This thesis has been submitted in fulfilment of the requirements for a postgraduate degree (e.g. PhD, MPhil, DClinPsychol) at the University of Edinburgh. Please note the following terms and conditions of use:

This work is protected by copyright and other intellectual property rights, which are retained by the thesis author, unless otherwise stated.

A copy can be downloaded for personal non-commercial research or study, without prior permission or charge.

This thesis cannot be reproduced or quoted extensively from without first obtaining permission in writing from the author.

The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the author.

When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given.
A Formal Verification Approach to Process Modelling and Composition

Petros Papapanagiotou

Doctor of Philosophy
Centre for Intelligent Systems and their Applications
School of Informatics
University of Edinburgh

2014
Abstract

Process modelling is a design approach where a system or procedure is decomposed in a number of abstract, independent, but connected processes, and then recomposed into a well-defined workflow specification. Research in formal verification, for its part, and theorem proving in particular, is focused on the rigorous verification of system properties using logical proof.

This thesis introduces a systematic methodology for process modelling and composition based on formal verification. Our aim is to augment the numerous benefits of a workflow based specification, such as modularity, separation of concerns, interoperability between heterogeneous (including human-based) components, and optimisation, with the high level of trust provided by formally verified properties, such as type correctness, systematic resource accounting (including exception handling), and deadlock-freedom.

More specifically, we focus on bridging the gap between the deeply theoretical proofs-as-processes paradigm and the highly pragmatic tasks of process specification and composition. To accomplish this, we embed the proofs-as-processes paradigm within the modern proof assistant HOL Light. This allows the formal, mechanical translation of Classical Linear Logic (CLL) proofs to \( \pi \)-calculus processes. Our methodology then relies on the specification of abstract processes in CLL terms and their composition using CLL inference. A fully diagrammatic interface is used to guide our developed set of high level, semi-automated reasoning tools, and to perform intuitive composition actions including sequential, parallel, and conditional composition.

The end result is a \( \pi \)-calculus specification of the constructed workflow, with guarantees of correctness for the aforementioned properties. We can then apply a visual, step-by-step simulation of this workflow or perform an automated workflow deployment as executable code in the programming language Scala.

We apply our methodology to a use-case of a holiday booking web agent and to the modelling of real-world collaboration patterns in healthcare, thus demonstrating the capabilities of our framework and its potential use in a variety of scenarios.
Acknowledgements

Work as a PhD student can arguably be a lonely process. Thankfully, I was surrounded by a number of people who supported me in different ways and without whom I would not have made it.

First and foremost, I would like to express my gratitude to my supervisor and mentor Jacques Fleuriot for his continuous guidance throughout the past years. His insightful, hands-on advice and constant encouragement, as well as his tolerance of my faults and shortcomings, contributed substantially not only to the quality of this work, but also to my own improvement as a person.

Secondly, this work would not have been possible without the love and support of my family. I am grateful to my parents for their endless support and encouragement and to my brother and his family for filling my life with love and inspiration.

I am thankful for the fruitful, close collaboration with my colleagues and friends Areti Manatakaki and Phil Scott. Working together with Sean Wilson, Adela Grando, Kanchan Patil, Stephen Morgan, and Justine Lowe has been not only constructive, but also a pleasant and fulfilling experience.

I would like to express my gratitude to my examiners Alan Smaill, John Harrison, and Andrew Ireland for making the viva an enjoyable process while providing crucial, insightful comments. I would also like to thank a number of people who contributed ideas and input, whether in the form of detailed feedback or with short but impactful comments, including but not limited to Dave Robertson, Michael Rovatsos, Steven Obua, Alan Bundy, Mark Adams, Tom Melham, Rob Arthan, Antonio Ravara, and Phil Wadler.

This work would have been impossible outside the amazing, enjoyable environment provided by the members of the DReaM group, my friends and fellow PhD students, and the members of the admin and support staff. Many thanks, also, to my close friends Enrico Marantidis and Claire Cordina for their love and support. Last but not least, I would like to thank Iro Zygoura for her support and tolerance.

This work was funded by an EPSRC doctoral scholarship. I have also received funding from EPSRC grant EP/J001058/1, the College of Science and Engineering, University of Edinburgh, and the Centre for Intelligent Systems and their Applications.
Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

Chapters 6 and 7 of this thesis include work from the following jointly-authored publication:


The research and implementation of the logic-based composition actions described in this, as well as the specification for the diagrammatic design, were developed by myself, under the supervision of Jacques Fleuriot and with contributions from Sean Wilson, who implemented the user interface in Java.

Chapter 10 of this thesis is based on work from the following jointly-authored publication:


The research described in this publication was performed as part of the current work by myself, under the supervision of Jacques Fleuriot.

