serial compositions in the mapping; the *covers* column is the number of commitment covering checks; the *formulas* column is the number of CTL formulas verified by the model checker; and the *time* column is the elapsed time in seconds for the complete refinement verification, including Proton preprocessing and MCMAS model checking.

5.5 Correctness of Our Refinement Verification Method

We now establish the correctness of the verification method of Section 5.3 with respect to the formal definitions of Section 5.2. Theorem 5.5.1 is our main correctness result: it ties the algorithm of Listing 5.5 with Definition 5.2.9. An interesting subtlety is that whereas Definition 5.2.9 "maps up," enriching the states in sub-runs, the algorithm "maps down," expanding expressions and ignoring elements not in $\mathscr{A}_P$.

Theorem 5.5.1. *Let P and Q be two protocols, and let M be a mapping from P to Q. Then,* $\textbf{refines}(P, M, Q)$ *returns true if and only Q refines P under M.*

Proof. Let $\mathscr{I}_Q$ be the Proton model generated from Q. Define mappings $M_Q = \textbf{means}(Q)$ and $M_P = M(\textbf{means}(P))$.

$\Rightarrow$ Assume **refines**(P,M,Q). Then all of the CTL formulas in Listing 5.5 are true. Because of Line 7, all sub-guards imply their super-guards. Because of the fairness condition at Line 10, all detached commitments eventually resolve. Because of Lines 14–15, all commitment coverings are valid. Because of Line 18, all serial compositions are valid. Because of Line 21, no guarded statement pairs overlap.

Let π_Q be a run in **runs**(Q). By Theorem B.0.7, there is a run $\pi'_Q \in \textbf{runs}(\textbf{col}(\textbf{dif}(M_Q(Q))))$. By Theorem B.0.2, there exists a run $\pi'_P \in \textbf{runs}(\textbf{col}(\textbf{dif}(M_P(P))))$ And, by Theorem B.0.7, there exists a run $\pi_P \in \textbf{runs}(P)$. Therefore, for every π_Q there is a π_P and Q refines P under M by Definition 5.2.9.

$\Leftarrow$ Assume Q refines P under M. For any $\pi_Q \in \textbf{runs}(Q)$ there exists a run $\pi_P \in \textbf{runs}(P)$ such that $\textbf{emb}(M(\pi_Q), \pi_P)$ by Definition 5.2.9. By Theorem B.0.7, there exists a $\pi'_Q \in \textbf{runs}(\textbf{col}(\textbf{dif}(M_Q(Q))))$ and a $\pi'_P \in \textbf{runs}(\textbf{col}(\textbf{dif}(M_P(P))))$. By Theorem B.0.2, $\textbf{emb}(M(\pi_Q), \pi_P)$ implies $(\forall a_i \in \mathscr{A}_P : \mathscr{I}, g_0 \models \textbf{AG}(a_i.\textit{sub-guard} \rightarrow a_i.\textit{super-guard}))$, which implies the CTL formulas at Line 7 are true.

Because all detached commitments must eventually resolve, the fairness formulas at Line 10 are true. Because all commitment coverings must be valid, the formulas at Lines 14-15 are true. Because all serial compositions must be valid, the formulas at Line 18 are true. Because protocols must be well defined, the formulas at Line 21 are true. Because all of the CTL formulas evaluate to true, **refines**(P,M,Q) returns true. □

5.6 Discussion

Commitments support finer guard granularity than propositions can. A proposition divides time into two stages: before and after it holds. A commitment divides time into four stages: null, active and conditional, active and detached, and resolved.

$$\mathit{null} \xrightarrow{\text{CREATE}} \mathit{cond} \xrightarrow{ant} \mathit{detached} \xrightarrow{csq} \mathit{resolved}$$

Rather than waiting for final resolution, a protocol can make progress sooner if an action's guard is enabled after one of the first three stages. Commitments increase protocol flexibility, because guards can specify earlier stages.

For example, suppose the Buyer role in *OrderPayShip* decides whether to pay based on the state of proposition ship. Since ship has only two stages, the role's decision can only be "all" (*ship* complete) or "nothing" (*ship* not complete). The "all" choice is represented by the guarded statement [*ship*] *pay*, and the "nothing" choice is represented by [*true*] *pay*. Using commitments, the protocol can guard *pay* based on any of the four commitment stages. A guard can enable *pay* as soon as the debtor commits to make *ship* true.

$$[\text{CREATE}(\mathsf{C}_{\text{Seller,Buyer}}(\mathit{pay}, \mathit{ship}))]\ \mathit{pay}$$

Incorporating commitments can improve flexibility over traditional protocol frameworks.

A necessary prerequisite of employing protocols is that the participants of a service engagement agree on the format and meanings of the messages they exchange. Note that such agreement is unavoidable: it is just that in today's practice the meanings are not expressed clearly and explicitly and any agreements are hardcoded in implementations.

Our definition of protocol refinement does not mean agents that can participate in a superprotocol can necessarily participate unchanged in a subprotocol. In our model, agents may need to be modified to participate in subprotocols. For example, an agent capable of participating in a basic payment protocol needs to handle the additional messages required in paying via check or credit card.

5.6.1 Relevant Literature

Proton is the first approach for protocol refinement that incorporates mapping super-elements to expressions of sub-elements. Proton supports mapping super-propositions to Boolean expressions of sub-propositions as well as mapping super-commitments to serial compositions of chains of sub-commitments.

Mallya and Singh [2007] propose a definition of protocol refinement (which they call subsumption) that compares the order of state pairs in state runs. For every pair of states in the superprotocol, there must be *some* matching pair of states in the subprotocol with the same order. However, this definition can create false positives when multiple state pairs in the superprotocol each match the same state pair in the subprotocol or when one super-state matches different sub-states. For example, all state pairs in the super-run $\langle 1,2,3\rangle$ have matching state pairs in the sub-run $\langle 2,1,3,2\rangle$ even though the two runs are very different. Our definition compares runs step-by-step and thereby so avoids the above problems, even if protocol looping is allowed.

Our definition of commitment covering is an extension of *commitment strength* as defined by Chopra and Singh [2009], who identify the basic requirements in Equations 3.11 and 3.12. We extend Chopra and Singh's definition with the role requirements in Equations 3.9 and 3.10, and we allow commitments to be at different levels of abstraction by including an abstraction mapping function.

Singh [2008] states rules for commitment chaining similar to those proposed here, but does not directly state a rule for stronger consequents, and does not directly state a rule similar to serial composition. The concrete commitment created by serial composition provides a midpoint in commitment reasoning, and can potentially make the comparison of commitments across protocols more explicit.

When we say a group of debtors are jointly and severally responsible for eventually making the consequent true, we mean this in the sense of Rescher's [1998] *legal* responsibility where "individual agents are responsible only for their own individual acts." We do not mean Rescher's notion of legal responsibility where the group as a whole becomes a legal person, nor his notion of *moral* responsibility where intentions are crucial (intentions are absent from our formulation). In Norman and Reed [2002], group imperatives can be addressed *distributively* (as a list of individuals) or as a *collective*. In both cases, group imperatives imply more than just a collection of individual imperatives. While joint and several responsibility is similar to distributive responsibility, because only one member of the group is required to act, it is different, because joint and several responsibility is only a summary of individual responsibilities, and does not impose additional responsibilities on roles that are members of a group.

Our work on protocols builds on the fundamental intuition that protocol states can be effectively characterized in terms of the commitments of the participants, and that such characterization can be used as a basis for correct enactments and for further reasoning. The earliest works that developed the above theme include the commitment machines approach of Yolum and Singh [2002] for business

protocols and McBurney and Parsons' [2002] framework for sequencing multiple dialogue games, allowing one dialogue game to be embedded inside another partially completed dialogue game.

We do not propose specific, desirable properties of protocols, but others have. Yolum [2007], Singh and Chopra [2010], and El-Menshawy et al. [2010] describe desirable properties of protocols in general and commitment protocols in particular, including fairness, safety, liveness, operability, and transparency.

We use Boolean guards to constrain actions, but other representations are possible. Baldoni et al. [2010a] and Marengo [2013] proposed constraints based on regulative specifications. Regulative specifications constrain the execution flow using special-purpose operators on state values. Gabbay [1987] proposes using past-temporal expressions for controlling when actions can occur and future-temporal expressions for controlling which actions must occur in the future. Past-temporal expressions are more expressive than our guard expressions.

5.6.2 Directions for Future Research

Constructing a mapping function from a superprotocol to a subprotocol can be a challenging task. Advice to guide protocol designers, in the form of a basic mapping methodology, would be a valuable addition to this work. Winikoff [2006; 2007] proposed a methodology for the related task of designing commitment-based protocols. Some of these ideas could be valuably adapted into a future commitment mapping methodology. The approach begins with an easily understood, but not exhaustive, set of sequence diagrams, and then specifies specific steps to generalize the protocol and expand its set of runs.

Model checkers have been extended to handle epistemic and strategic modal operators [Alur et al., 2002; Fagin et al., 1995]. We have begun investigating the inclusion of such concepts into our definitions. Building on top of a model checker that already handles those concepts, such as MCMAS, will simplify our future extensions. Another important enhancement would be to expand the class of protocols to those that support iteration.

We thank the anonymous reviewers for valuable comments and Amit Chopra for his suggestion of describing complex messages as sets of primitive actions.

Chapter 6

Evolving Protocols and Agents in Multiagent Systems

We consider multiagent systems that involve two or more business partners interacting via autonomous software agents. Such systems pose a major challenge with requirements evolution. Current approaches couple agent and protocol designs, requiring coordinated changes. In contrast, we propose an approach that decouples agent and protocol designs, while maintaining interoperability. We build on the well-known architectural construct of an *interceptor*. We introduce *interaction refactorings* to transform interactions in response to evolving requirements, with each refactoring incrementally changing agents, interceptors, and the protocol. We identify three main forms of requirements evolution and propose an extensible library of refactorings, called *Rho*, that helps address each form. We demonstrate the approach through examples and a JADE prototype.

6.1 Introduction

We consider cross-organizational multiagent systems that arise when two or more business partners interact, for example, to carry out complex service engagements. Each business partner implements a software *agent* that appears to the rest of the system to be autonomous and active (both proactive and reactive). To facilitate the interoperation of the partners' agents, such systems are often built using *(business) protocols* that specify the messages that the agents may exchange along with any constraints on such messages.

Although such protocols are valuable for engineering, they result in architectural coupling: Designers cannot deploy a new protocol until all parties agree and their agents are modified accordingly. In general, protocol and agent interfaces must change together.

In essence, the decentralized nature of multiagent systems makes it difficult to handle evolving requirements since any change appears to demand bulk (concurrent and coordinated) updates,

which are precisely ill-suited for a decentralized system. In today's practice, the business partners negotiate such updates by personal communication. The traditional approach faces a vicious cycle. First, without numerous agent implementations that exploit a new protocol, protocol adoption is hindered. Second, without wide protocol adoption, agent designers are little motivated to implement a new protocol.

6.1.1 Problem: Requirements Evolution

Agents are designed by agent designers and protocols are designed by protocol designers. We assume agent and protocol designers are distinct. When requirements change, to break the vicious cycle of the traditional approach, we desire a system where (1) concurrent and coordinated deployments are unnecessary; (2) agents can interoperate using an evolved protocol, without agent code changes; (3) each designer can work independently; and (4) designers can collaborate when necessary.

To illustrate our proposal, we use two payment protocols, *Pay* in Figure 2.1b and *PayViaCheck* in Figure 2.1d on page 14. We identify three forms of requirements evolution in the above setting.

Protocol Designer Independence **(PDI)** Assume the agents initially employ *Pay*. A new legal requirement arises to ensure all payments are traceable, which *PayViaCheck* addresses. How can a protocol designer evolve protocol *Pay* to *PayViaCheck* to address this requirement without having to ask that all agents be concurrently updated?

Agent Designer Independence **(ADI)** Assume the agents initially employ a payment protocol that supports travelers checks and other forms of payment. A new requirement arises for a specific PAYER agent to reduce costs by ceasing to use travelers checks. How can this agent simplification result in protocol simplification?

Designer Collaboration **(DC)** At times, designers must collaborate, with one agent's changes propagating to other agents. DC changes are an integral element of the protocol simplification just mentioned.

6.1.2 Approach: Refactoring Interactions

Our approach builds on the time-honored architectural abstraction of an *interceptor* or Chain of Responsibility pattern [Gamma et al., 1995; Vinoski, 2002]. Extending this, we show how to construct interceptors modularly in a rule-based manner from logical specifications of refactorings. Specifically, each interceptor is expressed as a series of reaction rules that are triggered by a message and which may refer to the interceptor's internal state. An *interceptor chain* is an ordered list of zero or more interceptors, that mediates all message flow to and from its agent. Incoming messages pass through an interceptor chain before arriving at the business logic component of an agent and outgoing messages likewise pass through the same interceptor chain in reverse order.

Given one or more agents that use a protocol, designers can incrementally *refactor* the agent

and protocol interactions while preserving interoperability of the agents. A refactoring defines a set of coordinated, incremental changes to agents, interceptor chains, and the protocol. We provide an extensible library of refactorings, *Rho*, from which designers may select and apply one or more refactorings to implement requirement changes. We partition refactorings into three groups based on the requirements evolution problem they address:

PDI For example, to evolve *Pay* to *PayViaCheck*, the protocol designer adds redemption processing by adding interceptors to convert each *pay* message to the message sequence *deposit*, *confirm*, *check*, *redeem*, and *payB*.

ADI Our approach enables agent and agent interface changes: (1) moving (internalizing or externalizing) functionality between the agent implementation and the interceptor chain, and (2) an agent declaring it will not send messages in cases enabled by the protocol.

DC These refactorings enable reorganizing, optimizing, and simplifying an interceptor chain.

Contribution and Organization

Our main contribution is the concept of *interaction refactorings* that enable independent and incremental evolution of interactions, decoupling the efforts of agent and protocol designers. Section 6.2 describes our enabling framework. Section 6.3 introduces representative refactorings and the underlying interceptor architecture. We apply refactorings to our example protocols in Section 6.4. We evaluate a prototype implementation in Section 6.5. Section 6.6 concludes with comparisons and a discussion of related work.

6.2 Approach Illustrated

For brevity and clarity, we introduce the key concepts and syntax for our approach via examples.

6.2.1 Applying Rule-Based Interceptors

Figure 6.1 shows a simple, concrete example of our architecture, in which Rebecca's agent R, enacting role *Requester*, sends a request message to Steve's agent S, enacting role *Service*. S performs its function and responds. When the interceptor chains are empty, R and S interoperate using the *ReqResp* protocol (top dotted line).

Suppose the protocol designer determines that R is actually placing an order, and S is actually returning a confirmation. The protocol designer then requires R and S must now interact with specialized protocol *Order* using messages *order* and *confirm* (bottom dotted line). Without needing to change either agent's implementation, the protocol designer provides two interceptors for each agent's interceptor chain, evolving protocols from *ReqResp* to *Order*.

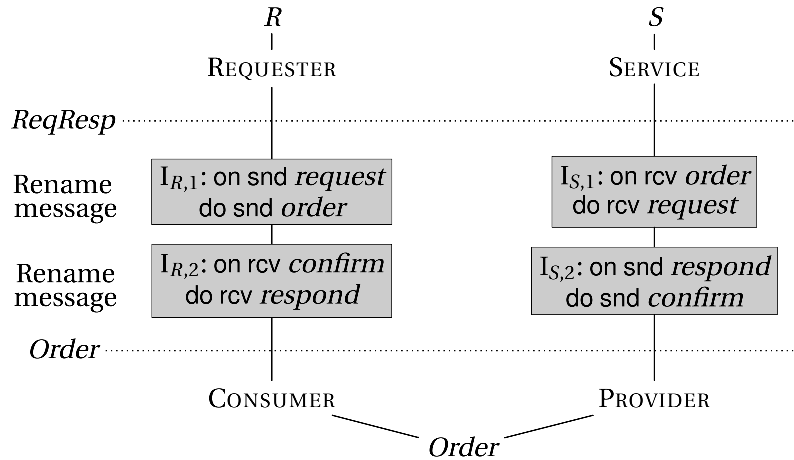


Figure 6.1: Evolving *ReqResp* to *Order*.

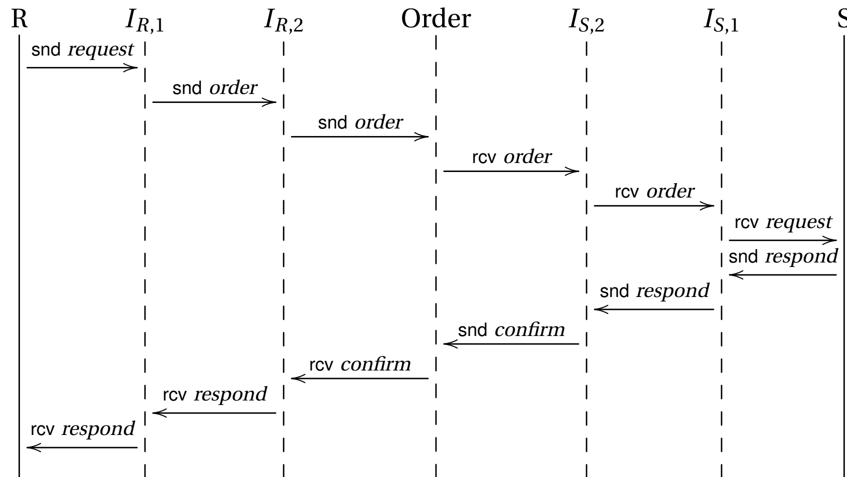


Figure 6.2: Detailed enactment of Figure 6.1.

Figure 6.2 is a message sequence chart of the interaction, including both agents (solid lifelines) and all their interceptors and protocol (dashed lifelines). When R sends *request*, R's top interceptor ($I_{R,1}$) converts it to send *order*, which flows down to the protocol end of R's interceptor chain, and is sent over the protocol (Order) via the messaging infrastructure to S's interceptor chain. The *order* message flows up S's interceptor chain. Its top interceptor ($I_{S,1}$) converts it back to message *request*. S's *response* is converted to *confirm* in its bottom interceptor ($I_{S,2}$), and is sent over the protocol (Order) to R, whose bottom interceptor ($I_{R,2}$) converts it back to *respond*. The essential point of this example is both R and S use the original *ReqResp* protocol, even though messages of the *Order* protocol are

what flow on the wire.

6.3 Interceptors and Refactorings

Given a set of agents that interoperate using a protocol, designers can incrementally *refactor* agent and protocol interactions to an evolved interaction, while preserving interoperability. *Interceptors* and *interceptor chains* mediate all message flow between an agent and a protocol using a reaction (event-based) architecture. A refactoring defines a set of coordinated and incremental changes to agents, interceptor chains, and the protocol. Interceptors and interceptor chains are the key elements that make refactorings possible.

6.3.1 Interceptor Chains

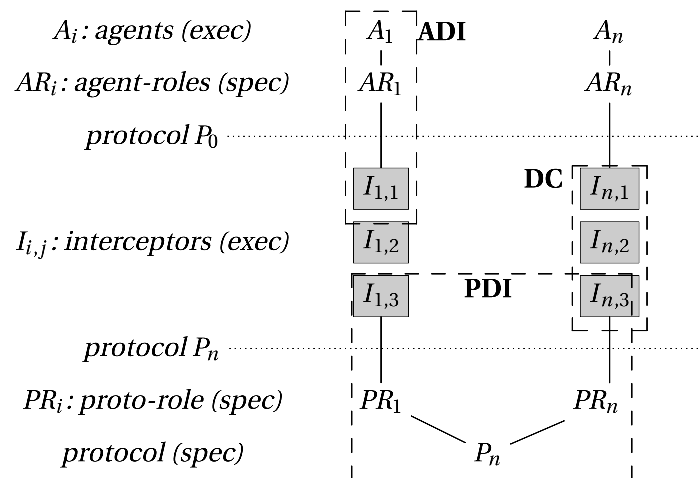


Figure 6.3: Interaction architecture. The dashed boxes signify the three refactorings types introduced above.

Figure 6.3 shows our *interaction architecture* consisting of agents (A_i), interceptor chains ($I_{i,*}$), interceptors ($I_{i,j}$), and protocols (P_k). Each agent, enacting role (or interface) AR_i of protocol P_0, communicates exclusively with the agent (top) end of its interceptor chain. An interceptor chain is an ordered list of interceptors (shaded boxes). The protocol (bottom) end of an interceptor chain, enacting role (or interface) PR_i of protocol P_n, communicates with the protocol end of other agents' interceptor chains. Agent implementations and interceptors are executable elements; roles and protocols are nonexecutable specifications.

In Figure 6.3, the top dotted line (P_0) separates the agent *implementation* and *Role* (above) from the *middleware* of the interceptor chain and protocol (below). The bottom dotted line (P_n) separates the agent and interceptor chain *nodes* (above) from the protocol *interconnection* (below). Figures 6.1 and 6.4 are concrete instances of Figure 6.3.

Interceptor chains are the key to our approach because they enable the required designer independence: they insulate agents from protocol changes and insulate protocols from agent changes.

6.3.2 Interceptor Syntax and Semantics

Interceptors are preprogrammed elements provided by the infrastructure, and require no designer implementation. Interceptor chains are constructed using the following grammar

$$
\begin{array}{lll}
\textit{chain} & := & \textit{role} : \textit{interceptor}^{*} \\
\textit{interceptor} & := & \textit{reaction} \,|\, \textit{assertion} \\
\textit{reaction} & := & \textit{onClause}(\textit{ifClause})^{?}\ \textit{doClause}; \\
\textit{onClause} & := & \mathsf{on}\ \textit{event} \\
\textit{ifClause} & := & \mathsf{if}\ \phi \\
\textit{doClause} & := & \mathsf{do}\ \textit{op} \,|\, \mathsf{do}\ \{\textit{op}^{*}\} \\
\textit{event} & := & \mathsf{rcv}\ m \,|\, \mathsf{snd}\ m\ \mathsf{to}\ \textit{role} \\
\textit{op} & := & \mathsf{rcv}\ m \,|\, \mathsf{snd}\ m\ \mathsf{to}\ \textit{role} \,|\, \mathsf{error} \,|\, \mathsf{call}\,\textit{proc} \\
\textit{assertion} & := & \mathsf{kill}\ \textit{event}
\end{array}
$$

where *role* is a role name, m is a message type, *proc* is a procedure name, ϕ is a Boolean expression, and BNF operators: $A_1|A_2$ (alternatives), A^* (zero or more repetitions), and $A^?$ (optional). The *doClause* is an ordered list of (1) receive operations (rcv m) that "call up" the chain, (2) send operations (snd m to role) that "call down" the chain, (3) throwing a run-time error (error), and (4) procedure calls to get or set interceptor chain data, or perform business functions. Assertions are design-time declarations and optionally perform run-time checks. The interceptor kill *event* asserts that *event* can never occur at a particular point in the chain at run time. It is used to propagate message deletions throughout the interaction.

At run time, an interceptor chain mediates all messages flowing in either direction between its agent and the protocol. The chain attempts to match each message event to each interceptor, in order. A send message event starts at the agent (top) end of the chain, and "calls down" the chain toward the protocol end (bottom), passing over every interceptor in the chain in turn. A receive message event starts at the protocol end, and "calls up" the chain toward the agent end.

A send event (snd m to role) matches an (on snd m′ to role′) reaction, and a receive event (rcv m) matches an (on rcv m′) reaction, if the message types match ($m = m'$) and to roles match ($\textit{role} = \textit{role}'$). When the event matches both (1) the message and (2) the reaction's ifClause evaluates to true in the

current state, then (3) the interceptor consumes the message event and executes the list of operations in the doClause. Message events that reach the agent end of the chain are given to the agent; message events that reach the protocol end are given to the messaging infrastructure for delivery to the receiver.

6.3.3 Refactorings Formalized

A *refactoring* is a design-time construct, which encapsulates a coordinated and incremental set of interaction changes. For example, refactoring *Add Message* encapsulates all the interceptors for both the message sender and receiver. And refactoring *Add Middleman* encapsulates all the interaction changes for rerouting an existing message through a middleman, including all interceptors for the sender, middleman, and receiver.

A refactoring is a five-tuple that encapsulates one interaction-level change in high-level terms. As necessary, it can change all agents, all interceptor chains, and the protocol, applying an interrelated set of changes throughout, for example consistently renaming a message at both sender and receiver.

$$R = \langle \mathit{parameters}, \mathit{precondition}, \Delta \mathit{Agent}, \Delta \mathit{Chain}, \Delta \mathit{Protocol} \rangle$$

Given refactoring *parameters*, the *precondition* must be true at design time for the refactoring to be applicable. The refactoring applies changes to *Agents*, *Chains*, and *Protocols* at design time. Refactorings names are italicized, and each refactoring tuple is described with these common sections (omitting any empty sections)

- Parameters: input parameters to the refactoring.
- Preconditions: the preconditions that must be true before the refactoring can be applied.
- ΔAgent: changes to agents' implementations.
- ΔChain: changes to agents' interceptor chains. The notations $\mathit{role.push}_A : I$, $\mathit{role.pop}_A : I$ $\mathit{role.push}_P : I$ and $\mathit{role.pop}_P : I$ mean push or pop interceptor I on to the chain's agent and protocol end, respectively.
- ΔProtocol: changes to the protocol.

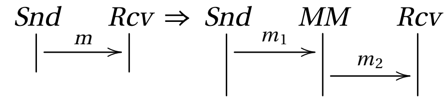
There are three groups of refactorings, each modifying a different set of elements, which we describe next.

6.3.4 Protocol Designer Independence

PDI refactorings modify the protocol, the protocol role, and the protocol end of interceptor chain. The protocol designer applies these refactorings to convert a group of interoperating agents from using protocol P_i to using protocol P_{i+1}. These refactorings isolate agents from protocol changes ($\Delta Agent = \emptyset$).

Add Middleman

This refactoring redirects a message to flow through a middleman agent. It replaces a single message m with a pair of sequential messages m_1 and m_2.

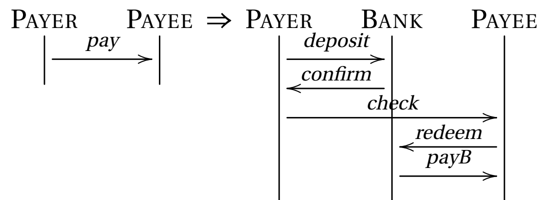


- Parameters:
  - m: existing message $Snd \to Rcv : [g]m$ means A
  - m_1: new message $Snd \to MM : [g_1]m_1$ means A_1
  - m_2: new message $MM \to Rcv : [g_2]m_2$ means A_2
- Preconditions:
  - Roles *Snd*, *MM*, and *Rcv* exist in the protocol.
  - m exists in the protocol.
  - m_1 and m_2 do not exist in the protocol.
  - $g_1 \vdash g$
  - All meanings in A, A_1, and A_2 exist
  - $A \subseteq A_1 \cup A_2$
- ΔChain: Add

$$\begin{aligned} Rcv.push_P &: \text{ on rcv } m_2 \text{ do rcv } m \\ MM.push_P &: \text{ on rcv } m_1 \text{ do snd } m_2 \text{ to } Rcv \\ Snd.push_P &: \text{ on snd } m \text{ to } Rcv \text{ do snd } m_1 \text{ to } MM \end{aligned}$$

- ΔProtocol: Delete m. Add m_1 and m_2.

This refactoring can be naturally extended to reroute through multiple middlemen, which is the variant we implement in our prototype. For example, it converts *pay* in *Pay* to *deposit*, *confirm*, *check*, *redeem*, and *payB* in *PayViaCheck*.



6.3.5 Agent Designer Independence

ADI refactorings modify the agent implementation, the agent role, and the agent end of the interceptor chain. The ADI refactorings isolate the protocol from agent changes ($\Delta Protocol = \emptyset$).

As examples, the agent designer can move functionality between the agent and its interceptor chain. Refactoring *Externalize Reaction* moves functionality out of the agent implementation and into the agent end of the interceptor chain, delegating agent functionality to a mechanistic reaction. Refactoring *Internalize Reaction* moves an interceptor off the agent end of interceptor chain and the agent designer merges that functionality into the agent's implementation.

Push Kill is another ADI refactoring that is described in Section 6.3.6.

6.3.6 Designer Collaboration

DC refactorings modify interceptors within a single interceptor chain. Protocol and agent designers can apply these refactorings to reorder, merge, or split interceptors within a chain to improve performance or to move interceptors toward one of the ends of the chain where they can be used in other refactorings. While refactorings in the other two groups isolate designers, these refactorings enable multiple designers to collaborate within an interceptor chain ($\Delta Agent = \emptyset = \Delta Protocol$).

Kill Message

This captures a set of three closely related refactorings—one ADI, one DC and one PDI—with similar parameters and preconditions, presented together for clarity. We iteratively apply these refactoring to move kill assertions around the interaction at design time (kill assertions are not typically present in chains at run time).

- Parameters:
  - Kill assertion kill snd m to Rcv
- Preconditions:
  - Protocol contains message m.
- ΔAgent: (ADI: *Push Kill*) Sending agent publicly declares it will never send m by publishing kill snd m onto its chain.
  - $Snd.push_A$: kill snd m to Rcv
- ΔChain: (DC: *Move Kill*) moves the kill up or down the chain. If a kill event matches the following on event, delete the reaction (left rule). If a kill event does not match the following onClause or doClause events, swap the interceptors (right rule). Similar rules apply for on rcv reactions, and two kill assertions commute (neither shown).
- ΔProtocol: (PDI: *Protocol Kill*) propagates the kill assertion from sender to receiver, deleting the message from the protocol.

$$\frac{\begin{array}{l}\text{kill snd } m\\ \text{on snd } m \text{ do snd } n\end{array}}{\text{kill snd } n} \qquad \frac{\begin{array}{l}\text{kill snd } m\\ \text{on snd } n \text{ do } \ldots\end{array}}{\begin{array}{l}\text{on snd } n \text{ do } \ldots\\ \text{kill snd } m\end{array}}$$

– $Snd.pop_P$: kill snd m to Rcv
– $Rcv.push_P$: kill rcv m
– Delete m from protocol

6.4 Methodology and Application

In this section, we describe a methodology for selecting a sequence of refactorings. Then, we show how to evolve protocol *Pay* to protocol *PayViaCheck*, and how to support requirements evolution by propagating kill message assertions.

6.4.1 Methodology for Protocol Evolution

These steps guide the protocol designer in the evolution of an interaction from protocol P_i to P_{i+1}.

M1 Add or rename any roles so all new roles exist in the target protocol. Use *Add Role* (adds new role and empty interceptor chain) or *Map Role*.

M2 If any message is too coarse (one message with a larger-than-necessary set of meanings), split it into multiple, parallel messages using *Split Message*.

M3 Rename existing messages with *Rename Message*, and add new messages using *Add Message*.

M4 If any message needs to pass through one or more intermediary roles (common when adding new roles), reroute the messages using *Add Middleman*.

M5 If business function changes are required, add and delete procedure calls to the doClauses using *Add Procedure* and *Delete Procedure*.

M6 Combine parallel messages using *Merge Message*.

M7 Delete unneeded elements using *Remove Middleman* and *Remove Message*.

M8 Delete unneeded roles using *Remove Role*.

6.4.2 Evolve Pay to PayViaCheck

We demonstrate how our refactorings convert *Pay* (Figure 2.1b, Algorithm 6.1) to *PayViaCheck* (Figure 2.1d, Algorithm 6.2), without requiring any agent implementations changes, using the following sequence of refactorings. Each step lists the methodology step number and the affected line numbers in Algorithm 6.2. Figure 6.4 shows the refactorings and interceptors.

1. *Add Role*: BANK. (Step M1, Line 1)

Listing 6.1 *Pay* Protocol

```
protocol Pay {
    role Payer, Payee;
    prop promise, pay;
    commitment
        C_pay = C(Payer, Payee, promise, pay);
    message
        Payer → Payee : promiseMsg
            means {promise, CREATE(C_pay)};
        Payer → Payee : [promise] payMsg
            means {pay};
}
```

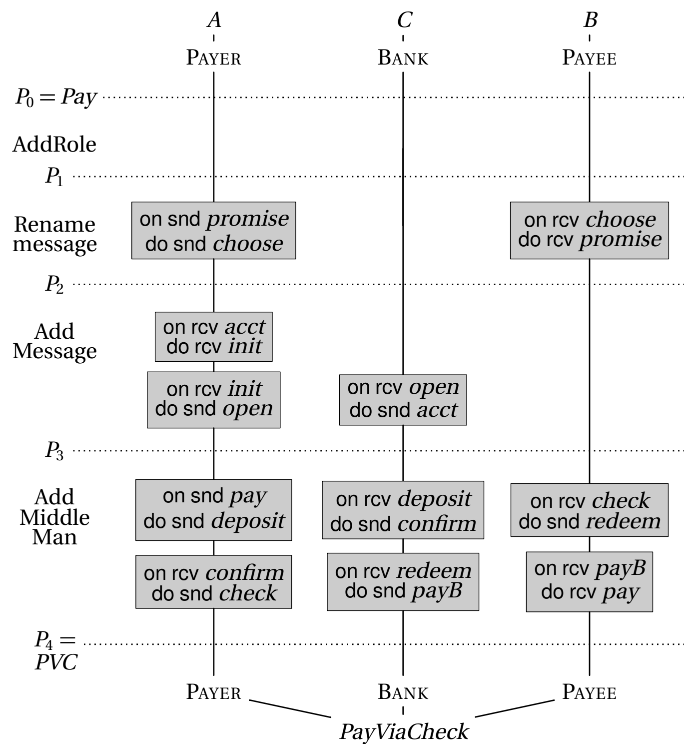


Figure 6.4: Evolution of *Pay* to *PayViaCheck* (PVC). Each shaded box is one interceptor.

2. *Rename Message*: *promiseMsg* ↦ *chooseMsg*. The two protocols use different names for the same message. (Step M3, Lines 9-17)
3. *Add Message*: *open* and *acct*. During initialization (rcv *init*), PAYER sends an *open* request, and BANK responds with an *acct* number. (Step M3, Lines 18-19)
4. *Add Middleman*: routes *pay* through multiple middlemen as *deposit*, *confirm*, *check*, *redeem*,

Listing 6.2 *PayViaCheck* Protocol

```
1: role Payer; Bank; Payee;
2: prop acct; deposit; choose; check; redeem; payB;
3: commitment
4:     C_payB = C(Payer, Payee, deposit ∧ choose, check);
5:     C_bank = C(Bank, Payer, deposit ∧ check ∧ redeem, payB);
6:     C_redeem = C(Payee, Bank, deposit ∧ check, redeem);
7: message
8: // Map promise to choose
9:     Payer → Payee : [acct] chooseMsg
10:        means {choose, CREATE(C_payB)};
11: // Add Message
12:     Payer → Bank : openMsg means {open};
13:     Bank → Payer : [open] acctMsg
14:        means {CREATE(C_bank), CREATE(C_redeem)};
15: // Add Middleman to pay
16:     Payer → Bank : depositMsg means {deposit};
17:     Bank → Payer : [deposit] confirmMsg means {};
18:     Payer → Payee : [acct ∧ choose ∧ CREATE(C_payB) ∧
19:        CREATE(C_bank) ∧ CREATE(C_redeem)] checkMsg;
20:        means {check};
21:     Payee → Bank : [choose ∧ check ∧ CREATE(C_payB) ∧
22:        CREATE(C_bank) ∧ CREATE(C_redeem)]
23:        redeemMsg means {redeem};
24:     Bank → Payee : [acct ∧ check ∧ redeem]
25:        payBMsg means {payB};
```

and *payB*. When PAYER pays, then deposit the money at BANK, wait for confirmation, and send *check* to PAYEE. PAYEE then redeems check at BANK, who responds with *payB*, which is converted back to *pay*. (Step M4, Lines 20-26)

6.4.3 Guard Propagation

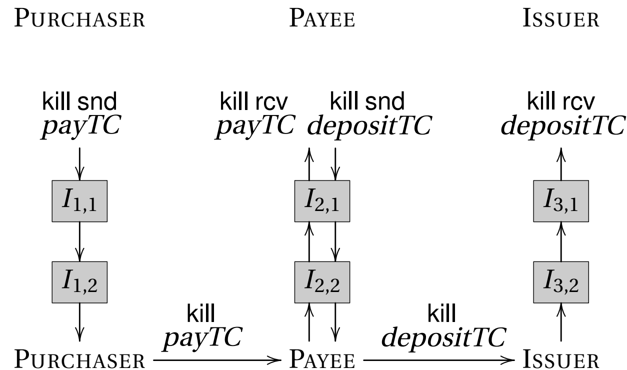


Figure 6.5: Removing unused message (false guard).

Applying refactorings from all three groups enables agent and protocol designers to collaborate on interaction-wide changes. Assume we have a set of interoperating agents, using a payment protocol that supports multiple forms of payment, including Travelers Checks. If one particular (but not necessarily every) PURCHASER decides to stop using Travelers Checks, it can publish that decision as a kill snd *payTC* assertion to its interceptor chain. Multiple refactorings propagate these change throughout the interaction as shown in Figure 6.5.

1. Applying *Push Kill*, PURCHASER declares it never sends *payTC*.
2. Repeated application of *Move Kill* moves this assertion down to PURCHASER's protocol end.
3. Applying *Protocol Kill*, the protocol designer propagates the kill assertion from sender to receiver and deletes *payTC* from the protocol.
4. Repeated application of *Move Kill* moves the kill assertion up to PAYEE's agent end.
5. PAYEE's agent designer pops the assertion off its agent end and internalizes it into its implementation.
6. Since PAYEE now never receives *payTC*, it realizes it never sends *depositTC*, and publishes kill snd *depositTC*, which propagates further.

The resulting protocol is different from the starting protocol, and is specialized for a particular set of agents. Overuse of these refactorings should be avoided to prevent an explosion of protocol

variations. But when applied sensibly, these refactorings provide a natural means to incrementally evolve old interactions to handle changing requirements.

Designers have the option, but not the hard requirement, to move assertions as described above. Any assertion simply remains at its last location, where its optional run-time check will succeed on any acceptable enactment.

6.5 Evaluation

We have prototyped these ideas using the JADE agent platform [Bellifemine et al., 2007], with one JADE agent for each agent and one JADE "chain agent" for each interceptor chain. Each chain agent has its own thread of execution and message input queue, so it can send and receive messages without blocking.

First, at design time, a program builds the interceptor chains by applying refactorings. Second, at run-time initialization, each chain agent reads its interceptors. Third, as message events arrive from agents, the chain agent walks each message event up and down the interceptors in the chain.

An agent can be connected to multiple, different interceptor chains in different situations, enabling that single agent to simultaneously interact over different protocols.