(Petros Papapanagiotou)
To my parents
# Table of Contents

1 Introduction .................................................. 1  
  1.1 Divide, Conquer, and Rule via Process Modelling ............... 1  
  1.2 Trust Through Formal Verification .................................. 2  
  1.3 Our Vision .................................................. 3  
  1.4 Our Approach: Assumptions and Outcomes ........................... 6  
  1.5 Achieving Our Goals - Thesis Outline ............................... 8  

2 Background ..................................................... 11  
  2.1 Process Modelling ............................................. 11  
      2.1.1 Web Services ............................................. 12  
      2.1.2 Business Process Modelling ................................ 14  
      2.1.3 Motivating Example ...................................... 16  
  2.2 Formal Methods via Theorem Proving .............................. 18  
      2.2.1 Modern Theorem Provers .................................. 19  
      2.2.2 HOL Light ............................................... 21  
      2.2.3 Formal Verification of Process Workflows ................. 21  
  2.3 Conclusion .................................................. 22  

3 The $\pi$-calculus .............................................. 25  
  3.1 Introduction ................................................ 25  
  3.2 Description .................................................. 26  
  3.3 Tools ....................................................... 28  
      3.3.1 Reasoning about Concurrent Systems ....................... 28  
      3.3.2 Visualisation and Simulation .............................. 29  
  3.4 Embedding the Polyadic $\pi$-calculus in HOL Light ................ 32  
      3.4.1 Syntax in HOL Light ..................................... 32  
      3.4.2 Functions about Names ................................... 33
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.4.3</td>
<td>Variable Substitution</td>
<td>34</td>
</tr>
<tr>
<td>3.4.4</td>
<td>Extra Theorems and Tools</td>
<td>36</td>
</tr>
<tr>
<td>3.5</td>
<td>Limitations and Future Work</td>
<td>36</td>
</tr>
<tr>
<td>3.6</td>
<td>Conclusion</td>
<td>38</td>
</tr>
<tr>
<td>4</td>
<td>The Proofs-as-processes Paradigm</td>
<td>39</td>
</tr>
<tr>
<td>4.1</td>
<td>Overview</td>
<td>39</td>
</tr>
<tr>
<td>4.2</td>
<td>Classical Linear Logic</td>
<td>40</td>
</tr>
<tr>
<td>4.3</td>
<td>Proof Terms and Inference Rules</td>
<td>44</td>
</tr>
<tr>
<td>4.4</td>
<td>Formalising the Proofs-as-processes Paradigm</td>
<td>47</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Syntax</td>
<td>48</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Using Multisets in Sequent Calculus Judgements</td>
<td>50</td>
</tr>
<tr>
<td>4.4.3</td>
<td>Sequent Calculus Embedding with Proof Terms in HOL Light</td>
<td>50</td>
</tr>
<tr>
<td>4.4.4</td>
<td>Proofs-as-processes Inference Rules</td>
<td>51</td>
</tr>
<tr>
<td>4.4.5</td>
<td>Inference in the Embedded Logic</td>
<td>54</td>
</tr>
<tr>
<td>4.5</td>
<td>Related Work</td>
<td>64</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Variations of Linear Logic</td>
<td>64</td>
</tr>
<tr>
<td>4.5.2</td>
<td>Evolution of the Proofs-as-processes Paradigm</td>
<td>69</td>
</tr>
<tr>
<td>4.6</td>
<td>Conclusion and Future Work</td>
<td>74</td>
</tr>
<tr>
<td>5</td>
<td>Formal Analysis of Rao et al.'s Approach</td>
<td>77</td>
</tr>
<tr>
<td>5.1</td>
<td>Introduction</td>
<td>77</td>
</tr>
<tr>
<td>5.2</td>
<td>The Work of Rao et al.</td>
<td>79</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Example</td>
<td>79</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Translation</td>
<td>80</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Composition as Proof</td>
<td>82</td>
</tr>
<tr>
<td>5.2.4</td>
<td>Extraction</td>
<td>86</td>
</tr>
<tr>
<td>5.3</td>
<td>Issues with the Process Calculus</td>
<td>89</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Encoding Sequence in π-calculus</td>
<td>90</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Channels</td>
<td>91</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Ports</td>
<td>92</td>
</tr>
<tr>
<td>5.4</td>
<td>Issues with the Proof to Process Correspondence</td>
<td>97</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Inconsistencies in Proofs Terms</td>
<td>98</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Two-sided Sequent Calculus and ILL</td>
<td>101</td>
</tr>
<tr>
<td>5.4.3</td>
<td>The Identity Axiom</td>
<td>103</td>
</tr>
<tr>
<td>5.5</td>
<td>Structural Congruence Rules</td>