Table 6.1 shows the number of refactorings and interceptors needed to evolve between protocols. Messages is the total number of messages in the final execution, including six initialization and six terminating, low-level messages not shown in the running examples in Figure 2.1 on page 14.

Table 6.1: Effort in evolving (refactorings and interceptors) and running (messages) sample protocols.

From	**To**	**Refactor**	**Intercept**	**Messages**
Pay	*PayViaMM*	2	5	19
Pay	*PayViaCheck*	4	11	24
Pay	*PayViaCredit*	5	12	24

Section 6.4.1's methodology offers guidance for designers, focusing on *Protocol Designer Independence* refactorings. However, Section 6.4.3 illustrates the key benefit that *Agent Designer Independence* and *Designer Collaboration* refactorings yield in interaction-wide changes. We claim our approach is easier than the traditional approach of manually changing agent implementation, because (1) predefined and verified refactorings are selected from a library, and (2) refactorings are at a much higher conceptual level than agent implementations.

Our interaction refactorings do not assist in refactoring business functions inside the agent's implementation.

Let reaction R be on snd m do snd n. If R is the only generator of snd n, then the *Move Kill* works correctly: if kill m, then kill n. But if R merges two existing messages, then it overkills snd n. While merging does not occur in the examples covered here, in general, it could. The current approach cannot adequately address such merging.

Refactorings can time-shift messages only within a limited range. The data values an interceptor passes in a message must come from previous messages or values stored in the interceptor chain. A message cannot be shifted earlier than the availability of all its parameters, and it cannot be shifted later than the next message that needs one of those values. This constraint required altering *PayVia-Check*'s deposit message, which originally required an up-front, one-time deposit. The refactored design can make a deposit just before sending each check. Without this change, the refactoring would not have been possible.

6.6 Discussion

Refactorings clearly communicate high-level, multirole changes. They mechanically generate pretested sets of interceptors.

We identify and describe three forms of requirement evolution: *Protocol Designer Independence* (PDI), *Agent Designer Independence* (ADI), and *Designer Collaboration* (DC). Each focuses on different parts of an interaction, two provide designer isolation, and one enables designer collaboration. We describe refactorings for all three forms. Applying refactoring from all three forms, in concert, supports interaction-wide evolution. Interceptors and interceptor chains are the critical elements that enable refactorings.

We demonstrated refactorings to transform *Pay* into *PayViaCheck*, without changing agent implementations. We also demonstrated an agent voluntarily restricting its behavior (*payTC*) and propagating that change throughout the interaction.

This paper covers just a few refactorings, but we have defined the Rho refactoring library with 30 refactorings. A JADE prototype demonstrates basic interceptor chain functionality, refactorings automatically generating reactions, and agent interoperability after refactoring to a new protocol.

We adopt a reaction-based, interceptor chain architecture that is effective and yet simple enough to yield refactorings that are easy to understand and apply. Because interceptors are predefined and simple, we can define refactorings to mechanically evolve them. Mechanical evolution of general-purpose agent implementation is likely intractable.

A refactoring's functional changes are orthogonal to other important topics. We briefly describe a few such topics. An agent must be able to *secure* its interceptor chain, allowing only certain agents to view or change the chain's interceptor and data contents; e.g. which protocol designers can add or remove interceptors? An agent must be able to control the *autonomy* its interceptor chain has to act as the agent's trustee, making commitments on its behalf; e.g. does Global Bank trust its interceptor

chain to commit to redeeming valid checks? An agent must be able to control the *trust* it grants its interceptor chain to process and store important pieces of information; e.g. does Alice trust her interceptor chain to store and use her bank account information? An agent must be able to control the *trust* its interceptor chain has in other agents; e.g. does Alice trust her interceptor chain to interoperate with Honest Bank or Shady Bank?

A refactoring transforms a protocol to a slightly modified version of the protocol. Savarimuthu and Winikoff [2013] introduce mutation operators for the cognitive agent-oriented programming language GOAL. Likewise, we can interpret each refactoring as a mutation operator. Protocol mutation can be used for *mutation testing* to assess the strength of a protocol test suite. Protocol mutation can also be used as a basis for genetic evolution of protocols.

6.6.1 Comparison to Design Patterns

Interceptor chains are a variation of the Chain of Responsibility (CoR) design pattern in Gamma et al. [1995]. Vinoski [2002] describes many uses of CoR. Servlet filters and filter chains [Sun, 2009] are a widely used example of CoR. But we know of no uses of CoR that support bidirectional flows. Nor do we know of a multi-interceptor design construct like ours.

Interceptor chains enable all the uses of servlet filters plus others. Servlet filter chains encourage servlet designers to consider moving function between a servlet and its filters (like ADI), and reordering filters within the chain (like DC). But servlet filters give no attention to the coordinated design of filters in different servlets (like PDI). Servlet chains do not support bidirectional flows.

Our interceptors are a bidirectional variant of the Bridge design pattern [Gamma et al., 1995], also called a protocol bridge. Where a protocol bridge is a custom implementation, our small preprogrammed interceptors are incrementally composed and refactored.

The Compatible Change pattern [Erl, 2008] describes a number of refactorings. The Service Refactoring pattern [Erl, 2008] applies only to service implementations, not interactions. Neither are mechanically applied.

6.6.2 Comparison to Agent Designs

In the traditional agent-only approach to agent design, evolution is subject to the vicious circle described in Section 6.1. Even when an agent designer decides to support a new protocol, implementation changes delay deployment. Agent designers can waste both time and effort implementing protocols that are never widely adopted. Using refactorings and interceptor chains breaks the vicious circle. Protocol designers dynamically update interceptor chains, essentially eliminating deployment delay. Agent designers spend time and effort implementing only after a protocol is widely adopted.

We demonstrated our approach by prototyping interaction evolution via interceptor chains in the popular agent platform JADE [Bellifemine et al., 2007]. Jason, using the AgentSpeak language,

is another popular agent platform. Our reaction's *onClause*, *ifClause*, and *doClause* are similar to an AgentSpeak plan's triggering event, context, and body, respectively. Both can send and receive "internal messages" (e.g., init) and call user-defined functions. Whereas beliefs, desires and intentions (BDI) are fundamental for autonomous functions in AgentSpeak, interceptor chains are not autonomous, so BDI does not apply. The primary problem these platforms have with evolution is that all computation occurs in designer-written agents, so all changes require designer effort, which can be expensive. It appears impossible in general to define mechanical refactorings that correctly evolve JADE behaviors or AgentSpeak plans. We *can* define mechanical interceptor chain refactorings only because interceptor chains have a simple structure, reducing the designer's burden.

Agent UML (AUML) [Odell et al., 2000] informally describes agent interaction protocols (AIP), and promotes them as a means to define protocol interactions. However, Odell et al. note that AIPs describe only one enabled sequence of message interactions. We formally define protocols as sets of guarded statements that capture all enabled message sequences. Guarded statements enable a relatively direct conversion [Gerard and Singh, 2013] to modern model checkers such as MCMAS [Lomuscio et al., 2009] and NuSMV [Clarke et al., 1999]. Gerard and Singh [2013] describe *protocol refinement*, but do not provide any guidance for constructing subprotocols. Here, we describe both refactorings and a methodology to incrementally evolve interactions.

6.6.3 Comparison to Other Work

Quenum et al. [2004] compose an agent from functional and interaction models via unification. They recreate (reconfigure) roles anew, in isolation, for each interaction model. We *incrementally evolve* (refactor) all agents in a protocol simultaneously.

Robinson and Purao [2009] describe protocol invariants using OCL, which is based on predicate calculus and linear temporal logic, but they provide no rules to rewrite OCL statements. Because we use a simple reaction-based architecture, we can mechanically modify interceptor chains.

Fowler [2000] refactors code; Wang et al. [2009] refactor commitments; we refactor interactions.

Baldoni et al. [2009] identify and discuss the important problem of patching agents to maintain interoperability. Seguel et al. describe protocol adaptors (interceptor chains) to resequence messages between a pair of agent for both synchronous [2009] and asynchronous cases [2010]. We use a declarative approach in contrast with these two operational approaches. Neither approach supports as many protocol changes as our refactorings, and they construct protocols rather than incrementally refactor them.

Serban and Minsky [2009] describe an infrastructure for changing a distributed system while it is running. Their laws and controllers roughly correspond to our protocols and interceptor chains. They enable changes on running systems, where we consider changes only while the system is quiesced. Their users must manually design, write and test a completely new set of laws for a set of interacting

agents; our designers expend less effort by incrementally evolving existing interceptor chains using our Rho library of predefined refactorings. They provide no guidance on how to design and construct laws. We provide a methodology for refactoring interactions.

6.6.4 Comparison of Mechanistic Capabilities

In Table 6.2 we list various related approaches and whether they *mechanistically* support PDI, ADI, and DC style changes. PDI indicates the protocol can be changed by renaming messages, adding middlemen, and so on. ADI indicates the messages an agents sends or receives can be changed. DC indicates the internal organization of the interceptor chain equivalent can be changed, or is NA if no equivalent exists.

Table 6.2: Compares representative agent and interaction programming approaches to mechanistically apply PDI, ADI, and DC changes. Some means partial support. (F) means unidirectional flow is from protocol to agent.

Approach	**PDI**	**ADI**	**DC**
Our Approach	Yes	Some	Yes
Chain of Responsibility (F) [Gamma et al., 1995; Vinoski, 2002]	No	Yes	No
Servlet Filter (F) [Sun, 2009]	No	Yes	No
Protocol Bridge [Gamma et al., 1995]	No	No	No
Compatible Change [Erl, 2008]	No	No	No
JADE [2007]	No	No	NA
Quenum et al. [2004]	No	Yes	NA
OCL [2009]	No	No	NA
Fowler [2000]	No	Yes	NA
Wang et al. [2009]	NA	NA	NA
Baldoni et al. [2009]	Some	Yes	NA
Seguel et al. [2009]	Some	No	No
Serban & Minsky [2009]	No	No	No

6.6.5 Future Directions

This work opens up interesting directions for future research. The current chain functionality is in a separate agent and requires minor changes to the way normal JADE agents send messages. Producing a modified JADE middleware that includes an interceptor chain component, whose contents can be changed at run time, supporting unmodified JADE programming patterns, would better enable evolution of service-oriented systems.

Replace the current extreme kill assertion with more flexible mechanisms that enable restricting a sender's, or relaxing a receiver's, private guard, capturing the "send less; receive more" intuition [Baldoni et al., 2009]. This will require careful tracking of valid events at every point throughout an interaction.

Provide formal verification of the soundness of our refactorings, possibly adapting techniques applied to protocols [Gerard and Singh, 2013] as well as traditional model checkers [Lomuscio et al., 2009; Clarke et al., 1999].

Acknowledgments

We are indebted to Jon Doyle, Anup Kalia, Pankaj Telang, Tao Xie, and the anonymous reviewers for helpful comments.

Chapter 7

Case Study

Using a single, realistic example, this chapter integrates the three previously described pieces: protocol composition from Chapter 4, protocol refinement from Chapter 5, and interaction refactoring from Chapter 6. Figure 7.1 shows the primary process of protocol transformations and verifications described in this chapter. It starts with a given application P_1 (upper left) and its Positron verification using Positron (lower left). Then a requirement is changed and refactorings create a modified model P_2 (upper right), which is also verified (lower right). Then, for suitable requirement changes, Proton verifies the modified model P_2 refines the original model P_1 (bottom center).

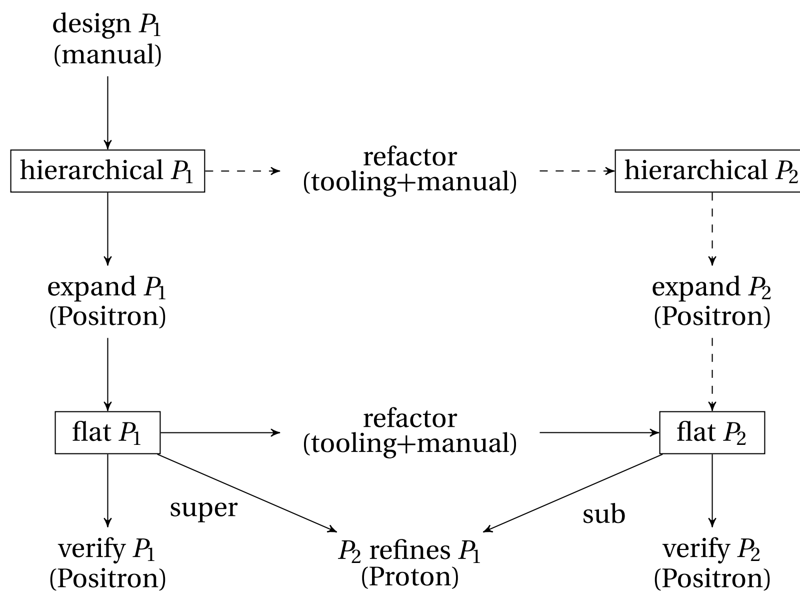


Figure 7.1: Protocol transformation and verification process applied to the example. Boxed elements are artifacts; unboxed elements are transformations or verifications.

The complete process consists of the following steps, including steps not depicted in Figure 7.1.

1. Construct a Composite Protocol Diagram (CPD) using the methodology defined in Section 4.3, including manual generation of the hierarchical Positron model (P_1).
2. Use Positron to verify P_1 meets its role and enactment requirements.
3. Identify a requirement change.
4. Refactor the existing model P_1, producing a new, refactored model P_2, using the methodology in Section 6.4.
5. Manually adjust requirements in the refactored model, if necessary. Since refactorings do not address role and enactment requirements, any requirements changes must be made manually.
6. Use Positron to verify P_2 meets its requirements.
7. (Optional) Verify whether protocol P_2 refines P_1. This need not be true for some possible requirement changes.

Figure 7.1 shows two approaches to combining refactoring and composition. The dashed steps first refactor each constituents of hierarchical protocol P_1, followed by expanding the hierarchical structure, creating a "flattened" version of P_1. The flattened version is then verified. Realizing this approach requires refactorings that operate on hierarchical compositions, likely including future refactorings for *Encapsulate Constituent Protocol* (move a set of protocol elements into a new constituent protocol) and *Unencapsulate Constituent Protocol* (essentially using protocol expansion from Section 4.2.1). Hierarchical refactoring is a practical requirement for any production deployment. But, because our Rho refactoring library does not contain such refactorings, we postpone further consideration of this attractive approach to future work.

The solid arrows first expand P_1 (a hierarchical collection of protocols like those in Figure 7.3), and then refactors the flat protocol to generate P_2. This approach is supported by the current Rho refactorings, and it is used for this case study.

Refactorings automate interaction changes, except role and enactment requirements. Role and enactment requirements may require manual updates. Future work could enhance some refactorings to correct some requirement formulas. For example, refactoring *Rename Message* could rename messages in all requirement formulas, and *Add Middleman* could add a coupling commitment for the middleman. However, this may not be possible for all refactorings. CTL is likely too expressive for some formula refactorings to be tractable. For example, both *Add Middleman* and *Remove Middleman* might require subtle and complex changes to requirement formulas. Further, what role requirement should *Add Role* introduce, since the new role sends and receives no messages immediately after that refactoring?

Refinement verification is not always a requirement because, depending on the refactorings applied, P_2 will not always refine P_1. But such verification is desirable, if possible. Where Positron verifies only explicitly stated good and bad enactments (as is expected of unit tests), Proton refinement verifies all paths in the two protocols. Further, designers will typically describe only partial paths for Positron; Proton evaluates full paths (runs). In particular, assuming P_2 refines P_1, Proton verifies that *every* enabled (good) path in P_2 is an enabled (good) path in P_1 and that *every* disabled (bad) path in P_1 is a disabled (bad) path in P_2. If designers have a certain level of confidence that P_1 correctly disables all bad paths, then they are guaranteed of the same level of confidence that P_2 correctly disables all of those bad paths too.

7.1 Application Selection Desiderata

The desiderata for selecting the example for this end-to-end case study are:

1. The example is defined in the literature, rather than defined by us, to reduce the chance of the example being overly tailored to our methodology;

2. The example is ambiguous to partially simulate real-world design situations where design must be partially inferred; and

3. The example is different from those we previously studied to maximize the chance for identifying new issues;

4. The example includes multiple roles because our experience is that two-party protocols are seldom complex enough for the kind of machinery proposed here;

5. The example's model checking run times are not so large that they inhibit experimentation.

7.2 Software Development Protocol Description

We choose the following example taken from [Lomuscio et al., 2008a,b], referred to here as "SWDev" (SW development process) , with minor changes to role names. With the few modification described below, it meets all our selection desiderata. Lomuscio et al. describe the use case:

> In the example, the participating contract parties ... comprise: a principal software provider (PSP), a software provider (SP), a software client (CLIENT), an insurance company (INSURER), a testing agency (TESTER), a hardware supplier (HW), and a technical expert (EXPERT). The high-level workflow of the composition is defined as follows: CLIENT wants to get a software developed and deployed on hardware supplied by HW. To deploy the

software, EXPERT is needed. Components of the software are provided by different software providers. We consider two software providers here: PSP and SP. The components need to be integrated by the providers before the software is delivered to CLIENT.

The software integration is carried out by PSP, when SP delivers its component. PSP and SP twice update each other and CLIENT about the progress of the software development. Should the client like any changes in the software, he can request them before the second round of updates. Any change suggested by the client after the second update is considered a violation and the client is charged a penalty. The client can recover from this violation by paying the penalty or by withdrawing the request for changes. If PSP and SP do not send their updates as per schedule, this is also considered a violation and they are charged a penalty. Every update is followed by a payment in part by CLIENT to PSP. Payment to SP is handled by PSP and is done once the software is deployed successfully.

PSP integrates the components and sends the integrated component to TESTER for testing. Results from testing are made available to all the parties, i.e., PSP, SP, and CLIENT. If the integration test fails, the components are revised and tested again. Components can be revised twice. If the third test fails, CLIENT cancels the contract with PSP. If the testing succeeds, CLIENT invokes INSURER to get the software insured. CLIENT then invokes HW to order the hardware. Finally CLIENT invokes EXPERT to get the software deployed. If the software cannot be deployed then the hardware and the components have to be reevaluated. Components can be revised twice. If the third test fails CLIENT always cancels the contract with PSP and HW.

7.3 SWDev Modifications for Positron

The case study began with only the protocol description given in Section 7.2 to satisfy our "no detailed description" desideratum. The actual SWDev we used is a modification of that process, modified for the reasons describe here. These modifications suggest possible future enhancements to Positron.

SWDev frequently uses a "twice is OK; thrice is an error" (twice/thrice) pattern. This is the first example we encountered with this kind of pattern. Even though MCMAS supports integers and Positron uses MCMAS as a post-processor, Positron's input language does not support integers. The twice/thrice pattern could be implemented by creating distinct messages with embedded numberings (deliveryMsg1, deliveryMsg2, deliveryMsg3, ...), but that leads to an artificial and unnatural protocol style. Instead of implementing the twice/thrice pattern specified by the original description, we enable unbounded iteration and use a "once is OK; twice in an error" (once/twice) pattern to classify when subsequent iterations are in error.

Rather than the *protocol* requiring CLIENT to cancel the contract after a three of iterations, and

because Positron lacks integers, we allow CLIENT to decide when to cancel. This strategy fits with our experience that clients often have such clauses in their contracts, and that three iterations are seldom sufficient for SW development. CLIENT can pursue additional iterations as long as it willing to pay rework penalties.