<td>104</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>7.6.2 Proof</td>
<td>170</td>
<td></td>
</tr>
<tr>
<td>7.6.3 Result</td>
<td>173</td>
<td></td>
</tr>
<tr>
<td>7.7 Conclusion and Future Work</td>
<td>174</td>
<td></td>
</tr>
<tr>
<td>8 Automated Workflow Deployment</td>
<td>179</td>
<td></td>
</tr>
<tr>
<td>8.1 Overview</td>
<td>179</td>
<td></td>
</tr>
<tr>
<td>8.2 Preliminaries</td>
<td>181</td>
<td></td>
</tr>
<tr>
<td>8.2.1 From CLL Types to Scala Types</td>
<td>181</td>
<td></td>
</tr>
<tr>
<td>8.2.2 From $\pi$-calculus to Executable Code: The PiLib Library</td>
<td>182</td>
<td></td>
</tr>
<tr>
<td>8.2.3 Extending the PiLib Library</td>
<td>186</td>
<td></td>
</tr>
<tr>
<td>8.2.4 Type Casting</td>
<td>189</td>
<td></td>
</tr>
<tr>
<td>8.3 Main Component Architecture</td>
<td>190</td>
<td></td>
</tr>
<tr>
<td>8.3.1 Atomic Processes as Traits</td>
<td>191</td>
<td></td>
</tr>
<tr>
<td>8.3.2 Composite Processes</td>
<td>195</td>
<td></td>
</tr>
<tr>
<td>8.4 From Proofs-as-processes to Functions</td>
<td>199</td>
<td></td>
</tr>
<tr>
<td>8.4.1 Atomic Processes</td>
<td>200</td>
<td></td>
</tr>
<tr>
<td>8.4.2 Composite Processes</td>
<td>210</td>
<td></td>
</tr>
<tr>
<td>8.5 The Credit Card Example in Scala Code</td>
<td>216</td>
<td></td>
</tr>
<tr>
<td>8.6 Conclusion and Future Work</td>
<td>217</td>
<td></td>
</tr>
<tr>
<td>9 Use Case: Holiday Booking Web Agent</td>
<td>225</td>
<td></td>
</tr>
<tr>
<td>9.1 Introduction</td>
<td>225</td>
<td></td>
</tr>
<tr>
<td>9.2 Component Services</td>
<td>227</td>
<td></td>
</tr>
<tr>
<td>9.3 Composing Services</td>
<td>233</td>
<td></td>
</tr>
<tr>
<td>9.3.1 Example of Compositionality: Credit Card Payments</td>
<td>233</td>
<td></td>
</tr>
<tr>
<td>9.3.2 Example of Exception Handling: Flight Search</td>
<td>234</td>
<td></td>
</tr>
<tr>
<td>9.4 Results</td>
<td>240</td>
<td></td>
</tr>
<tr>
<td>9.5 Conclusion</td>
<td>242</td>
<td></td>
</tr>
<tr>
<td>10 Formal Verification of Collaboration Patterns in Healthcare</td>
<td>245</td>
<td></td>
</tr>
<tr>
<td>10.1 Introduction</td>
<td>245</td>
<td></td>
</tr>
<tr>
<td>10.2 Motivation</td>
<td>247</td>
<td></td>
</tr>
<tr>
<td>10.3 Collaborative Healthcare Patterns</td>
<td>249</td>
<td></td>
</tr>
<tr>
<td>10.3.1 A Realistic Scenario</td>
<td>249</td>
<td></td>
</tr>
<tr>
<td>10.3.2 Keystones as Processes</td>
<td>250</td>
<td></td>
</tr>
<tr>
<td>10.3.3 Collaboration Patterns as Process Compositions</td>
<td>254</td>
<td></td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>10.4 The Application of our Framework</td>
<td>256</td>
<td></td>
</tr>
<tr>
<td>10.4.1 Enforcing Policies</td>
<td>260</td>
<td></td>
</tr>
<tr>
<td>10.4.2 Simulation</td>
<td>261</td>
<td></td>
</tr>
<tr>
<td>10.4.3 Exception Handling</td>
<td>261</td>
<td></td>
</tr>
<tr>
<td>10.4.4 Ontological Classes as Types</td>
<td>267</td>
<td></td>
</tr>
<tr>
<td>10.5 System Deployment: The DigiHealth Prototype</td>
<td>269</td>
<td></td>
</tr>
<tr>
<td>10.5.1 Model: A Relational Database</td>
<td>271</td>
<td></td>
</tr>
<tr>
<td>10.5.2 Controller: The PHP API</td>
<td>273</td>
<td></td>
</tr>
<tr>
<td>10.5.3 Workflow Execution: The Scala Implementation</td>
<td>274</td>
<td></td>
</tr>
<tr>
<td>10.5.4 View: A Web-based User-interface</td>
<td>278</td>
<td></td>
</tr>
<tr>
<td>10.6 Related Work</td>
<td>281</td>
<td></td>
</tr>
<tr>
<td>10.7 Ongoing and Future Work</td>
<td>283</td>
<td></td>
</tr>
<tr>
<td>10.8 Conclusion</td>
<td>284</td>
<td></td>
</tr>
<tr>
<td>II Conclusion</td>
<td>287</td>
<td></td>
</tr>
<tr>
<td>11.1 On the Achievements and Limitations of Our Work</td>
<td>287</td>
<td></td>
</tr>
<tr>
<td>11.1.1 Formalisation of Proofs-as-processes</td>
<td>288</td>
<td></td>
</tr>
<tr>
<td>11.1.2 Clarification of the Original Foundations</td>
<td>289</td>
<td></td>
</tr>
<tr>
<td>11.1.3 Logic-based Process Modelling Tools</td>
<td>289</td>
<td></td>
</tr>
<tr>
<td>11.1.4 Automated Workflow Deployment</td>
<td>290</td>
<td></td>
</tr>
<tr>
<td>11.1.5 A Variety of Applications</td>
<td>291</td>
<td></td>
</tr>
<tr>
<td>11.2 Future Work</td>
<td>291</td>
<td></td>
</tr>
<tr>
<td>11.3 Concluding Remarks</td>
<td>293</td>
<td></td>
</tr>
<tr>
<td>A Linear Logic Theorem Provers</td>
<td>295</td>
<td></td>
</tr>
<tr>
<td>B Extra Theorems and Definitions</td>
<td>297</td>
<td></td>
</tr>
<tr>
<td>C Application of the Commutativity Filter on a Process</td>
<td>299</td>
<td></td>
</tr>
<tr>
<td>D Code Excerpts</td>
<td>303</td>
<td></td>
</tr>
<tr>
<td>Bibliography</td>
<td>307</td>
<td></td>
</tr>
</tbody>
</table>
List of Figures