Positron's lack of integers also prevents a reasonable implementation of a "schedule" for determining when PSP or SP are in error. We reinterpret this requirement to be: after an order is placed, PSP and SP must make a delivery *eventually*.

It is all too easy to construct model checker state spaces that are too large. An early version of our design contained 10^{66} states, far larger than the 10^{21} states reported in [Lomuscio et al., 2008b]. That version ran for 12 hours without completing, before we terminated the run. To achieve acceptable run times, the INSURER and HW roles and their functionality were eliminated because they are not central to the overall application and they bring out no new essential challenges. This and other changes significantly improved model checking performance, enabling greater experimentation.

SWDev describes payments and penalties for CLIENT, PSP, and SP, but it describes neither payments nor penalties for TESTER or EXPERT. One of our goals is the composition of reusable constituent protocols. We include payments and penalties for TESTER and EXPERT and this enabled us to reuse SP's protocol, which is desirable for reusability, is realistic from a business perspective, and provides additional function over the original SWDev.

The original specification explicitly states pricing and ordering are assumed to be successfully completed before SWDev starts. We added both pricing and ordering for all roles into our design because payments are central to many role requirements, protocol composition made it convenient and simple to add them, and it created additional hierarchical levels of composition (Figure 7.3).

7.4 Design and Composition

The design of SWDev follows the methodology in Section 4.3 and produces the CPD of Figure 7.2.

CM1 (Roles): There are five roles CLIENT (CL), PSP, SP, TESTER (TE), and EXPERT (EX). INSURER and HW were eliminated.

CM2 (Constituent Selection): Protocol design is a complex task, and much of that work occurs in this step. SWDev is more complicated than many previous protocols we have modeled. Figure 7.3 shows SWDev's hierarchical protocol structure. The highest level protocol is the composite *SWDev*.

We manually transformed our preexisting *Order* protocol into the simple protocols *Price*, *Pay*, and *Ship* (not shown). Next, we manually transformed *Ship* to *Serve* by converting the single shipMsg to enable an unbounded number of partial deliveries as required by SWDev.

We partitioned our preexisting *OrderPayShip* into protocols *Price* (messages reqQuote and quote), *Pay* (message pay), and *Ship* (message ship). Then *Ship* was transformed to *Serve* by enabling multiple, partial deliveries.

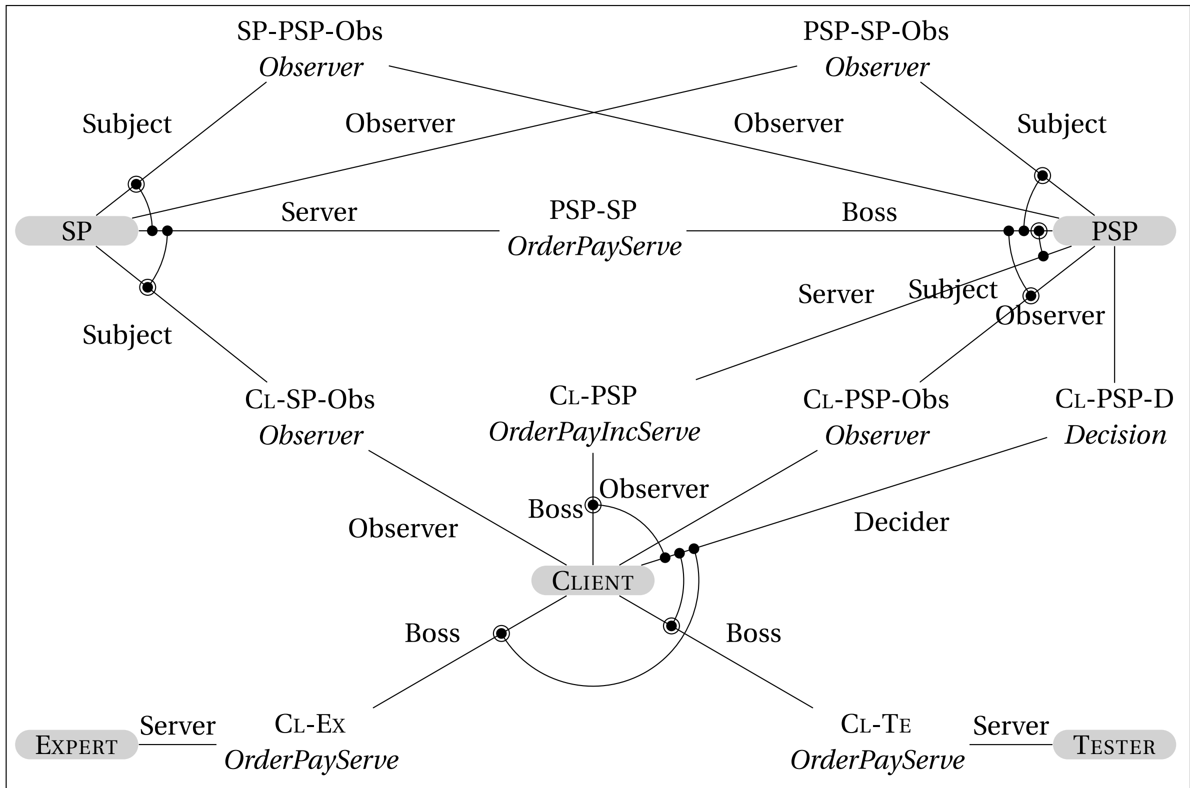


Figure 7.2: SWDev as a composite protocol diagram.

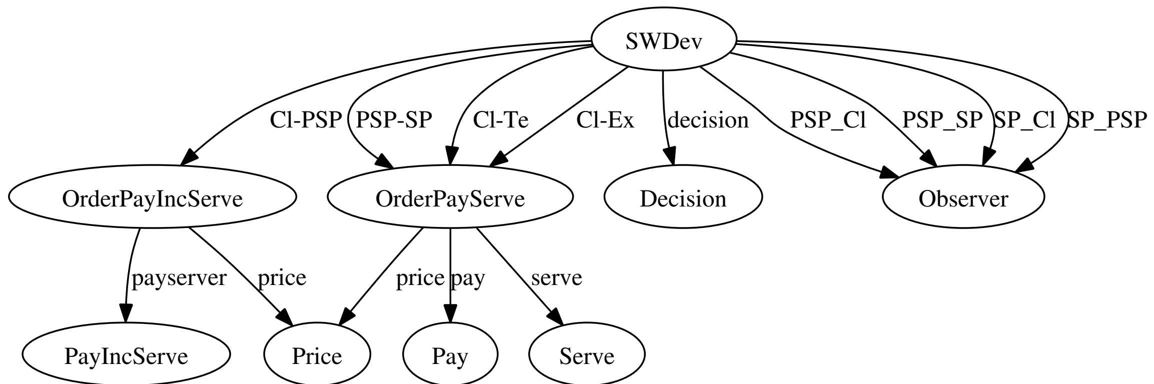


Figure 7.3: Nested protocol structure for SWDev composite.

Composite protocol *OrderPayServe* is composed from *Price*, *Pay*, and *Serve*. This single payment and unbounded deliveries protocol is used for PSP-SP, CL-TE, and CL-EX.

CLIENT makes a partial payment for each PSP partial delivery. An iterative version of *Pay* enabling multiple partial payments, paralleling *Serve* for deliveries, would have been cumbersome as the

payment and delivery messages are closely intertwined. Instead, we created protocol *PayIncServe* (incrementally pay and serve), which allows an unbound number of partial payment and partial delivery pairs. Composite *OrderPayIncServe* is composed from *Price* and *OrderPayServer*. CL-PSP uses the multipayment and multidelivery *OrderPayIncServe*.

Rather than burying notifications inside other protocols, we use the generally useful protocol *Observer* is a single message protocol between a SUBJECT role and a single OBSERVER role. We include it in the example even though it adds little to the overall example and somewhat clutters the CPD in Figure 7.2. In a real-world situation, such secondary constituents could be omitted from the diagram.

SWDev must atomically choose whether to start the request for quotation process, whether to place orders with the subcontractors, and whether to cancel all existing orders. This should be CLIENT's internal decision. In Positron, only messages can have meanings; there are no internal messages. A workaround was necessary. We introduce the artificial protocol *Decision* to make and record these atomic decisions. PSP is the arbitrary recipient of those messages but PSP ignores these messages. Future Positron support for internal actions could eliminate the need for *Decision*.

Finally, protocol *SWDev* is composed from *OrderPayIncServe*, *OrderPayServer*, *Decision*, and *Observer* constituents.

CM3 (Role Requirements): If CLIENT pays, it must receive either successful or failed deliverables from PSP, TESTER, and EXPERT. Similarly, if PSP pays, it requires deliverables from SP. Conversely, PSP, SP, TESTER, and EXPERT each require payment from CLIENT if they deliver their work. Without the extension of SWDev to include payments for TESTER and EXPERT, there are no natural role requirements for TESTER and EXPERT.

CM4 (Enactments): An exhaustive list of good and bad enactment requirements is not practical. We use the following enactments to ensure the complex composite protocol SWDev satisfies many of our expectations. We verify good paths: (1) the order decision is followed by order execution and deliveries, (2) deliveries and testing can occur repeatedly, (3) testing and deployments can occur repeatedly, (4) observer notifications can occur, (5) CLIENT, PSP, and SP penalties can occur, and (6) CLIENT can cancel. We additionally verify bad paths: (7) testing cannot occur before a delivery, and (8) deployment cannot occur before testing.

CM5 (Coupling Commitments): CLIENT couples the decisions it makes in *Decision* to the other ordering protocols. By slight abuse of notation and to minimize visual clutter, each CLIENT-centered arc in Figure 7.2 represents three coupling commitments: for the requestQuote, quote and cancel decisions. PSP couples those same requests from CL-PSP to PSP-SP. The remaining arcs, centered on PSP and SP, represent commitments to use *Observer*.

CM6 (Positron): The Positron source code was then written.

CM7 (Verification): The verification results for SWDev are reported in Table 7.1.

7.5 Requirement Changes

After a period of successful use of SWDev, assume CLIENT makes the following requirement changes:

1. CLIENT requires all deliverables (software, testing reports, and deployment reports) be stored in a common, central repository, accessible to all parties.
2. The current set of Observer constituent protocols ensure CLIENT, PSP, and SP are fully aware of all software deliveries, but TESTER and EXPERT are not. CLIENT requires all parties be informed.

These requirement changes are implemented by introducing a new REPOSITORY role to the design. Other roles now send their deliverable to REPOSITORY, who saves either the successful or failed deliverables in its internal central repository, and then notifies all other parties (excluding the sender) of the new deliverable. This satisfies both centralized storage of all deliverables, and notifying all roles. This makes the *Observer* protocol redundant, and it should be eliminated.

7.5.1 Implement Requirement Changes by Applying Refactorings

Apply the refactoring methodology in Section 6.4.1 to convert protocol *SWDev1* (P_1) to *SWDev2* (P_2). We apply the complete refactoring methodology twice: once for each requirement change.

Add Central Repository

The first complete application of the refactoring methodology adds the central repository. The involves adding REPOSITORY as a middleman to the current successful and failed delivery messages. We also add additional messages from REPOSITORY to the "observing" roles. In the following, Snd is the role making a delivery, succ is "succ" for successful deliverable or "fail" for a failed one, and Rcv is the role being notified. Snd and Rcv can be any role except REPOSITORY.

RM1 (Add Roles): The only role change is the addition of REPOSITORY.

- *Add Role*: REPOSITORY (RE).

RM3 (Add Messages): Add new notification messages between REPOSITORY and all the other roles.

- *Add Message*: for each [Snd, succ, Rcv] triple (excluding Snd=Rcv), add message "notify_-succSnd_Rcv_Msg" from REPOSITORY to role Rcv indicating that role Snd deposited a successful or failed (succ) deliverable.

RM4 (Add Middlemen): Add middleman REPOSITORY to all existing deliverable messages.

- *Add Middleman*: for each [Snd→ Rcv, succ] deliverable message pair, add REPOSITORY as a middleman.

These refactorings applied 33 distinct refactorings: one *Add Role* plus 8 (= 4 deliveries * 2 succ/fail) copies of one *Add Middleman* and four *Add Message* additional notifications.

Kill Observer Messages

We apply the complete refactoring methodology a second time to delete all the redundant *Observer* instances. Only a single step of the methodology is needed.

RM7 (Merge Messages): Publish kill messages to delete all Observer messages. Refactoring *Publish Kill* repeatedly applies *Push Kill*, *Move Kill*, and *Protocol Kill* to move kill message assertions throughout the interaction.

- *Publish Kill*: All Observer messages.

These refactorings applied 20 distinct refactorings: 4 copies of one *Push Kill*, one *Protocol Kill*, and multiple *Move Kill* refactorings.

7.5.2 Refactoring Summary

Altogether, 53 distinct refactoring are applied in the conversion of *SWDev1* to *SWDev2*. Appendix D shows the generated interceptor chains for each role.

Further *Agent Designer Independence* refactorings can be applied when each agent implementations is modified. In Appendix D, many reactions for a role have an empty do clause. These reactions consume the received message because that role does not (yet) accept that message. These reactions can be internalized into their agent's implementation using refactoring *Internalize Reaction*.

7.6 Results

Table 7.1 shows the Positron statistics for *SWDev1*, *SWDev2*, and *SWDevF* along side the results from the other large protocols from Table 4.2. *SWDevF* is identical to *SWDev1* but with an additional failing path enactment, to show the effect of verification failures on run time and space requirements.

SWDev1 uses 20 constituent instances as shown in Figure 7.3. *SWDev2* has no constituent instances because it is a fully expanded (flat) protocol. Compared to *SWDev1*, *SWDev2* has the additional REPOSITORY role, more Positron statements due to the additional messages, and otherwise the same counts. The state space sizes and run times are similar to each other. These two protocols are, for practical purposes, tied for second largest among all five protocols in the table. The counts for *SWDevF* are identical to those for *SWDev1*. The run times and state space size for *SWDevF* and *SWDev1* are essentially identical.

Proton verified three refinements with the results shown in Table 7.2. *SWDev2* refines *SWDev1* verifies that *SWDev2* does not enable any good paths that are not enabled by *SWDev1*. Protocol refinement is defined to be reflexive. Though not required, we demonstrate Proton's implementation works correctly, even on large protocols, by verifying both *SWDev1* refines*SWDev1* and *SWDev2* refines *SWDev2*.

Table 7.1: Protocol verification statistics from Positron for AGFIL (insurance), Quote To Cash (QTC) (manufacturing), ASPE (healthcare), and SWDev1, SWDev2, and SWDevF (software development). (M is 10^6 and G is 10^9.)

Composite Metric	**AGFIL**	**QTC**	**ASPE**	**SWDev1**	**SWDev2**	**SWDevF**
Constituent instances	11	6	12	20	–	20
Roles	6	6	5	5	6	5
Propositions	22	37	18	46	46	46
Commitments (all)	24	43	12	40	40	40
Coupling commitments	9	21	2	12	12	12
Messages	22	55	20	48	80	48
CTL formulas	9	17	14	29	29	29
Role requirements	8	13	7	10	10	10
Enactment requirements	1	4	7	19	19	20
Positron statements	94	164	81	188	201	188
State space size	120M	381G	1.47M	6.6G	6.6G	6.6G
Positron processing time	1.98s	3.16s	1.68s	1.08s	0.81s	1.09s
MCMAS processing time	4.29s	1274s	5.78s	53s	54s	54s
Total time	6.27s	1278s	7.46s	54s	55s	55s
All CTL formulas verified	✓	✓	✓	✓	✓	×

However, final checking of these results revealed both *SWDev1* and *SWDev2* violate detailed preconditions in an earlier proof. Specifically, these two protocols do not start in the empty state, and they do reset some propositional values, as they must to implement looping. We extended the proof to support non-empty initial states as shown in Appendix B. We believe the proof can be enhanced to address looping. But until then, even though Proton reports successful verification of the refinement conditions, refinement in SWDev is not demonstrated.

Protocol looping was the single most challenging design issue we encountered in SWDev. Designing Boolean propositions to ensure protocols looped correctly required significant design time and effort. In all other protocols we studied, multiple interactions (e.g., multiple insurance claims in AGFIL) are simply handled by different instances of a nonlooping protocol. Separate instances was not appropriate for SWDev, where there are interacting loops for development, testing, and deployment. Protocols *Server* and *PayIncServe* contains commitments to force movement around each loop iteration. The reworkMsg resets the commitments that have fired early in a loop interaction, so they become active for the next iteration. This challenge traces back to the fact that a commitment can only fire once, without being reset. A loop-friendly commitment definition is an interesting problem for future work.

We used an agile process, growing the design incrementally, alternating between Steps CM2 (Constituent Selection) and CM7 (Verification). We added single or related groups of enactments, and

Table 7.2: Refinement verification statistics from Proton. (M is 10^6 and G is 10^9.)

Refinement Metric	***SWDev1* refines *SWDev1***	***SWDev2* refines *SWDev1***	***SWDev2* refines *SWDev2***
Roles	5	6	6
Propositions	46	46	46
Commitments (all)	40	40	40
Messages	48	80	80
CTL formulas	153	153	153
Proton statements	81	181	201
State space size	212G	212G	425G
Proton processing time	1.25s	1.88s	1.10s
MCMAS processing time	65.83s	38.12s	878s
Total time	67.08s	40.00s	879s
All CTL formulas verified	✓	✓	✓
Satisfies preconditions	×	×	×

immediately resolved any model checking errors by adding coupling commitments.

The final constituent protocols have more parameters than originally expected. Many propositions needed to be shared between composite and constituent. Many propositions are global. The number of propositions is inflated partially because Positron uses only Boolean propositions; the number could be reduced if Positron supported integers or multifield structures, which convey more information per parameter. Unlike parameters in imperative programing languages, protocol parameters should not be viewed as unchanging inputs and outputs. They are constantly changing fluents that convey information in both directions across the composite/constituent boundary. The number of propositions is also inflated because Positron supports passing single propositions, but it does not support passing *expressions* of propositions. Valuable future Positron extensions include: a more general examination of general parameter passing strategies for constituents protocols; and a semantics for mutating (setting or clearing) expression like $p \vee q$ or $p \wedge q$.

Early in the design process, the protocols were designed with few ordering constraints. A common response to model checking errors was to eliminate unexpected bad paths found by the model checker, by surgically strengthening message guards. The frequent reoccurrence of such path reductions is consistent with our belief that our initial designs error on the side of too much flexibility. We desire to reduce flexibility only to satisfy requirements.

Writing and debugging enactment expressions can be challenging. We ran into unexpected model checking failures, including (1) checking state variables (which remain true for extended periods) rather than checking message send occurrences, and (2) checking for message occurrences rather

than *successful* message occurrences. Our role requirement expressions (Section 4.2.2) and path expressions (Section 4.2.3) were not sufficient for expressing all requirements in SWDev. Two checks could not be captured using the functions we have described so far: no testing before SW delivery, and no deployments before testing. For these, we implemented the *before* specification, based on Marengo [2013]: if q occurs, then it must be preceded by p.

$$\mathit{before}(p,q) := \neg\mathbf{E}(\neg p\,\mathbf{U}\,q) \tag{7.1}$$

Attie et al. [1993], Singh et al. [2003], and Marengo et al. [2011] proposed an alternative *before* operator using an event-based logic rather than the state-based logic we use here. Their before operator ($p \cdot q$) allows q to occur with or without a preceeding p.