1.1 Our envisioned methodology for process modelling supported by logical proof. ............................................. 3
1.2 Thesis organisation. ......................................................... 10

2.1 The process lifecycle in BPM. .............................................. 14
2.2 Diagrams of the services involved in a credit card payment. ........ 17
2.3 Diagram of a composite service for a credit card payment. .......... 17
2.4 Diagram of the composition of the three services involved in a credit card payment. .............................................. 18

3.1 The $\pi$-calculus syntax in BNF grammar. ............................ 26
3.2 The PiVizTool visualisation of the Main process in example (3.4). . 31
3.3 The definition of the syntax for polyadic $\pi$-calculus terms in HOL Light. ......................................................... 33
3.4 The definition of the $FN$ function that returns the list of free names of a $\pi$-calculus term in HOL Light. ......................... 33
3.5 The definition of the substitution function for $\pi$-calculus. ............. 35

4.1 The equations used to define linear negation. .......................... 41
4.2 The one-sided sequent calculus versions of the CLL inference rules. . 43
4.3 The CLL inference rules annotated with process calculus proof terms and the corresponding $\pi$-calculus processes. ................. 45
4.4 The definition of the syntax for CLL terms in HOL Light. .............. 48
4.5 HOL Light definitions and abbreviations for each CLL connective. .. 49
4.6 The definition of the linear negation in HOL Light. ..................... 49
4.7 The definition of linear consequence in HOL Light. This introduces the CLL inference rules combined with their $\pi$-calculus translations. . 53
4.8 The interactive HOL Light scripts for the proof of TimesComm. ....... 57
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.9</td>
<td>The one-sided sequent calculus versions of the CLL inference rules for the exponentials.</td>
</tr>
<tr>
<td>4.10</td>
<td>The CLL inference rules for the exponentials annotated with process calculus proof terms and the corresponding $\pi$-calculus processes.</td>
</tr>
<tr>
<td>4.11</td>
<td>The inference rules of ILL.</td>
</tr>
<tr>
<td>4.12</td>
<td>The $CP$ syntax in BNF grammar.</td>
</tr>
<tr>
<td>5.1</td>
<td>Diagrams of the available services for the Ski example.</td>
</tr>
<tr>
<td>5.2</td>
<td>Diagram of the requested service for the Ski example.</td>
</tr>
<tr>
<td>5.3</td>
<td>Excerpt of the DAML-S web service profile for the “Congo” book service (Ankolekar et al., 2002).</td>
</tr>
<tr>
<td>5.4</td>
<td>The Ski example translated into ILL with proof terms.</td>
</tr>
<tr>
<td>5.5</td>
<td>The grammar for Rao et al.’s process calculus which differs significantly from the $\pi$-calculus.</td>
</tr>
<tr>
<td>5.6</td>
<td>ILL inference rules combined with process calculus proof terms according to Rao et al.</td>
</tr>
<tr>
<td>5.7</td>
<td>Structural congruence rules for ILL with process calculus proof terms as introduced by Rao et al.</td>
</tr>
<tr>
<td>5.8</td>
<td>Part of the proof for the Ski example.</td>
</tr>
<tr>
<td>5.9</td>
<td>The resulting process calculus formula from the proof for the Ski example provided by Rao et al.</td>
</tr>
<tr>
<td>5.10</td>
<td>The description of the resulting composite service from the Ski example.</td>
</tr>
<tr>
<td>6.1</td>
<td>The available services and the request for the credit card example specified as CLL judgements.</td>
</tr>
<tr>
<td>6.2</td>
<td>The annotated CLL specifications of the available services for the credit card example and the corresponding $\pi$-calculus specifications.</td>
</tr>
<tr>
<td>6.7</td>
<td>A screenshot of our diagrammatic interface. The graph shows the current state of a proof in progress. The list on the left contains a list of all the atomic and composite services defined so far.</td>
</tr>
<tr>
<td>6.8</td>
<td>Diagrams of the available processes involved in the credit card example and the requested process for credit card payments.</td>
</tr>
<tr>
<td>6.9</td>
<td>Excerpt of the class hierarchy tree for the Pizza ontology.</td>
</tr>
<tr>
<td>7.1</td>
<td>Specialised versions of CLL inference rules.</td>
</tr>
<tr>
<td>7.2</td>
<td>Derived inference rules involving buffers.</td>
</tr>
</tbody>
</table>
7.3 The TENSOR action on two services $P$ and $Q$. .......................... 147
7.4 The proof tree and diagrammatic representation of the WITH action on