Automated refactorings are fast. A Java program applied the sequence of 53 refactorings described above, generating 76 interceptors, in two seconds of processing running on a 2.3 GHz Intel Core i7 processor.

7.7 Evaluation

The SWDev case study met all of our example desiderata in Section 7.1. It also uncovered a number of weaknesses as well as demonstrated many strengths of both Proton and Positron.

7.7.1 Weaknesses Uncovered

SWDev uncovered a number of Positron limitations. Protocol design can be tedious, and further improvements are needed. The single biggest challenge encountered in the SWDev exercise was Positron's need for improved support for looping protocols. This Positron limitation complicated the SWDev implementation requiring intricate proposition assignments to implement messaging loops and recurring commitments. Telang et al. [2013] propose maintenance commitments to address recurring commitments. They would be a valuable concept to incorporate in our future work.

SWDev required all interconstituent message ordering capabilities to be "built into" the constituent protocols. A valuable Positron addition would be generic enablement for interconstituent message ordering. Others have proposed mechanisms to address message ordering. Desai and Singh [2007] propose message ordering axioms, but it does not directly accomodate looping protocols.

Positron needs support for bounded integers, a common requirement for looping protocols. This precluded SWDev implementing the "twice/thrice" patterns, maximum number of rework phases before cancellation, and the "scheduling" requirement.

Better approaches and best practices are desired to reduce the number of parameters on constituent protocols. These will naturally emerge over time with continued use.

Positron currently has an incomplete implementation for message parameters, specified after the message name. For example,

Re → PSP   notifyMsg**(sender, success)**   means { notified.set }

Such support would have made little difference in the protocols we studied previously, but would have enabled two simplifications in SWDev. Protocols *Serve* and *PayIncServe* required distinct messages for successful and failed deliverables (software, testing, or deployment) because they had different meanings. A similar problem occurs in REPOSITORY's notification messages. *SWDev2* has distinct messages for each [Snd, succ, Rcv] triple. Passing Snd and succ as message parameters would have markedly reduced the number of messages required to implement notification. These messages could not be combined because the current Positron implementation for messages does not support parameter-dependent meanings.

Both Proton and Positron generates many MCMAS variables, generating large model checking state spaces. A more careful analysis of model generation might identify better variable encodings, reducing state space size and making larger models tractable.

7.7.2 Strengths Demonstrated

SWDev demonstrated Positron's many strengths. The CPD for SWDev in Figure 7.2 concisely describes many important, high-level features of the composite protocol for SWDev: (1) all roles, (2) all high level (but not deeply nested) constituent protocols, (3) all constituent roles enacted by each composite role, (4) all inter-role communication pathways as lines, and (5) all high level (but not deeply nested) role responsibilities as coupling commitment arcs.

We successfully composed a protocol to implement all essential elements of SWDev. Plus, we extended the original definition with additional functionality including support for ordering and payment, and deliverable notifications to and from TESTER and EXPERT.

The original SWDev specification was silent on why each role would even agree to participate in the protocol at all. We made each role's requirements explicit, demonstrated they were straightforward to express, and demonstrated both *SWDev1* and *SWDev2* protocols satisfy those role requirements.

Coupling commitments constraint each role's external behavior without constraining any agent's internal implementation, preserving agent autonomy.

Expressing SWDev role and path requirements, using using our Req and path expressions, allowed us to design at a high conceptual level. This eliminated the need to write and debug many long and complicated CTL formulas.

While designing and verifying the original SWDev protocol was time consuming, automatic refactorings significantly reduced the time to construct and successfully verify *SWDev2*.

Proton and Positron checking are complementary. Positron's role and enactment requirements check specific features of a protocol, but are not exhaustive. Proton's checking checks all paths (runs)

between a putative superprotocol and subprotocol, but does not support expression of specific path checks.

Chapter 8

Conclusions

Section 8.1 assert our major claims and their supporting evidence. Section 8.2 summarizes our contributions. Finally, Section 8.3 outlines directions for future work.

8.1 Claims

We assert the following major claims and supporting evidence.

Claim 1 Positron successfully composes and verifies protocols against role requirements and enactments.

We defined protocols and protocol composition. Role accountabilities describe the interconstituent actions roles must take. Role responsibilities and enactments describe properties agent and protocol designers require of the protocol. We successfully expressed and verified four case studies from the literature (Sections 4.4.1, 4.4.2, and 4.4.3, and Chapter 7). All case studies were realistic and non-trivial protocols from different domains.

Claim 2 Proton provides an implementable definition of protocol refinement, between a putative subprotocol and a putative superprotocol.

We implemented our protocol refinement definition as CTL, which can be evaluated by a model checker (Section 5.3), and proved the CTL is equivalent to our definition (Section 5.5). We demonstrated expected and reasonable refinements on ten Pay protocols (Figure 5.1 and Section 5.4).

Claim 3 We define an interaction architecture and demonstrate requirements evolution via interaction refactorings (Rho).

We demonstrated protocol Pay can be refactored to PayViaCheck (Section 6.4.2), and demonstrated interaction-wide propagation of changed guards (Section 6.4.3). Our JADE imple-

mentation demonstrated two agents using three concurrent instances of protocol Pay, while they actually exchange messages from PayViaMM (Section 6.5). We demonstrated protocol SWDev1 can be refactored to SWDev2 (Section 7.5.1). In the last two examples, the protocols were refactored programatically.

Claim 4 Positron, Proton, and Rho express and verify end-to-end, realistic examples.

Where Claim 1 addresses only composition, this claim encompasses all elements of our approach, including the interfaces between elements. In spite of the problems encountered in SWDev, our approach successfully expressed and verified the SWDev software development case study (Chapter 7). It covered composite protocol design, single-protocol verification (*SWDev1*), protocol evolution due to changing requirements (*SWDev2*), single-protocol verification of the refactored version (*SWDev2*), and refinement checking between them.

8.2 Summary of Contributions

We now summarize the main contributions of this dissertation.

Define the protocol refinement relation between a putative superprotocol and a putative subprotocol (Section 5.2), define its implementation as CTL formulas (Section 5.3), and prove the definition of protocol refinement holds if and only if its implementation in CTL holds (Section 5.5 and Appendix B). Extend commitments to enable sets of debtors and creditors (Section 2.2). Define serial composition of two commitments and proved it is idempotent, not commutative, and not associative (Section 3.2.1). Define scalar serial composition of an expression and a commitment (Section 3.2.2). Defined the commitment covering relation between two commitments (Section 3.2.3).

Implement a decision procedure and mechanical verification of protocols with respect to role requirements, role accountabilities, enactments, and compiling formulas to temporal logic by employing the MCMAS model checker to verify whether the composite protocol satisfies those formulas (Section 4.2.5). Define protocol composition, based on role specific concepts and high-level verification functions (Section 4.2.1). Describe composite protocol diagrams (CPD) as a graphical notation, conveying important features of the composite protocol to business and technical stakeholders (Section 4.2.6). Define role requirements, a high-level function to express such requirements, and its expansion to CTL (Section 4.2.2). Define role accountabilities as coupling commitments (Section 4.2.4). Define high-level enactment verification functions and their expansion to CTL (Section 4.2.3).

Define an architecture for evolving requirements (Section 6.3.1) by using refactorings (Section 6.3.3). Identify three types of interaction refactorings: *Protocol Designer Independence, Agent Designer Independence*, and *Designer Collaboration* (Section 6.3.3). Define the Rho library of interaction refactorings (Section 6.3.3 and Appendix C).

Evaluation of Positron's composition on three realistic protocols from insurance (AGFIL), manufacturing (Quote To Cash), and healthcare (ASPE) (Section 4.4). End-to-end evaluation of all our methods and tooling on the realistic SWDev (software development) example, identifying both strengths and weaknesses (Section 7).

8.2.1 Protocol Composition

Although protocols offer significant benefits over traditional approaches, protocols are not fully viable for the following reasons. One, specifying in one shot an adequate protocol for a complex scenario is nontrivial. Two, implementing agents who can play roles in such a comprehensive protocol is difficult because the differing details of the protocols complicate reusing parts of agent implementations. Our contribution is to show how complex protocols can be constructed by composing existing protocols. Previous relevant research falls into these categories: (a) commitments but not composition [Gerard and Singh, 2013]; (b) composition but no commitments [Miller and McGinnis, 2008; Singh, 2011]; and (c) composition and commitments. The last category can be categorized as (c1) purely abstract description without a specification language or tools [Mallya and Singh, 2007]; (c2) composition of commitment-based protocols based on regulative constraints [Marengo, 2013]; and (c3) our approach to composition of commitment-based protocols based on role responsibilities and accountabilities.

Our approach, *Positron*, extends our Proton approach to provide a clear syntax and semantics for composite protocols. Where Proton checks protocol refinement, Positron composes protocols. Positron (a) recursively expands nested constituent protocols; (b) introduces *composite protocol diagrams* as a graphical notation, conveying important features of the composite protocol to business and technical stakeholders; (c) introduces *role requirements* and *role accountabilities*; (d) incorporates a methodology for composing commitment protocols; and (e) implements a decision procedure and mechanical verification of protocols with respect to role requirements, role accountabilities, enactments, and compiling formulas to temporal logic, and employing MCMAS [Lomuscio et al., 2009], a leading model checker, to verify the composite protocol satisfies those formulas.

8.2.2 Refinement

We formulate refinement in technical terms and show how to compute it via a tool called *Proton*. We specify a protocol declaratively in terms of (a) its roles, (b) the guarded messages the roles exchange, and (c) the meaning of each message as a set of actions on the public state of the roles, sometimes termed the *social state* [Baldoni et al., 2010a]. Commitments between roles are central to our approach [Singh, 1999].

We define the semantics of a protocol precisely in terms of the runs (i.e., sequences of actions) it allows. Informally, a *subprotocol* refines a *superprotocol* if and only if the latter allows all the runs the former allows. However, refinement is nontrivial because the protocols may involve different roles

and messages, the messages may have different meanings, and the meanings may be at different levels of abstraction. Hence, we define refinement only with respect to a mapping of meanings from the superprotocol to the subprotocol. For example, the payment in *Pay* maps to two payments in *PayViaMM*.

Our approach for verifying refinement takes three inputs: formal descriptions of a putative superprotocol and subprotocol, and a mapping between them. We reduce the protocol descriptions to their canonical forms, taking into account the mapping provided. We generate an input to an existing model checker consisting of (a) a specification of a temporal logic model and (b) temporal formulae whose truth in the model verifies refinement.

One, we offer the *first* approach that computes the refinement for protocols based on static analysis of protocol specifications. Two, we formulate a notion of the serial composition of commitments, which can have broader applications than this paper, e.g., in the treatment of commitments in coalitions.

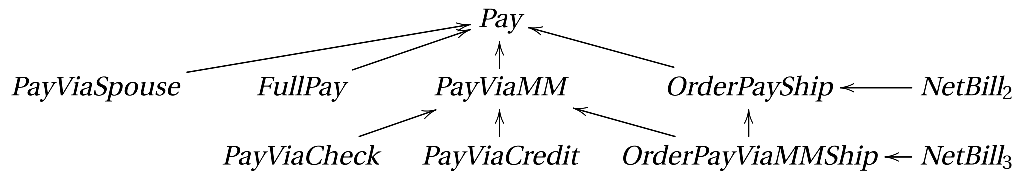


Figure 8.1: Refinements demonstrated by Proton (arrows point from subprotocols to superprotocols).

Further, we have implemented our approach in the Proton tool that overlays the well-known model checker MCMAS (http://www-lai.doc.ic.ac.uk/mcmas/). Figure 8.1 summarizes ten protocol refinements that Proton verifies (under the obvious mappings) based on the above and other examples known from the literature.

8.2.3 Interaction Refactorings

Our main contribution is the concept of *interaction refactorings* that enable independent and incremental evolution of interactions, decoupling the efforts of agent and protocol designers.

We identify and describe three forms of requirement evolution: *Protocol Designer Independence* (PDI), *Agent Designer Independence* (ADI), and *Designer Collaboration* (DC). Each focuses on different parts of an interaction, two provide designer isolation, and one enables designer collaboration. We describe refactorings for all three forms. Applying refactoring from all three forms, in concert, supports interaction-wide evolution. Interceptors and interceptor chains are the critical elements that enable refactorings.

We demonstrated refactorings to transform *Pay* into *PayViaCheck*, without changing agent implementations. We also demonstrated an agent voluntarily restricting its behavior (*payTC*) and propagating that change throughout the interaction.

We have our Rho library, with 30 refactorings. A JADE prototype demonstrates basic interceptor chain functionality, refactorings automatically generating reactions, and agent interoperability after refactoring to a new protocol.

We adopt a reaction-based, interceptor chain architecture that is effective and yet simple enough to yield refactorings that are easy to understand and apply. Because interceptors are predefined and simple, we can define refactorings to mechanically evolve them. Mechanical evolution of general-purpose agent implementation is likely intractable.

Refactorings clearly communicate interaction changes. We mechanically transform refactorings into sets of interceptors. Interceptor chains can store important pieces of state (e.g., an agent's checking account information) and can make commitments on behalf of the agent (e.g., committing to redeem valid checks). In this case, the interceptor chain becomes a trustee of the agent, sometimes a necessity when unmodified agents participate in new protocols. However, it also raises autonomy concerns about the interceptor chain. Agents should be able to limit the trust and autonomy they grant to their interceptor chains.

8.2.4 Case Study

The end-to-end case study incorporated all aspects of our approach: composing protocols from constituent protocols, constructing a Composite Protocol Diagram, refactoring the original *SWDev1* protocol to *SWDev2* based on changed requirements, and demonstrating all good paths of the refactored protocol are also good paths of the original protocol. The case study uncovered a number of weaknesses as well as demonstrated many strengths of both Proton and Positron. We successfully completed the SWDev case study in spite of the problems encountered.

8.3 Future Work

Whereas this research has a strong theoretical focus, necessary for a solid foundation, we desire to increase future focus on the practical challenges surrounding protocols. We see three general areas for future work.

Expressiveness

The first area extends *expressiveness*. Investigate the extent to which our methods and tools can express, regardless of user sophistication, realistic and real-world business cases. A key area is better

support for looping protocols. Mechanisms are needed to express the construction of looping protocols and their message guard conditions. The commitments in Section 2.2 are inherently non-looping: once a commitment is satisfied, it is complete. We had to explicitly reset commitments for SWDev. Adopting maintenance commitments from Telang et al. [2013] would likely better express the commitments found in looping protocols. The work of Marengo [2013] would also substantially extend the expressiveness of enactment checking, and the addition of "scopes" [Dwyer et al., 1999] would allow direct support for looping.

Other extensions would also increase expressiveness: (1) support for Positron integers to handle patterns like "twice/thrice", (2) support for inter-constituent message ordering similar to that in Desai [2009], (3) support for passing argument *expressions* to constituent protocols, and (4) support for parameter-dependent meanings in messages.

Libraries

The second area is to continue expanding our *libraries* with the identification and capture of additional constituent protocols for inclusion in our protocol library, and additional interaction refactorings for inclusion in our Rho refactoring library. New case studies will naturally suggest new constituent protocols, new refactorings, and additional challenges.

Usability

The third area explores *usability*. Investigate the issues with, and the extant to which, industry users can apply our methods and tools including protocol construction, CPD construction, writing role requirements, writing role accountabilities (coupling commitments), writing precise enactment specification, and interpreting model checking failures. This applies to both business and multiagent applications.

References

Rajeev Alur, Thomas A. Henzinger, and Orna Kupferman. Alternating-time temporal logic. *Journal of the ACM*, 49(5):672–713, September 2002. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/585265.585270.

ASPE. The importance of radiology and pathology communication in the diagnosis and staging of cancer: Mammography as a case study, November 2010. Office of the Assistant Secretary for Planning and Evaluation, U.S. Department of Health and Human Services; available at http://aspe.hhs.gov/sp/reports/2010/PathRad/index.shtml.

Paul C. Attie, Munindar P. Singh, Amit P. Sheth, and Marek Rusinkiewicz. Specifying and enforcing intertask dependencies. In *Proceedings of the 19th International Conference on Very Large Data Bases (VLDB)*, pages 134–145, August 1993.

Matteo Baldoni, Cristina Baroglio, Amit K. Chopra, Nirmit Desai, Viviana Patti, and Munindar P. Singh. Choice, interoperability, and conformance in interaction protocols and service choreographies. In *Proceedings of the 8th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, pages 843–850, Budapest, May 2009. IFAAMAS.

Matteo Baldoni, Christina Baroglio, and Elesa Marengo. Constraints among commitments: Regulative specification of interaction protocols. In *Proceedings of the International Workshop on Agent Communication*, pages 10–29, Toronto, 2010a.

Matteo Baldoni, Cristina Baroglio, and Elisa Marengo. Behavior-oriented commitment-based protocols. In *Proceedings of the 19th European Conference on Artificial Intelligence (ECAI)*, pages 137–142, 2010b.

Fabio Bellifemine, Giovanni Caire, and Dominic Greenwood. *Developing Multi-agent Systems with JADE*. Wiley, West Sussex, England, 2007.

Sinead Browne and Michael Kellett. Insurance (motor damage claims) scenario. Document D1.a, CrossFlow Consortium, 1999.

Christopher Cheong and Michael P. Winikoff. Hermes: Designing flexible and robust agent interactions. In Virginia Dignum, editor, *Handbook of Research on Multi-Agent Systems: Semantics and Dynamics of Organizational Models*, chapter 5, pages 105–139. IGI Global, Hershey, PA, 2009.

Amit K. Chopra and Munindar P. Singh. Multiagent commitment alignment. In *Proceedings of the 8th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, pages 937–944, Budapest, May 2009. IFAAMAS.

Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, MA, 1999.

Mika Cohen, Mads Dam, Alessio Lomuscio, and Francesco Russo. Abstraction in model checking multi-agent systems. In *Proceedings of the 8th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, pages 945–952, Budapest, 2009. IFAAMAS. ISBN 978-0-9817381-7-8.

Nirmit Desai and Munindar P. Singh. A modular action description language for protocol composition. In *Proceedings of the 22nd Conference on Artificial Intelligence (AAAI)*, pages 962–967, Vancouver, July 2007. AAAI Press.

Nirmit Desai, Ashok U. Mallya, Amit K. Chopra, and Munindar P. Singh. Interaction protocols as design abstractions for business processes. *IEEE Transactions on Software Engineering*, 31(12): 1015–1027, December 2005.

Nirmit Desai, Zhengang Cheng, Amit K. Chopra, and Munindar P. Singh. Toward verification of commitment protocols and their compositions. In *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, pages 33:1–33:3. ACM, 2007.

Nirmit Desai, Amit K. Chopra, and Munindar P. Singh. Amoeba: A methodology for modeling and evolution of cross-organizational business processes. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 19(2):6:1–6:45, October 2009.

Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In Barry W. Boehm, David Garlan, and Jeff Kramer, editors, *ICSE*, pages 411–420. ACM, 1999. ISBN 1-58113-074-0.

Mohamed El-Menshawy, Jamal Bentahar, Wei Wan, and Rachida Dssouli. Verifying conformance of commitment protocols via symbolic model checking. In *Proceedings of the International Workshop on Agent Communication*, pages 53–72, Toronto, 2010.

Thomas Erl. *SOA Design Patterns*. Prentice Hall, Boston, 2008. ISBN 0-13-613516-1.

Ronald Fagin, Joseph Y Halpern, Yoram Moses, and Moshe Y Vardi. *Reasoning about Knowledge*. MIT Press, Cambridge, MA, USA, 1995. ISBN 0-262-56200-6.

Despina Filippidou. Designing with scenarios: A critical review of current research and practice. *Requirements Engineering*, 2(1):1–22, March 1998.

James A. Fitzsimmons and Mona J. Fitzsimmons. *Service Management: Operations, Strategy, Information Technology*. McGraw-Hill, New York, NY, USA, 6th edition, 2008. ISBN 978-0-07-722849-1.

Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, Boston, 2000. ISBN 0-201-48567-2.

Dov Gabbay. The declarative past and imperative future: Executable temporal logic for interactive systems. In *Proceedings of the Colloquium on Temporal Logic in Specification*, volume 398 of *LNCS*, pages 409–448. Springer-Verlag, 1987.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, Reading, MA, 1995.

Scott N. Gerard and Munindar P. Singh. Formalizing and verifying protocol refinements. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 4(2), 2013.

Scott N. Gerard, Pankaj R. Telang, Anup Kalia, and Munindar P. Singh. Positron: Composing commitment protocols. In *Proceedings of the First International Workshop on Engineering Multiagent Systems (EMAS)*, pages 1–16, St. Paul, MN, 2013.

Paul Grefen, Rik Eshuis, Nikolay Mehandjiev, Giorgos Kouvas, and George Weichhart. Internet-based support for process-oriented instant virtual enterprises. *Internet Computing, IEEE*, 13(6):65–73, 2009. ISSN 1089-7801. doi: 10.1109/MIC.2009.96.

Akın Günay, Michael Winikoff, and Pınar Yolum. Commitment protocol generation. In *Proceedings of the 10th AAMAS Workshop on Declarative Agent Languages and Technologies (DALT)*, pages 51–66, 2012.

HL7. Health Level Seven, 2007. http://www.hl7.org.

Alessio Lomuscio and Franco Raimondi. Model checking knowledge, strategies, and games in multi-agent systems. In *Proceedings of the 5th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 161–168, Hakodate, Japan, 2006. ACM. ISBN 1-59593-303-4. doi: http://doi.acm.org/10.1145/1160633.1160660.

Alessio Lomuscio, Hongyang Qu, and Monika Solanki. Towards verifying contract regulated service composition. In *Web Services, 2008. ICWS '08. IEEE International Conference on*, pages 254–261, September 2008a. doi: 10.1109/ICWS.2008.115.

Alessio Lomuscio, Hongyang Qu, and Monika Solanki. Towards verifying compliance in agent-based web service compositions. In *AAMAS: Proceedings of the 7th International Conference on Autonomous Agents and Multiagent Systems*, pages 265–272, Richland, SC, 2008b. International Foundation for Autonomous Agents and Multiagent Systems. ISBN 978-0-9817381-0-9.

Alessio Lomuscio, Hongyang Qu, and Franco Raimondi. MCMAS: A model checker for the verification of multi-agent systems. In *Proceedings of the International Conference on Computer Aided Verification*, LNCS, pages 682–688, 2009.

Alessio Lomuscio, Hongyang Qu, and Monika Solanki. Towards verifying contract regulated service composition. *Journal of Autonomous Agents and Multi-Agent Systems*, 24(3):345–373, May 2012.

Ashok U. Mallya and Munindar P. Singh. An algebra for commitment protocols. *Journal of Autonomous Agents and Multi-Agent Systems*, 14(2):143–163, April 2007.

Thomas W. Malone, Kevin Crowston, and George A. Herman, editors. *Organizing Business Knowledge: The MIT Process Handbook*. MIT Press, Cambridge, MA, 2003.

Elisa Marengo. *2CL Protocols: Interaction Patterns Specification in Commitment Protocols*. PhD thesis, Universitá Degli Studi di Torino, February 2013.

Elisa Marengo, Matteo Baldoni, Amit K. Chopra, Cristina Baroglio, Viviana Patti, and Munindar P. Singh. Commitments with regulations: Reasoning about safety and control in REGULA. In *Proceedings of the 10th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, pages 467–474, Taipei, May 2011. IFAAMAS.

Peter McBurney and Simon Parsons. Games that agents play: A formal framework for dialogues between autonomous agents. *Journal of Logic, Language and Information*, 11:315–334, Summer 2002. ISSN 0925-8531. doi: 10.1023/A:1015586128739. URL http://portal.acm.org/citation.cfm?id=595851.596051.

Jarred McGinnis and David Robertson. Dynamic and distributed interaction protocols. In *Adaptive Agents and Multi-Agent Systems*, volume 3394 of *Lecture Notes in Computer Science*, pages 167–184. Springer, 2005.

Tim Miller and Peter McBurney. Propositional dynamic logic for reasoning about first-class agent interaction protocols. *Computational Intelligence*, 27(3):422–457, 2011.

Tim Miller and Jarred McGinnis. Amongst first-class protocols. In *Proceedings of the 8th International Workshop on Engineering Societies in the Agents World (ESAW 2007)*, volume 4995 of *LNCS*, pages 208–223. Springer, 2008.

Robert M Monczka, Robert B Handfield, Larry C Giunipero, and James L Patterson. *Purchasing and Supply Chain Management*. South-Western, Cengage Learning, Mason, OH, USA, 2011. ISBN 978-0-538-47642-3.

Timothy J. Norman and Chris Reed. Group delegation and responsibility. In *Proceedings of the 1st International Joint Conference on Autonomous Agents and Multiagent Systems*, AAMAS, pages 491–498, Bologna, 2002. ACM. ISBN 1-58113-480-0. doi: http://doi.acm.org/10.1145/544741.544856. URL http://doi.acm.org/10.1145/544741.544856.

James Odell, H. Van Dyke Parunak, and Bernhard Bauer. Extending UML for agents. In *Proceedings of the Agent Oriented Information Systems Workshop at the 17th National Conference on Artificial Intelligence (AAAI)*, 2000.

OMG. Business process model and notation (BPMN), version 2.0 beta, June 2010. Object Management Group. http://bpmn.org/.

OMG. *Business Process Model and Notation (BPMN)*. Object Management Group, 2011. URL http://www.omg.org/spec/BPMN/2.0.

Oracle. Automating the Quote-to-Cash process, 2009. http://www.oracle.com/us/industries/045546.pdf.

H. Van Dyke Parunak. Visualizing agent conversations: Using enhanced Dooley graphs for agent design and analysis. In *Proceedings of the 2nd International Conference on Multiagent Systems*, pages 275–282, Kyoto, 1996. AAAI Press.

José Quenum, Aurélien Slodzian, and Samir Aknine. Automatic derivation of agent interaction model from generic interaction protocols. In *Agent-Oriented Software Engineering IV*, volume 2935, pages 193–229. Springer Berlin / Heidelberg, 2004. URL http://dx.doi.org/10.1007/978-3-540-24620-6_10. 10.1007/978-3-540-24620-6_10.

Nicholas Rescher. Collective responsibility. *Journal of Social Philosophy*, 29(3):46–58, December 1998.

William N. Robinson and Sandeep Purao. Specifying and monitoring interactions and commitments in open business processes. *IEEE Software*, 26(2):72–79, March 2009.

RosettaNet. Home page, 2009. http://www.rosettanet.org.

Sharmila Savarimuthu and Michael Winikoff. Mutation operators for cognitive agent programs. In *Proceedings of the First International Workshop on Engineering Multiagent Systems (EMAS)*, pages 286–301, St. Paul, MN, 2013.

Ricardo Seguel, Rik Eshuis, and Paul Grefen. Constructing minimal protocol adaptors for service composition. In *WEWST: Proceedings of the 4th Workshop on Emerging Web Services Technology*, pages 29–38, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-776-9. doi: http://doi.acm.org/10.1145/1645406.1645411.

Ricardo Seguel, Rik Eshuis, and Paul Grefen. Generating minimal protocol adaptors for loosely coupled services. *2012 IEEE 19th International Conference on Web Services*, 0:417–424, 2010. doi: http://doi.ieeecomputersociety.org/10.1109/ICWS.2010.14.

Constantin Serban and Naftaly H. Minsky. In vivo evolution of policies that govern a distributed system. In *POLICY*, pages 134–141, 2009.

Munindar P. Singh. An ontology for commitments in multiagent systems: Toward a unification of normative concepts. *Artificial Intelligence and Law*, 7(1):97–113, March 1999.

Munindar P. Singh. Distributed enactment of multiagent workflows: Temporal logic for service composition. In *Proceedings of the 2nd International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, pages 907–914, Melbourne, July 2003. ACM Press.

Munindar P. Singh. Semantical considerations on dialectical and practical commitments. In *Proceedings of the 23rd Conference on Artificial Intelligence*, pages 176–181. AAAI Press, 2008. ISBN 978-1-57735-368-3.

Munindar P. Singh. Information-driven interaction-oriented programming: BSPL, the Blindingly Simple Protocol Language. In *Proceedings of the 10th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, pages 491–498, Taipei, May 2011. IFAAMAS.

Munindar P. Singh and Amit K. Chopra. Correctness properties for multiagent systems. In *Proceedings of the 6th AAMAS Workshop on Declarative Agent Languages and Technologies (DALT 2009)*, volume 5948 of *LNAI*, pages 192–207, Budapest, 2010. Springer.

Howard Smith and Peter Fingar. *Business Process Management: The Third Wave*. Megan-Kiffer Press, Tampa, 2002.

Jim Spohrer, Paul P. Maglio, John Bailey, and Daniel Gruhl. Steps toward a science of service systems. *Computer*, 40(1):71 –77, January 2007. ISSN 0018-9162. doi: 10.1109/MC.2007.33.

Java Servlet Specification. Sun Microsystems, 3.0 edition, December 2009.

Pankaj R. Telang and Munindar P. Singh. Specifying and verifying cross-organizational business models: An agent-oriented approach. *IEEE Transactions on Services Computing*, 5(3):305–318, July 2012.

Pankaj R. Telang, Neil Yorke-Smith, and Munindar P. Singh. Maintenance commitments and goals. unpublished, 2013.

Steve Vinoski. Chain of responsibility. *IEEE Internet Computing*, 6(6):80–83, 2002.

Mingzhong Wang, Kotagiri Ramamohanarao, and Jinjun Chen. Reasoning intra-dependency in commitments for robust scheduling. In *AAMAS: Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems*, pages 953–960, Budapest, 2009. IFAAMAS. ISBN 978-0-9817381-7-8.

Michael Winikoff. Designing commitment-based agent interactions. In *Proceedings of the IEEE/WIC/ACM International Conference on Intelligent Agent Technology*, pages 363–370, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2748-5. doi: http://dx.doi.org/10.1109/IAT.2006.53.

Michael Winikoff. Implementing commitment-based interactions. In *Proceedings of the 6th International Conference on Autonomous Agents and Multiagent Systems*, pages 1–8, Honolulu, 2007. ACM. ISBN 978-81-904262-7-5. doi: http://doi.acm.org/10.1145/1329125.1329283.

WS-CDL. Web services choreography description language version 1.0, November 2005. http://www.w3.org/TR/ws-cdl-10/.

Pınar Yolum. Design time analysis of multiagent protocols. *Data and Knowledge Engineering Journal*, 63:137–154, 2007.

Pınar Yolum and Munindar P. Singh. Commitment machines. In *Proceedings of the 8th International Workshop on Agent Theories, Architectures, and Languages (ATAL 2001)*, volume 2333 of *LNAI*, pages 235–247, Seattle, 2002. Springer.

Appendices

Appendix A

Proton Source Code

Listing A.1 Pay Protocol

```
1: protocol Pay { {
2:     role
3:         Payer;
4:         Payee;
5:     prop
6:         promise;
7:         pay;
8:     commitment
9:         scpay = C_{Payer,Payee}(promise.isSet(), pay.isSet());
10:    message
11:        Payer → Payee : [true] promiseMsg means {promise.set(), scpay.create()};
12:        Payer → Payee : [promise.isSet() ∧ scpay.isCreate()] payMsg means {pay.set()};
13: }
```

Listing A.2 PayViaMM Protocol

```
1: protocol PayViaMM { {
2:     role
3:         Payer;
4:         MM;
5:         Payee;
6:     prop
7:         promise;
8:         payP;
9:         payM;
10:    commitment
11:        scpayP = C_{Payer,Payee}(promise.isSet(), payP.isSet());
12:        scpayM = C_{MM,Payer}(payP.isSet(), payM.isSet());
13:    message
14:        MM → Payer : [true] pledgeMsg means {scpayM.create()};
15:        Payer → Payee : [true] promiseMsg means {promise.set(), scpayP.create()};
16:        Payer → MM : [promise.isSet()∧scpayP.isCreate()∧scpayM.isCreate()] payPMsg means {payP.set()};
17:        MM → Payee : [true] payMMsg means {payM.set()};
18: }
```

Listing A.3 *PayViaCheck* Protocol

1: role
2:     *Payer*
3:     *Bank*;
4:     *Payee*;
5: prop
6:     *acct*;
7:     *deposit*;
8:     *choose*;
9:     *check*;
10:     *redeem*;
11:     *payB*;
12: commitment
13:     $\mathsf{C}_{payB} = \mathsf{C}(Payer, Payee, deposit \wedge choose, check)$;
14:     $\mathsf{C}_{bank} = \mathsf{C}(Bank, Payer, deposit \wedge check \wedge redeem, payB)$;
15:     $\mathsf{C}_{redeem} = \mathsf{C}(Payee, Bank, deposit \wedge check, redeem)$;
16: message
17:     $Payer \rightarrow Payee$: [*acct*] *chooseMsg* means {*choose*, CREATE(C_{payB})};
18:     $Payer \rightarrow Bank$: *openMsg* means {*open*};
19:     $Bank \rightarrow Payer$: [*open*] *acctMsg* means {CREATE(C_{bank}), CREATE(C_{redeem})};
20:     $Payer \rightarrow Bank$: *depositMsg* means {*deposit*};
21:     $Bank \rightarrow Payer$: [*deposit*] *confirmMsg* means {};
22:     $Payer \rightarrow Payee$: [$acct \wedge choose \wedge$ CREATE(C_{payB}) $\wedge$ CREATE(C_{bank}) $\wedge$ CREATE(C_{redeem})]
23:         *checkMsg* means {*check*};
24:     $Payee \rightarrow Bank$: [$choose \wedge check \wedge$ CREATE(C_{payB}) $\wedge$ CREATE(C_{bank}) $\wedge$ CREATE(C_{redeem})]
25:         *redeemMsg* means {*redeem*};
26:     $Bank \rightarrow Payee$: [$acct \wedge check \wedge redeem$] *payBMsg* means {*payB*};

Listing A.4 PayViaCredit Protocol

```
1:  protocol PayViaCredit { {
2:      role
3:          Payer;
4:          Issuer;
5:          Payee;
6:      prop
7:          apply;
8:          acct;
9:          choose;
10:         bill;
11:         payBill;
12:         credit;
13:         redeem;
14:         pay;
15:     commitment
16:         scpay = C_{Payer,Payee}(acct.isSet() and choose.isSet(), credit.isSet());
17:         scissuer = C_{Issuer,Payer}(acct.isSet() and credit.isSet() and redeem.isSet(), pay.isSet());
18:         scredeem = C_{Payee,Issuer}(acct.isSet() and credit.isSet(), redeem.isSet());
19:         scbill = C_{Payer,Issuer}(credit.isSet() and bill.isSet(), payBill.isSet());
20:     message
21:         Payer → Issuer : [true] applyMsg means {apply.set(), scbill.create()};
22:         Payer → Payee : [choose.isSet() ∧ scpay.isCreate() and scredeem.isCreate() ∧
    scissuer.isCreate()] creditMsg means { credit.set()};
23:         Payer → Issuer : [bill.isSet()] payBillMsg means {payBill.set()};
24:         Issuer → Payer : [true] issuerMsg means {scissuer.create()};
25:         Issuer → Payer : [apply.isSet()] acctMsg means {acct.set()};
26:         Issuer → Payee : [acct.isSet() ∧ credit.isSet() ∧ redeem.isSet()] payMsg means {pay.set()};
27:         Issuer → Payer : [true] billMsg means {bill.set()};
28:         Payee → Issuer : [true] willRedeemMsg means {scredeem.create()};
29:         Payee → Payer : [acct.isSet()] chooseMsg means {choose.set(), scpay.create()};
30:         Payee → Issuer : [credit.isSet() ∧ scpay.isCreate() ∧ scredeem.isCreate() ∧
    scissuer.isCreate()] redeemMsg means {redeem.set()};
31: }
```

Listing A.5 OrderPayShip Protocol

```
1:  protocol OrderPayShip {  {
2:      role
3:          Buyer;
4:          Seller;
5:      prop
6:          reqQuote;
7:          sendQuote;
8:          order;
9:          pay;
10:         ship;
11:     commitment
12:         scpay = C_{Buyer,Seller}(order.isSet(), pay.isSet());
13:         scship = C_{Seller,Buyer}(order.isSet(), ship.isSet());
14:     message
15:         Buyer → Seller : [true] reqQuoteMsg means {reqQuote.set()};
16:         Buyer → Seller : [sendQuote.isSet()∧scship.isCreate()] orderMsg means {order.set(), scpay.create()};
17:         Buyer → Seller : [order.isSet() ∧ scpay.isCreate()] payMsg means {pay.set()};
18:         Seller → Buyer : [reqQuote.isSet()] sendQuoteMsg means {sendQuote.set(), scship.create()};
19:         Seller → Buyer : [order.isSet() ∧ scpay.isCreate()] shipMsg means {ship.set()};
20: }
```

Appendix B

Refinement Theorems

The first theorem connects interpreted system models with our definitions.

Theorem B.0.1. *Let $P = \langle \mathscr{R}, \mathscr{M}, \mathscr{C}, \mathscr{A}, \mathscr{S}, \mathscr{S}^0, \mathscr{G} \rangle$ be a protocol and let $\mathscr{I}$ be its Proton model. A run is allowed by a Proton model $\mathscr{I}$ (Definition 5.2.2) if and only if it is a well-defined run (Definition 5.2.3).*

Sketch. Proton models allow interleaved, but not concurrent, messages. At each step, the environment schedules some role $r \in \mathscr{R}$. Role r chooses some enabled message $m \in Act^r$, and the ISPL joint action Act is equal to m,

Runs for both ISPL and Definition 5.2.3 begin in an initial state $s_0 \in \mathscr{S}^0$. At every step in a run, a message is enabled in ISPL by local strategy AP^r if and only if that message's guard is enabled. Therefore, a message can be appended to an ISPL run if and only if it can be appended to a Proton run. □

The next theorem shows embedding from Definition 5.2.8 is equivalent to the model checker verifying guards with Equation 5.16. The idea behind this theorem is that it assumes the two protocols are already mapped, so the guards of the superprotocol can be evaluated in the Proton model generated from the subprotocol.

Let π^i denote the path consisting of the first i steps of π, let $|\pi|$ be the length of path π, and let $\pi + \langle a, s \rangle$ be path π extended by action a resulting in new state s.

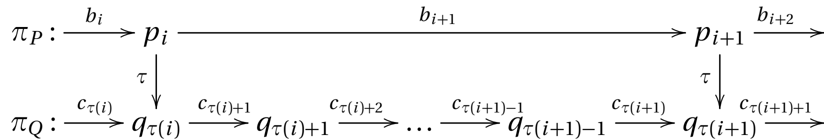


Figure B.1: The mapping between entities in π_P and π_Q.