input $A$ of process $P$ and input $C$ of process $Q$. .......................... 148
7.5 The JOIN action between service $P$ with an atomic output and service
$Q$ with a matching input. .................................................. 151
7.6 The JOIN action between service $P$ with a composite output and service
$Q$ with two matching inputs. .......................................... 152
7.7 The JOIN action between service $P$ with a composite output and service
$Q$ with only one matching input. ....................................... 153
7.8 The JOIN action between service $P$ with an optional output and service
$Q$ with a matching input. .................................................. 154
7.9 The JOIN action between service $P$ with an optional output $X \oplus Y$ and
service $Q$ that only has $X$ as input and $Y$ as output. .............. 155
7.10 The JOIN action between service $P$ with an optional output $X \oplus Y$ and
service $Q$ that only has $X$ as input and $C \oplus Y$ as output. ........ 156
7.11 The two composition strategies for the credit card example visualised
diagrammatically. Both strategies have the same initial state and the
same result. ................................................................. 159
7.12 The CLL proof scripts constructed by the CreditCardPayment com-
position using Strategy A (7.14) and Strategy B (7.15). .............. 160
7.13 The resulting $\pi$-calculus terms for the CreditCardPayment composi-
tion for each of the two possible strategies. ............................ 161
7.14 The JOIN action between service $T$ and conditional composite service $S$. 168
7.15 The JOIN action between process $T$ and process $Q$. ................. 169
7.16 The CLL specification and corresponding diagrammatic representation
of the available services and the requested service for the Ski example. 171
7.17 The available services and the Request, Response, and Main pro-
cesses for the Ski example defined as $\pi$-calculus processes. ........ 171
7.18 The CLL proof of the requested service GetSki in the Ski example. ... 172
7.19 Diagrammatic representation of the workflow constructed with the proof
of the Ski example. ......................................................... 173
7.20 The resulting $\pi$-calculus formula from the proof of GetSki. ......... 173
7.21 Consecutive snapshots of the Ski example $\pi$-calculus result taken from
PiVizTool. ................................................................. 175
8.1 Type definitions in PAPPLIB
8.2 Implementation of a non-deterministic choice function in PAPPLIB
8.3 The general roadmap of the workflow deployment procedure
8.4 UML diagram of the generated trait for the atomic process ExampleProcess
8.5 UML diagram of the user-provided ExampleProcessInstance class
8.6 Template of generated Scala trait for a atomic process ExampleProcess
8.7 Generated code template for ExampleProcessInstance which inherits the ExampleProcess template and determines the functionality of the process. The user is merely required to fill in the code for the apply function.
8.8 UML diagram of the generated Scala class for the composite process CompositeExample
8.9 Template of generated class for composite process CompositeExample
8.10 Diagram of the CompositeExample process connected to the necessary pseudo-processes that allow it to be treated as a Scala function.
8.11 Implementation of the PiOut function for atomic output in PAPPLIB
8.12 Generated code for atomic output x of type A
8.13 Implementation of the PiIn function for atomic input in PAPPLIB
8.14 Generated code for atomic input of type A
8.15 Implementation of the PiTimes function for parallel output in PAPPLIB
8.16 Generated code for parallel output out = (x, y) where x and y are atomic outputs of type A and B respectively
8.17 Generated code for parallel output out = (x, (y, w)) where x, y, w are atomic outputs of type X, Y, and W respectively
8.18 Implementation of the PiPar function for parallel input in PAPPLIB
8.19 Generated code for parallel input input = (x, y) where x and y are atomic inputs of type A and B respectively
8.20 Implementation of the PiPlus function for optional output in PAPPLIB
8.21 Generated code for optional output out of either Left(a) or Right(b)
8.22 Implementation of the PiWith function for optional input in PAPPLIB
8.23 Generated code for optional input of either type A or B
8.24 Implementation of the PiId function for buffering in PAPILIB.

8.25 Code template for the $\pi$-calculus ‘tensor’ pattern $(\nu x, y)(\xi(x, y). (F \parallel G))$

in PAPILIB.

8.26 Implementation of the PiParI function for the CLL Par rule in PAPILib.

8.27 Code template for the $\pi$-calculus ‘par’ pattern $\pi(x, y). F$

in PAPILIB.

8.28 Implementation of the PiPlus1 and PiPlus2 functions for the two

CLL $\oplus$ rules in PAPILIB.

8.29 Code template for the $\pi$-calculus pattern corresponding to the CLL $\oplus L$

rule $(\nu x)(\xi(x) \cdot \lambda(x). P)$ in PAPILIB.

8.30 PAPILIB code template for the $\pi$-calculus ‘with’ pattern, i.e. the term

$(\nu u, v)(\xi(u, v). (u(x). P + v(y). Q))$.  

8.31 Implementation of the PiCut function for the CLL Cut rule in PAPILIB.

8.32 Code template for the $\pi$-calculus pattern corresponding to the CLL Cut

rule $\text{Cut}^+(F, G)$ in PAPILIB.

8.33 Generated Scala code for the available processes CreditCardInit

and UserPINInput of the credit card example.

8.34 Generated Scala code for the CreditCardTransaction process of the

credit card example.

8.35 Generated Scala code for the composite process CreditCardPayment.

The apply method is shown in Figure 8.36 as it could not fit within the

size of this page.

8.36 Generated Scala code for the apply method of the class corresponding

to composite process CreditCardPayment.

8.37 Generated Scala code template for the implementation of the avail-
able processes in the credit card example and the execution of the

CreditCardPayment workflow.

9.1 Web services for booking flights (1 of 2).

9.2 Web services for booking flights (2 of 2).

9.3 Web services for booking hotels.

9.4 Web services for renting a car (1 of 2).

9.5 Web services for renting a car (2 of 2).

9.6 Example of composing using CreditCardPayment as a component.
9.7 The CLL proof constructed by the composition of CreditCardPayment and FlightBooking with the JOIN tactic. ........................................... 236

9.8 Snapshot of the PiVizTool simulation of the execution of a composition containing CreditCardPayment. The composite service is automatically split into its components. ........................................... 237

9.9 The FlightSearchLowBudget exception handler. ........................................... 237

9.10 Using FlightSearchLowBudget as an exception handler for FlightSearch. If no exception is thrown, FL_LIST is merely forwarded (grey edge). ........................................... 237

9.11 Simulation of the BookAnyFlight composition using the PiVizTool: FlightSearch returns a list of available flights. ........................................... 238

9.12 Simulation of the BookAnyFlight composition using the PiVizTool: FlightSearch throws an EX_FL_NOT_FOUND exception. ........................................... 239

9.13 BookFlight, BookAnyFlight, BookHotel, and BookCar as composed services. ........................................... 241

10.1 Keystones represented as processes (1 of 3). ........................................... 251

10.2 Keystones represented as processes (2 of 3). ........................................... 252

10.3 Keystones represented as processes (3 of 3). ........................................... 253

10.4 Automated procedures used in the assignment and delegation patterns. 254

10.5 Diagram of the assignment pattern as a verified process composition. 257

10.6 Diagram of the delegation pattern as a verified process composition. 258

10.7 Diagrams of the assignment and delegation patterns, each as a single composite keystone. ........................................... 259

10.8 An intermediate snapshot of the simulation of the AssignmentPattern in PiVizTool. ........................................... 262

10.9 Diagram of the interhospital transfer pattern as a single composite keystone. ........................................... 265