Theorem B.0.2. *Let P and Q be two protocols, and M a mapping between them. Let $\mathscr{I}_Q$ be the Proton model for Q as specified in Definition 5.2.2. Let $\pi_P = \langle p_0, b_1, p_1, \ldots \rangle$, and $\pi_Q = \langle q_0, c_1, q_1, \ldots \rangle$. Then, for all runs $\pi_Q \in$ **runs**(Q), there exists a run $\pi_P \in$ **runs**(P) such that **emb**$(M(\pi_Q), \pi_P)$ if and only if* $(\forall a_i \in \mathscr{A}_P : \mathscr{I}, g \models \mathbf{AG}(a_i.\textit{sub-guard} \rightarrow a_i.\textit{super-guard}))$

Proof. From Theorem B.0.1, checking for well-defined runs is the same as checking the interpreted system model. Figure B.1 diagrams the relationships between entities in π_P and π_Q.

Let RHS be **emb**$(M(\pi_Q), \pi_P)$, and let LHS be $(\forall a_i \in \mathscr{A}_P : \mathscr{I}, g \models \mathbf{AG}(a_i.\textit{sub-guard} \rightarrow a_i.\textit{super-guard}))$.

$\Rightarrow$ Assume RHS. Let $LHS^j = (\forall i : \tau(i) \leq j, a_i \in \mathscr{A}^j : \mathscr{I}, g \models \mathbf{AG}(a_i.\textit{sub-guard} \rightarrow a_i.\textit{super-guard}))$ where set $\mathscr{A}^j = \{b_k \mid j = \tau(i) \wedge b_k \in \pi_P^i\}$ We prove LHS^j by induction on path length j in π_Q.

- Base case:
  - Define $\mathscr{A}^0 = b_0 \in \mathscr{S}_P^0$.
  - Define $\tau(0) = 0$.
  - LHS^0 is vacuously true.
- Inductive Step: Assume LHS^j. Consider action c_j.
  - Case: There are no more actions c_j. We are at the end of π_Q and setting $j = |\pi_Q|$ gives $LHS^{|\pi_Q|} = LHS$.
  - Case: $c_j.\textit{actexp} \notin \mathscr{A}_P$. This corresponds to the case $\tau(i) < j < \tau(i+1)$.
    - c_j has no effect on P.
    - $\widehat{M}(q_{\tau(i)}) = \widehat{M}(q_j)$ by Definition 5.2.8.
    - Define $\mathscr{A}^{j+1} = \mathscr{A}^j$.
    - Therefore, $LHS^{j+1} = LHS^j$ is true.
  - Case: $c_j.\textit{actexp} \in \mathscr{A}_P$. This corresponds to the case $j = \tau(i+1)$.
    - Define $j = \tau(i+1)$.
    - $p_i = \widehat{M}(q_{\tau(i+1)-1})$ by Definition 5.2.8.
    - $p_{i+1} = \widehat{M}(q_{\tau(i+1)})$ by RHS.
    - Since $q_{\tau(i+1)} = q_{\tau(i+1)-1} \cup c_{\tau(i+1)}.\textit{actexp}$ and $p_{i+1} = p_i \cup b_{i+1}.\textit{actexp}$, then $c_{\tau(i+1)}.\textit{actexp} = b_{i+1}.\textit{actexp}$, and $c_{\tau(i+1)} = b_{i+1}$.
    - Let $a_{i+1} = c_{\tau(i+1)} = b_{i+1}$ be the name of the action in LHS.
    - Define $\mathscr{A}^{j+1} = \mathscr{A}^j \cup b_{i+1}$. b_{i+1} might already be a member of $\mathscr{A}^j$.
    - $\widehat{M}(q_{\tau(i+1)-1}) \models c_{\tau(i+1)}.\textit{guard}$, since π_Q is well defined.
    - $p_i \models b_{i+1}.\textit{guard}$, since π_P is well defined,
    - Therefore, $c_{\tau(i+1)}.\textit{guard} = b_{i+1}.\textit{guard}$ in state p_i.

- $c_{\tau(i+1)}.guard \rightarrow b_{i+1}.guard$ in state p_i, by previous step. Induction shows it holds for all states in all $\pi_Q \in \mathbf{runs}(Q)$. Since we do not consider all runs $\pi_P \in \mathbf{runs}(P)$, there may be states where $b_{i+1}.guard$ is true, but $c_{\tau(i+1)}.guard$ is not true.
- $a_{i+1}.sub\text{-}guard \rightarrow a_{i+1}.super\text{-}guard$.
- Since there is a $\pi_Q \in \mathbf{runs}(Q)$ for every true guard, it is true for all reachable states (**AG**).
- Therefore, LHS^{j+1} is true.

$\Leftarrow$ Assume LHS. Let $RHS^j = (\forall i : \tau(i) \le j : \pi_P^i$ *is well defined* $\wedge\, \mathbf{emb}(M(\pi_Q^j), \pi_P^i))$. Given any π_Q, we will construct a π_P such that RHS^j by induction on path length j in π_Q.

- Base case:
  - $q_0 = \widehat{M}(q_0) = p_0 \in \mathscr{S}_P^0$.
  - Define $\tau(0) = 0$.
  - Define $\pi_P^i = \langle p_0 \rangle$ which is well defined.
  - Since $p_0 = \widehat{M}(q_0)$, then $\mathbf{emb}(M(\pi_Q^0), \pi_P^0) = RHS^0$.
- Inductive Step: Assume RHS^j. Then $\mathbf{emb}(M(\pi_Q^{\tau(i)}), \pi_P^i)$ so that $p_i = \widehat{M}(q_{\tau(i)})$. Consider the next action $c_{j+1} \in \pi_Q$.
  - Case: There are no more actions c_{j+1}. We are at the end of π_Q and setting $j = |\pi_Q|$ gives $RHS^{|\pi_Q|} = RHS$.
  - Case: Action $c_{j+1} \rightarrow \mathscr{A}_P$.
    - c_{j+1} does not change $\widehat{M}(q_j)$ by Definition 5.2.8.
    - $RHS^{j+1} = RHS^j$ is true.
  - Case: c_{j+1} equals some $b' \in \mathscr{A}_P$.
    - $c_{j+1} \in \mathscr{A}_P$ and $c_j.guard$ and $c_j.actexp$ are also elements in $\mathscr{A}_P$.
    - Define $\tau(i+1) = j+1$.
    - Let a_{i+1} in LHS equals $c_{j+1} = c_{\tau(i+1)}$ and $b' = b_{i+1}$ in RHS, so that $a_{i+1} = c_{\tau(i+1)} = b_{i+1}$. And a_{i+1}'s sub-guard is $c_{\tau(i+1)}.guard$ and a_{i+1}'s super-guard is $b_{i+1}.guard$.
    - Define $p_{i+1} = p_i \cup b_{i+1}.actexp$.
    - Define $\pi_P^{i+1} = \pi_P^i + \langle b_{i+1}, p_{i+1} \rangle$.

    Show π_P^{i+1} is well defined.

    1. - $q_{\tau(i+1)-1} \models c_{\tau(i+1)}.guard$, since π_Q is well defined.
       - $\widehat{M}(q_{\tau(i+1)-1}) \models c_{\tau(i+1)}.guard$, since $c_{\tau(i+1)}.guard$ is an element of $\mathscr{A}_P$.
       - $\widehat{M}(q_{\tau(i)}) \models c_{\tau(i+1)}.guard$ by Definition 5.2.8.
       - $p_i \models c_{\tau(i+1)}.guard$ by Definition 5.2.8 and RHS^j.

- a_{i+1}.*sub-guard* $\rightarrow a_{i+1}$.*super-guard* by LHS.
- Since $c_{\tau(i+1)} = a_{i+1} = b_{i+1}$ are all the same action, $c_{\tau(i+1)}$.*guard* $\rightarrow b_{i+1}$.*guard*.
- Therefore, $p_i \models b_{i+1}$.*guard*.

2. $p_{i+1} = p_i \cup b_{i+1}$.*actexp* by definition above.

Show $\mathbf{emb}(\pi_Q^{\tau(i+1)}, \pi_P^{i+1})$.

- $p_i = \widehat{M}(q_{\tau(i+1)-1})$ by Definition 5.2.8 and RHS^j.
- Since $c_{\tau(i+1)} = a_{i+1} = b_{i+1}$ are all the same action, $p_i \cup b_{i+1}$.*actexp* $= \widehat{M}(q_{\tau(i+1)-1}) \cup c_{\tau(i+1)}$.*actexp*.
- This reduces to $p_{i+1} = \widehat{M}(q_{\tau(i+1)})$.

Therefore, RHS^{j+1}.

□

The next definition characterizes that action a whose a.*actexp* causes m.*actexp* to become true.

Definition B.0.3 (Decisive)**.** *Let P be a protocol, let e, e' be two Boolean expressions, and let s be any state and $s' = s \cup e'$ be the next valid state after s where e' holds. e' is* decisive *for e, at a state s if and only if $s \not\models e' \wedge s' \models e'$ implies $s \not\models e \wedge s' \models e$.*

The state s' in the definition is the state in which e becomes true. An expression e' is decisive for expression e in state s if and only if, making e' true also makes e true. A change in e' causes a change in e.

In particular, we say an action a is decisive for message m at state s exactly when expression a.*actexp* is decisive for expression m.*actexp* at state s.

The next theorem shows that diffusion and collection properly maintain the guards and action expressions as a message is decomposed from a protocol P to its derived protocol $P' = \mathbf{col}(\mathbf{dif}(\mathbf{means}(M(P))))$. We prove it for mappings that contain individual actions and conjunctions, as well as mapping that contain disjunctions even though disjunction is not required by later proofs. This theorem is used between the superprotocol's super-gMsg and super-gAct, and between the subprotocol's sub-gMsg and sub-gAct in Figure 5.4b.

Theorem B.0.4 (Diffusion and Collection Preserve Guards)**.** *Let P be any protocol, let M be any mapping function, and let $gs \in \mathscr{G}$ be any guarded statement in P, possibly containing guarded action expressions. If gs_i is any guarded statement derived from gs by diffusion, and if gs_i is decisive for gs at state s then*

$$s \models gs_i.guard \longleftrightarrow s \models gs.guard$$

Proof. Show diffusion preserves guards.

Diffusion breaks one guarded statement gs into a set of guarded statements gs_i. Let $LHS^i = ((gs_i \text{ is decisive for } gs \text{ at } s) \rightarrow (s \models gs_i.guard \longleftrightarrow s \models gs.guard))$ where gs_i is derived from gs_{i-1} by diffusion and collection. We prove LHS^i by induction on the structure of $gs_i.actexp$.

- Base case:
  - $gs_0 = gs$ and $gs_0.guard = gs.guard$ is trivially true for all states s.
  - LHS^0 is true.
- Inductive Step: Assume LHS^i.
  - Case: There is no outermost operator in $gs_i.actexp$.
    - Then $gs_i.actexp$ is a single guarded action.
    - $gs_{i+1}.guard = gs_i.guard$ by Equation 5.15.
    - $gs_{i+1}.guard = gs.guard$ by LHS^i.
    - LHS^{i+1} since this holds in all states s.
  - Case: The outermost operator of $gs_i.actexp$ is disjunction.
    - Equation 5.13 applied to gs_i creates multiple guarded statements, one for each disjunct. Let gs_{i+1} be any of those disjuncts.
    - $gs_{i+1}.guard = gs_i.guard$ by Equation 5.13,
    - Each gs_{i+1} is decisive for gs.
    - $gs_{i+1}.guard = gs.guard$ by LHS^i.
    - LHS^{i+1} since this holds in all states s.
  - Case: The outermost operator of $e.actexp$ is conjunction.
    - Equation 5.14 applied to gs_i creates multiple guarded statements, one for each conjunct. Let gs_{i+1} be any of those conjuncts.
    - For gs_{i+1} to be decisive at s, all other conjuncts must be true at s.
    - $gs_{i+1}.guard = gs_i.guard$ by Equation 5.14 because all other conjuncts are true.
    - $gs_{i+1}.guard = gs.guard$ by LHS^i.
    - LHS^{i+1} since this holds in all states s for which gs_{i+1} is decisive.

By the no overlap constraint of Definition 5.2.1, in any state s, at most one of the guarded statement combined by collection can be enabled at a time. Therefore, collection also preserves guards. □

The next three theorems relate runs of a protocol P expressed in terms of messages and the runs of its derived protocol $P' = \mathbf{col}(\mathbf{dif}(M(P)))$ expressed in terms of actions. These theorems relate runs between both (1) the super-gMsg and super-gAct protocols, and (2) the sub-gMsg and sub-gAct protocols as shown in Figure 5.4. The first theorem proves every run of P embeds a run of P'.

Theorem B.0.5. *If Let $P = \langle \mathscr{R}, \mathscr{M}, \mathscr{C}, \mathscr{A}, \mathscr{S}, \mathscr{S}^0, \mathscr{G} \rangle$ be a protocol, possibly containing guarded action expressions, and let M be any mapping function. Let $P' = \textbf{\textit{col}}(\textbf{\textit{dif}}(M(P)))$ be the protocol derived from P by mapping, diffusion, and collection. Then*

$$\forall \pi_r \in \textbf{\textit{runs}}(P) \quad : \quad (\exists \pi_s \in \textbf{\textit{runs}}(P') : \textbf{\textit{emb}}(\pi_s, \pi_r))$$

Proof. Denote $\pi_r = \langle h_0, m_1, h_1, \ldots \rangle \in \textbf{runs}(P)$ with message $m_i \in \mathscr{M}$, and $\pi_s = \langle g_0, a_1, g_1, \ldots \rangle \in \textbf{runs}(P')$ with mapped action $a_i \in M(\mathscr{A})$. Let $LHS^i = (\pi_s^{\mu(i)}$ *is well defined* $\wedge\, \textbf{emb}(\pi_s^{\mu(i)}, \pi_r^i))$. We will construct a well-defined run $\pi_s \in \textbf{runs}(P')$, and show LHS^i by induction on $0 \le i \le |\pi_r|$.

- Base case:
  - Select any $g_0 \in \mathscr{S}^0$.
  - Define $\pi_s^0 = \langle g_0 \rangle$ which is well defined.
  - Define $\mu(0) = 0$.
  - Then $h_0 = \widehat{M}(g_0) = \widehat{M}(g_{\mu(0)})$.
  - LHS^0 is true.
- Inductive Step: Assume LHS^i where $h_i = \widehat{M}(g_{\mu(i)})$. Let m_{i+1} be the next message in π_r.
  - Case: No such m_{i+1} exists. All messages in π_r have been considered. $\pi_s = \pi_s^{\mu(i)}$ is a well-defined run and μ demonstrates $\textbf{emb}(\pi_s^{|\pi_s|}, \pi_r^{|\pi_r|}) = \textbf{emb}(\pi_s, \pi_r)$. $LHS^{|\pi_r|} = LHS$ is true.
  - Case: m_{j+1} exists.
    - Let $n = |\textbf{means}(m_{i+1})|$ be the number of actions in m_{i+1}'s meaning.
    - Define $g_{k+1} = g_k \cup a_{k+1}.\textit{actexp}\, \forall k : \mu(i) \le k < \mu(i) + n$.
    - Define $\pi_s^{\mu(i+1)} = \pi_s^{\mu(i)} + \sum_{k:\mu(i) \le k < \mu(i)+n} \langle a_{k+1}, g_{k+1} \rangle$ by appending all the actions a_k in $\textbf{means}(m_{i+1})$ onto the end, in any order.
    - Let a_d be the decisive action for m_{i+1} in π_r where $\mu(i) \le d \le \mu(i) + n$.
    - Define $\mu(i+1) = d$.

    Show $\pi_s^{\mu(i+1)}$ is well defined.

    1. Show $g_k \models a_{k+1}.\textit{guard}\, \forall k : \mu(i) \le k < \mu(i) + n$
       - $h_i \models m_{i+1}.\textit{guard}$ because π_r is well defined.
       - For each action $a_j \in \textbf{means}(m_{i+1})$, $m_{i+1}.\textit{guard} \rightarrow a_j.\textit{guard}$ by Equations 5.13, 5.14, and 5.15. Since collection disjoins guards, each of the actions' guard is true after collection.
       - Therefore, $h_i \models a_k.\textit{guard}\, \forall k : \mu(i) \le k < \mu(i) + n$.
    2. $g_{k+1} = g_k \cup a_k.\textit{actexp}\, \forall k : \mu(i) \le k < \mu(i) + n$ by the definition above.

Show $\mathbf{emb}(\pi_s^{\mu(i+1)}, \pi_r^{i+1})$.

- $g_{\mu(i+1)} = g_{\mu(i)} \bigcup_{\mu(i) \leq k < \mu(i)+n} a_k.actexp$ by definition of $g_{\mu(i+1)}$ above.
- $\widehat{M}(g_{\mu(i+1)}) = \widehat{M}(g_{\mu(i)} \bigcup_{\mu(i) < k \leq \mu(i)+n} a_k.actexp)$.
- For all j, $\widehat{M}(g_j \cup a_{j+1}.actexp) = \widehat{M}(g_{j+1})$ if a_{j+1} is not decisive for m_{j+1} in π_s.
- For all j, $\widehat{M}(g_j \cup a_{j+1}.actexp) = \widehat{M}(g_{j+1}) \cup m_{j+1}.actexp$ if a_{j+1} is decisive for m_{j+1} in π_s.
- $\widehat{M}(g_{\mu(i+1)}) = \widehat{M}(g_{\mu(i)}) \cup m_{j+1}.actexp$, by applying the previous two reductions for all a_{j+1}.
- $\widehat{M}(g_{\mu(i+1)}) = h_i \cup m_{j+1}.actexp$, since $h_i = \widehat{M}(g_{\mu(i)})$ by LHS^{i+1}.
- $\widehat{M}(g_{\mu(i+1)}) = h_{i+1}$ by the definition of h_{i+1}.

- Therefore, LHS^{i+1} is true.

□

The next theorem shows the reverse: every run in P' embeds a run in P.

Theorem B.0.6. *If Let $P = \langle \mathscr{R}, \mathscr{M}, \mathscr{C}, \mathscr{A}, \mathscr{S}, \mathscr{S}^0, \mathscr{G} \rangle$ be a protocol, possibly containing guarded action expressions, and let M be any mapping function. Let $P' = \textbf{col}(\textbf{dif}(M(P)))$ be the protocol derived from P by mapping, diffusion, and collection. Then*

$$\forall \pi_s \in \textbf{runs}(P') \quad : \quad (\exists \pi_r \in \textbf{runs}(P) : \textbf{emb}(\pi_s, \pi_r))$$

Proof. Denote $\pi_r = \langle h_0, m_1, h_1, \ldots \rangle \in \mathbf{runs}(P)$ with message $m_i \in \mathscr{M}$, and $\pi_s = \langle g_0, a_1, g_1, \ldots \rangle \in \mathbf{runs}(P')$ with mapped action $a_i \in M(\mathscr{A})$.

Let $LHS^j = (i = argmax_k\, \mu(k) \leq j : \pi_r^i$ *is well defined* $\wedge \mathbf{emb}(\pi_s^j, \pi_r^i))$. We will construct $\pi_r \in \mathbf{runs}(P)$ and show LHS^j by induction on the path length $0 \leq j \leq |\pi_s|$. We allow additional actions in π_s after $\mu(i)$ as long as they have no effect on P.

- Base case:
  - Select any $h_0 \in \mathscr{S}^0$,
  - Define $\pi_r^0 = \langle h_0 \rangle$,
  - π_r^0 is well defined.
  - Define $\mu(0) = 0$.
  - Then $\widehat{M}(g_0) = \widehat{M}(g_{\mu(0)}) = h_0$ implies $\mathbf{emb}(\pi_s^0, \pi_r^0)$ with $\mu(0) \leq 0$.
  - LHS^0 is true.
- Inductive Step: Consider the next action $a_{j+1} \in \pi_s$.

- Case: No such a_{j+1} exists. All actions in π_s have been considered. $\pi_r = \pi_r^i$ is a well-defined run, and μ demonstrates $\mathbf{emb}(\pi_s^{|\pi_s|}, \pi_r^{|\pi_r|}) = \mathbf{emb}(\pi_s, \pi_r)$ with $\forall i : \mu(i) \leq j$. $LHS^{|\pi_r|} = LHS$.
- Case: a_j exists but it is *not* decisive for any $m \in \mathscr{A}_P$.
  - Leave π_r^i unchanged which is still well defined.
  - Leave μ unchanged.
  - No additional $h_i = \widehat{M}(g_{\mu(i)})$ conditions are required to established $\mathbf{emb}(\pi_s^{j+1}, \pi_r^i)$ with $\forall i : \mu(i) \leq j$, and the existing conditions are true by the induction hypothesis.
  - LHS^{j+1} is true.
- Case: a_{j+1} exists and it is decisive for some $m \in \mathscr{A}_P$.
  - There is at most one such message m by the no overlap constraint of Definition 5.2.1. Denote the message by $m_{i+1} = m$.
  - Define $h_{i+1} = h_i \cup m_{i+1}.\mathit{actexp}$.
  - Define $\pi_r^{i+1} = \pi_r^i + \langle m_{i+1}, h_{i+1} \rangle$.
  - Define $\mu(i+1) = j+1$.

  Show π_r^{i+1} is well defined.

  1. Show $h_i \models m_{i+1}.\mathit{guard}$.
     - $g_j \models a_{j+1}.\mathit{guard}$ because π_s is well defined.
     - $\widehat{M}(g_j) \models \widehat{M}(a_{j+1}.\mathit{guard})$.
     - $\widehat{M}(g_j) \models m_{i+1}.\mathit{guard}$ because $a_{j+1}.\mathit{guard} = m_{i+1}.\mathit{guard}$ by Theorem B.0.4.
     - $\widehat{M}(g_{\mu(i+1)-1}) \models m_{i+1}.\mathit{guard}$ by definition of $\mu(i+1)$.
     - $\widehat{M}(g_{\mu(i)}) \models m_{i+1}.\mathit{guard}$ by Definition 5.2.8.
     - Therefore, $h_i \models m_{i+1}.\mathit{guard}$ by LHS^j.
  2. $h_{i+1} \models m_{i+1}.\mathit{actexp}$ by definition of h_{i+1} above.

  Show $\mathbf{emb}(\pi_s^{j+1}, \pi_r^{i+1})$.

  - $g_{j+1} = g_j \cup a_{j+1}.\mathit{actexp}$ since π_s is well defined.
  - $\widehat{M}(g_{j+1}) = \widehat{M}(g_j \cup a_{j+1}.\mathit{actexp})$.
  - For all j, $\widehat{M}(g_j \cup a_{j+1}.\mathit{actexp}) = \widehat{M}(g_j) \cup m_{j+1}.\mathit{actexp}$ if a_{j+1} is decisive for $m_{j+1}.\mathit{actexp}$ in π_s.
  - $\widehat{M}(g_{j+1}) = \widehat{M}(g_j) \cup m_{j+1}.\mathit{actexp}$, by applying the previous reduction.
  - $\widehat{M}(g_{\mu(i+1)}) = \widehat{M}(g_{\mu(i+1)-1}) \cup m_{j+1}.\mathit{actexp}$, by definition of $\mu(i+1)$.
  - $\widehat{M}(g_{\mu(i+1)}) = \widehat{M}(g_{\mu(i)}) \cup m_{j+1}.\mathit{actexp}$, by Definition 5.2.8.
  - $\widehat{M}(g_{\mu(i+1)}) = h_i \cup m_{j+1}.\mathit{actexp}$, since $h_i = \widehat{M}(g_{\mu(i)})$ by LHS^j.
  - $\widehat{M}(g_{\mu(i+1)}) = h_{i+1}$ by the definition of h_{i+1}.
- LHS^{i+1} is true.

□

Theorem B.0.7. *If Let* $P = \langle \mathscr{R}, \mathscr{M}, \mathscr{C}, \mathscr{A}, \mathscr{S}, \mathscr{S}^0, \mathscr{G} \rangle$ *be a protocol, possibly containing guarded action expressions, and let M be any mapping function. Let* $P' = \boldsymbol{col}(\boldsymbol{dif}(M(P)))$ *be the protocol derived from P by mapping function M, diffusion, and collection. Then*

$$\begin{aligned} \forall \pi_r \in \boldsymbol{runs}(P) \quad &: \quad (\exists \pi_s \in \boldsymbol{runs}(P') : \boldsymbol{emb}(\pi_s, \pi_r)) \\ \forall \pi_s \in \boldsymbol{runs}(P') \quad &: \quad (\exists \pi_r \in \boldsymbol{runs}(P) : \boldsymbol{emb}(\pi_s, \pi_r)) \end{aligned}$$

Proof. Follows immediately from Theorems B.0.5 and B.0.6. □

Appendix C

Rho Refactoring Library

We list the refactorings in our Rho refactoring library, along with a brief description. Incomplete refactorings are marked [In progress].

Many authors consider refactorings to be special transformations that do not modify some invariant (e.g, a program's computational output). Because we do not propose such an invariant for protocols, our use of the term "refactoring" is more generalized.

C.1 Protocol Designer Independence Refactorings

1. *Add Role* adds a new role and its corresponding chain to the protocol with no function or responsibilities. This refactoring creates a new empty agent with an empty chain. This refactoring does not create any reactions and does not materially change the protocol.
2. *Map Role* maps one role to a set of roles. [In progress]
3. *Remove Role* removes an existing role and its corresponding chain from the protocol. This can only be done if neither role nor chain have any function or responsibilities. This refactoring does not create or delete any reactions and does not materially change the protocol.
4. *Add Proposition* adds a Boolean proposition to the protocol.
5. *Map Proposition* maps a proposition to a Boolean expression of propositions. [In progress]
6. *Remove Proposition* removes an existing Boolean proposition from the protocol.
7. *Add Commitment* adds a commitment to the protocol.
8. *Map Commitment* maps a commitment from a higher-level commitment to a serial composition of lower-level commitments. [In progress]
9. *Remove Commitment* removes an existing commitment from the protocol.
10. *Add Meaning* adds a new meaning to a message. Meanings have the form *op(obj)* or *obj.op* where *obj* is a proposition or commitment in the protocol, and *op* is a predefined operation for *obj*.

11. *Remove Meaning* removes a meaning from a message.
12. *Add Message* inserts a new message m into the generated protocol. The new message must be *triggered* by some other event. The sender and receiver can be any existing roles.
13. *Rename Message* renames the message type of a source protocol message m to a new type n in the generated protocol. It does not change the number of messages, nor the message's sender (*snd*) or receiver (*rcv*). Message data fields can be extended, reordered or reformatted.
14. *Remove Message* removes an existing message m from a generated protocol. The sender and receiver can be any existing roles. This refactoring can only be applied if the receiver does not need to know the message occurred.
15. *Split Message* splits a single message m into two, parallel messages m_1 and m_2. Both new messages have the same sender and receiver as the original message. It sends message meanings and data via two messages and at different times.
16. *Merge Message* merges two messages m_1 and m_2, both with the same sender and receiver, into a single message m. It simplifies the protocol by merging the function of two parallel messages into a single message.
17. *Add Middleman* replaces a single message m with a pair of messages m_1 and m_2. Adding a middleman reroutes a single message through a new middleman role. Message m_1 must occur before message m_2.
18. *Remove Middleman* removes a middleman from a pair of adjacent messages. A pair of adjacent messages m_1 and m_2 with. This refactoring can only be applied when the removed middleman does not need the know the message m_1 has occurred. This refactoring eliminates a middleman.
19. *Change Sender* many refactorings convert one path into a different path, but maintain the starting and ending roles. This refactoring, which changes the starting roles of a path, requires special care, because it requires the new sender snd_2 possesses additional knowledge beyond that required by the protocol. [In progress]
20. *Change Receiver* changes the receiver of message m from rcv_1 to rcv_2. The main issues are rcv_1's loss of knowledge about (1) whether or not the message was ever sent, and (2) the values of the message data fields. [In progress]
21. *Protocol Kill* propagates the kill assertion from sender to receiver, deleting the message from the protocol.

C.2 Agent Designer Independence Refactorings

1. *Internalize Reaction* moves a reaction out of the role end of a chain and into the agent's internal implementation. It is the inverse of *Externalize Reaction*.
2. *Externalize Reaction* moves a reaction out of the agent's implementation and into a reaction at the role end of the chain.

3. *Push Kill* Sending agent publicly declares it will never send m by pushing kill snd m onto its chain.
4. *Pop Kill* Receiving agent accepts it will never receive m by popping kill snd m off its chain.

C.3 Designer Collaboration Refactorings

1. *Move Kill* moves a kill declaration up or down the chain.
2. *Add Procedure* adds a procedure call to an existing *if-clause* or *do-clause*. This enables chains to save or get, additional information during the protocol's enactment. This refactoring modifies an existing reaction, but does not add any new reactions. The call can be inserted at any point in the *do-clause* or *if-clause*. Normal programming rules apply, such as a call must not be inserted before its parameters are available.
3. *Delete Procedure* deletes an existing procedure call from a *if-clause* or *do-clause*. This allows chains to simplify the protocol's enactment. This refactoring modifies an existing reaction, but does not add any new reactions. Normal programming rules apply, such as a call must not be delete if its results are still needed.
4. *Swap Reactions* interchanges the order of two, adjacent reactions in a chain. We must prevent interchanging these reactions, where the do-clause of the first matches the on-clause of the second. [In progress]
5. *Merge Reaction* combines two adjacent reactions. The general pattern of all these rules is merge two reactions when $clause_2$ appears in a do-clause adjacent to an on-clause.

Appendix D

SWDev Interceptor Chains

Applying the refactorings produces the following interceptor chains for Interaction *SWDev1* to *SWDev2* (nonessential elements omitted to improve clarity).

Further *Agent Designer Independence* refactorings can be applied when each agent implementations is modified. Many reactions for a role have an empty do clause. These reactions consume the received message, because that role does not (yet) accept that message. These reactions can be internalized into their agent's implementation using refactoring *Internalize Reaction*.

- Cl:
  1. RoleEnd Cl
  2. on rcv notify_succPSP_Cl_Msg do {rcv Cl_PSP_payserver_deliverMsg}
  3. on rcv notify_failPSP_Cl_Msg do {rcv Cl_PSP_payserver_cantDoMsg}
  4. on rcv notify_succSP_Cl_Msg do {}
  5. on rcv notify_failSP_Cl_Msg do {}
  6. on rcv notify_succTe_Cl_Msg do {rcv Cl_Te_server_deliverMsg}
  7. on rcv notify_failTe_Cl_Msg do {rcv Cl_Te_server_cantDoMsg}
  8. on rcv notify_succEx_Cl_Msg do {rcv Cl_Ex_server_deliverMsg}
  9. on rcv notify_failEx_Cl_Msg do {rcv Cl_Ex_server_cantDoMsg}
  10. ProtocolEnd Cl
- PSP:
  1. RoleEnd PSP
  2. on snd Cl_PSP_payserver_deliverMsg to Cl do {snd Cl_PSP_payserver_deliverMsg to Re}
  3. on snd Cl_PSP_payserver_cantDoMsg to Cl do {snd Cl_PSP_payserver_cantDoMsg to Re}
  4. on rcv notify_succSP_PSP_Msg do {rcv PSP_SP_server_deliverMsg}
  5. Kill snd pspNotifySP to SP
  6. Kill snd pspNotifyCl to Cl
  7. on rcv notify_failSP_PSP_Msg do {rcv PSP_SP_server_cantDoMsg}
  8. on rcv notify_succTe_PSP_Msg do {}
  9. on rcv notify_failTe_PSP_Msg do {}
  10. on rcv notify_succEx_PSP_Msg do {}

11. on rcv notify_failEx_PSP_Msg do {}
12. ProtocolEnd PSP

- SP:
  1. RoleEnd SP
  2. on rcv notify_succPSP_SP_Msg do {}
  3. Kill snd spNotifyPSP to PSP
  4. Kill snd spNotifyCl to Cl
  5. on rcv notify_failPSP_SP_Msg do {}
  6. on snd PSP_SP_server_deliverMsg to PSP do {snd PSP_SP_server_deliverMsg to Re}
  7. on snd PSP_SP_server_cantDoMsg to PSP do {snd PSP_SP_server_cantDoMsg to Re}
  8. on rcv notify_succTe_SP_Msg do {}
  9. on rcv notify_failTe_SP_Msg do {}
  10. on rcv notify_succEx_SP_Msg do {}
  11. on rcv notify_failEx_SP_Msg do {}
  12. ProtocolEnd SP
- Te:
  1. RoleEnd Te
  2. on rcv notify_succPSP_Te_Msg do {}
  3. on rcv notify_failPSP_Te_Msg do {}
  4. on rcv notify_succSP_Te_Msg do {}
  5. on rcv notify_failSP_Te_Msg do {}
  6. on snd Cl_Te_server_deliverMsg to Cl do {snd Cl_Te_server_deliverMsg to Re}
  7. on snd Cl_Te_server_cantDoMsg to Cl do {snd Cl_Te_server_cantDoMsg to Re}
  8. on rcv notify_succEx_Te_Msg do {}
  9. on rcv notify_failEx_Te_Msg do {}
  10. ProtocolEnd Te
- Ex:
  1. RoleEnd Ex
  2. on rcv notify_succPSP_Ex_Msg do {}
  3. on rcv notify_failPSP_Ex_Msg do {}
  4. on rcv notify_succSP_Ex_Msg do {}
  5. on rcv notify_failSP_Ex_Msg do {}
  6. on rcv notify_succTe_Ex_Msg do {}
  7. on rcv notify_failTe_Ex_Msg do {}
  8. on snd Cl_Ex_server_deliverMsg to Cl do {snd Cl_Ex_server_deliverMsg to Re}
  9. on snd Cl_Ex_server_cantDoMsg to Cl do {snd Cl_Ex_server_cantDoMsg to Re}
  10. ProtocolEnd Ex
- Re:
  1. RoleEnd Re
  2. on rcv Cl_PSP_payserver_deliverMsg do {snd notify_succPSP_Cl_Msg to Cl}
  3. on snd notify_succPSP_Cl_Msg to Cl do {snd notify_succPSP_Cl_Msg to Cl, snd notify_-succPSP_SP_Msg to SP}
  4. on snd notify_succPSP_Cl_Msg to Cl do {snd notify_succPSP_Cl_Msg to Cl, snd notify_-succPSP_Te_Msg to Te}

5. on snd notify_succPSP_Cl_Msg to Cl do {snd notify_succPSP_Cl_Msg to Cl, snd notify_-succPSP_Ex_Msg to Ex}
6. on rcv Cl_PSP_payserver_cantDoMsg do {snd notify_failPSP_Cl_Msg to Cl}
7. on snd notify_failPSP_Cl_Msg to Cl do {snd notify_failPSP_Cl_Msg to Cl, snd notify_-failPSP_SP_Msg to SP}
8. on snd notify_failPSP_Cl_Msg to Cl do {snd notify_failPSP_Cl_Msg to Cl, snd notify_-failPSP_Te_Msg to Te}
9. on snd notify_failPSP_Cl_Msg to Cl do {snd notify_failPSP_Cl_Msg to Cl, snd notify_-failPSP_Ex_Msg to Ex}
10. on rcv PSP_SP_server_deliverMsg do {snd notify_succSP_PSP_Msg to PSP}
11. on snd notify_succSP_PSP_Msg to PSP do {snd notify_succSP_PSP_Msg to PSP, snd notify_-succSP_Cl_Msg to Cl}
12. on snd notify_succSP_PSP_Msg to PSP do {snd notify_succSP_PSP_Msg to PSP, snd notify_-succSP_Te_Msg to Te}
13. on snd notify_succSP_PSP_Msg to PSP do {snd notify_succSP_PSP_Msg to PSP, snd notify_-succSP_Ex_Msg to Ex}
14. on rcv PSP_SP_server_cantDoMsg do {snd notify_failSP_PSP_Msg to PSP}
15. on snd notify_failSP_PSP_Msg to PSP do {snd notify_failSP_PSP_Msg to PSP, snd notify_-failSP_Cl_Msg to Cl}
16. on snd notify_failSP_PSP_Msg to PSP do {snd notify_failSP_PSP_Msg to PSP, snd notify_-failSP_Te_Msg to Te}
17. on snd notify_failSP_PSP_Msg to PSP do {snd notify_failSP_PSP_Msg to PSP, snd notify_-failSP_Ex_Msg to Ex}
18. on rcv Cl_Te_server_deliverMsg do {snd notify_succTe_Cl_Msg to Cl}
19. on snd notify_succTe_Cl_Msg to Cl do {snd notify_succTe_Cl_Msg to Cl, snd notify_-succTe_PSP_Msg to PSP}
20. on snd notify_succTe_Cl_Msg to Cl do {snd notify_succTe_Cl_Msg to Cl, snd notify_-succTe_SP_Msg to SP}
21. on snd notify_succTe_Cl_Msg to Cl do {snd notify_succTe_Cl_Msg to Cl, snd notify_-succTe_Ex_Msg to Ex}
22. on rcv Cl_Te_server_cantDoMsg do {snd notify_failTe_Cl_Msg to Cl}
23. on snd notify_failTe_Cl_Msg to Cl do {snd notify_failTe_Cl_Msg to Cl, snd notify_failTe_-PSP_Msg to PSP}
24. on snd notify_failTe_Cl_Msg to Cl do {snd notify_failTe_Cl_Msg to Cl, snd notify_failTe_-SP_Msg to SP}
25. on snd notify_failTe_Cl_Msg to Cl do {snd notify_failTe_Cl_Msg to Cl, snd notify_failTe_-Ex_Msg to Ex}
26. on rcv Cl_Ex_server_deliverMsg do {snd notify_succEx_Cl_Msg to Cl}
27. on snd notify_succEx_Cl_Msg to Cl do {snd notify_succEx_Cl_Msg to Cl, snd notify_-succEx_PSP_Msg to PSP}
28. on snd notify_succEx_Cl_Msg to Cl do {snd notify_succEx_Cl_Msg to Cl, snd notify_-succEx_SP_Msg to SP}
29. on snd notify_succEx_Cl_Msg to Cl do {snd notify_succEx_Cl_Msg to Cl, snd notify_-succEx_Te_Msg to Te}

30. on rcv Cl_Ex_server_cantDoMsg do {snd notify_failEx_Cl_Msg to Cl}
31. on snd notify_failEx_Cl_Msg to Cl do {snd notify_failEx_Cl_Msg to Cl, snd notify_failEx_-PSP_Msg to PSP}
32. on snd notify_failEx_Cl_Msg to Cl do {snd notify_failEx_Cl_Msg to Cl, snd notify_failEx_-SP_Msg to SP}
33. on snd notify_failEx_Cl_Msg to Cl do {snd notify_failEx_Cl_Msg to Cl, snd notify_failEx_-Te_Msg to Te}
34. ProtocolEnd Re