10.10 Diagram of the ExtAssignmentPattern pattern as a composition of AssignmentPattern and InterhospitalTransferPattern. ........................................... 266

10.11 Diagram of the extended assignment pattern as a single composite keystone. ........................................... 267

10.12 Class hierarchy tree for the medicalsoa.owl ontology. ........................................... 268

10.13 System architecture for the deployment of the DigiHealth system. 270

10.14 Diagram of the ER model of the database for the DigiHealth system. 272
10.15 Constructor and execution method for the implementation of the keystone AssgResponsible. It involves an API call and a message retrieval in order to interact with the PHP server. 277

10.16 Embedding the implemented AssgResponsible keystone as a trait instantiation in the Scala workflow. 278

10.17 Sample screenshot from the prototype DigiHealth system. 279

D.1 Implementation code for BUFFER_TAC. 304
D.2 HOL Light proof script for the Cut’ rule. 305
D.3 A useful tactic for constructing filters. 306
D.4 HOL Light proof script for Strategy A of the composition for the credit card example. 306
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Mappings of CLL process specifications to $\pi$-calculus terms based on proofs-as-processes rules.</td>
<td>120</td>
</tr>
<tr>
<td>8.1</td>
<td>Mapping of $\pi$-calculus syntax to PiLib.</td>
<td>183</td>
</tr>
<tr>
<td>8.2</td>
<td>Mappings of $\pi$-calculus terms for atomic processes to corresponding Scala functions.</td>
<td>200</td>
</tr>
<tr>
<td>8.3</td>
<td>Mapping of proofs-as-processes $\pi$-calculus patterns to Scala functions in PAPPiLib.</td>
<td>211</td>
</tr>
<tr>
<td>10.1</td>
<td>List of available API calls in the DigiHealth system and the corre- sponding keystones they implement.</td>
<td>275</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Research in formal methods, and more specifically formal verification using logic and theorem proving, has been rapidly advancing over the past 40 years. Extended theories, in combination with powerful reasoning systems and proof assistants, are constantly being developed for the verification of increasingly complex systems. We believe that targeting such tools and methodologies towards practical, everyday, industrial scale applications and making them available and accessible to non-experts can have a large impact, especially in heterogeneous, distributed, or safety critical systems. The primary aim of our research is to combine and use the rich theory of proofs-as-processes and mechanical theorem proving to develop a pragmatic methodology for the development of trustworthy, correct by construction process workflows. We begin by describing the main involved concepts in the next sections.

1.1 Divide, Conquer, and Rule via Process Modelling

The conceptually simple but powerful concept of *divide et impera* or *divide and conquer* has been widely used over the past years for the modelling of large scale systems. From the breakdown of a complicated, collaborative, software project, to the business operations and infrastructure of a large enterprise, the key modelling principle involves the construction of a hierarchy of components that work independently, achieve individual tasks and goals, and communicate with each other. This breakdown facilitates top-down control and governance over the entire structure, adaptability and maintainability, separation of concerns, and increased efficiency.
Depending on the point-of-view and particular application, the modelled components are often named operations, (human or computer) agents, services, or more generally processes. In particular, considerable effort has been put towards the development of process modelling methodologies and tools in the past 10 years, especially as the technology is advancing and human and computer processes become increasingly intertwined. More importantly, the challenge lies not only in the modelling and standardization of the individual processes, but also in the modelling of the workflow, i.e. the execution order (control flow) and the means of collaboration and exchanged information between the processes (information flow). Workflow modelling and the analysis of the emergent properties of workflows has been a large area of interest for research and development both in academia and in the industry.

1.2 Trust Through Formal Verification

Another fast growing research area with constant new developments and increasing importance is that of formal methods. Formal methods are techniques used for software and hardware specification and verification based on mathematics and logic. The main effort is to validate that the software or hardware is correct in the sense that it is mathematically guaranteed to give the expected result based on its specification. The guarantees of correctness are particularly important in critical systems where an error or bug in the code may lead to unwanted results, for example in a life-support system or an emergency response system.

In our work, we focus on logical frameworks and the use of theorem provers for formal system verification. In this context, formal verification is accomplished by creating an abstract, logical specification of the desired system using a language that is embedded within a core logical framework. Mathematical proofs of the desired properties of the system are then developed using the methods and tactics of a theorem prover. These proofs provide guarantees of correctness of the constructed system specification. The specification is then translated to executable code with the same proven properties, thus resulting in an implemented system that is correct by construction.
1.3 Our Vision

As of now, few attempts have been made towards the formal verification of process models especially in a general, abstract way (as opposed to an approach tailored to a particular use case). The aim of the current research is to develop a generic process modelling methodology that allows the construction of rigorous, formally verified, correct by construction workflows with a number of mathematically guaranteed properties. We believe that an implementation of this methodology offers process based solutions to a variety of problems from different domains, combining the benefits of both process modelling and formal methods.

More specifically, we envision a particular methodology for process modelling that is supported by formal methods, as shown in Figure 1.1. Following a standard process modelling approach, an initially underspecified, “fuzzy” idea, policy, practice, or system is analysed and decomposed to smaller, independent, loosely specified processes (analysis stage). This stage is usually performed in close collaboration with domain experts and practitioners who can provide the necessary domain knowledge. Then, we develop formal, well-defined, logic based models for each of the smaller processes (modelling stage). Using rigorous means, the specified processes are composed to form a well-defined workflow for the original complex system (composition stage), which can then be deployed as an executable solution. A rigorous theoretical framework is used in the background throughout the process modelling and composition phases. Using logical inference, the framework provides guarantees of correctness and a variety of other verified properties for the end result.

![Figure 1.1: Our envisioned methodology for process modelling supported by logical proof.](image)

Therefore, we aim towards a methodology for formally verified composition of abstract processes that achieves a dual goal. On the one hand, the constructed, formally verified workflows are process-based solutions that are characterized by the following
properties:

- **Management and governance:** Models can be conceptually decomposed to increasingly smaller tasks that can be managed separately and easily. The smaller tasks are then composed using rigorous methods that allow full control over the workflow and the implementation of the desired workflow policies at every level.

- **Flexibility and scalability:** Process models are both abstract and composable and therefore adaptable to any context, both small and large scale.

- **Effectiveness and separation of concerns:** Each component in a process workflow can be implemented and optimised independently. This facilitates the monitoring and optimisation of the system effectiveness (for example by identifying bottlenecks) while any occurring issues can be dealt with locally without affecting the entire system.

- **Integration of technology:** Process workflow can consist of both human and computer based processes. As a result, a clear description of the interaction between humans and computers can be specified.

- **Simulation:** The capability of running simulations of process models helps identify issues such as bottlenecks and unhandled situations prior to a potentially expensive system deployment.

- **Maintainability:** The abstract nature of the specifications combined with the capability for automated deployment of a constructed model to executable code make the workflows adaptable to change. This is particularly useful in dynamic environments where the policies and structure of the process-based system change over time. It also facilitates efforts towards process optimisation.

On the other hand, the use of a logic-based methodology allows reasoning about the information flow of a constructed process workflow, and therefore provides guarantees of the following benefits:

- **Explicit, verified information/resource flow and information governance:** The constructed models aim to make the information and resource flow between component processes explicit and consistent, so that the modeller has full control of that flow and the policies that are applied to it.

- **Verified resource management and accounting:** Resources are managed and
accounted for automatically using a mathematical model. This facilitates the modelling task, especially when a large number of resources and cases are involved. For example, guarantees are provided that no information will be lost both during workflow composition and in the process communications during execution.

- **Type correctness of the composed process:** The correctness of the types of the inputs and outputs of the composed process is formally verified with respect to the type signatures of its atomic components. For example, connections between the outputs and inputs of any two components are created and tracked using rigorous means.

- **Efficiency and freedom of deadlocks and livelocks:** The end result of our process composition methodology is a workflow where the component processes are executed concurrently, therefore maximising efficiency, while we guarantee the model is inherently free of deadlocks (e.g. two processes waiting on the same resource and blocking each other) and livelocks (endless loops).

- **Explicit handling of exceptions:** Our logic based process specifications cater for possible exceptions and alternative outputs. These are accounted for explicitly and can either be handled internally within the workflow or forwarded to the end user.

- **Trust:** The mathematical guarantees for the properties described above provide a high level of trust to the final, implemented system. This is particularly important in cases where multiple independent stakeholders are involved.

- **Automated workflow deployment:** The formally verified composite workflow is described in process calculus terms which can be directly implemented in a programming language. Therefore, workflow deployment is automated and only depends on existing deployments of the component processes that satisfy their respective specifications.

We believe the combination of all of the featured described above, offers a unique, rigorous, and pragmatic solution that is particularly applicable to information critical systems. In healthcare, for example, information management plays a major role towards efficient, error free procedures. Lack of rigour in information exchanges in a busy hospital setting can cause preventable errors and delays. We have used our developed methodology to tackle such issues head on.
It must be noted that, even though our approach is founded upon a deeply theoretical logic framework, our vision is to develop a pragmatic and highly practical methodology and application. For this reason, we avoid delving deeply into aspects such as the meta-theoretical analysis of the properties described above, the formalisation of the soundness and completeness proofs of the logical background, or the investigation of potential theoretical advances. Instead, we trust the work of the architects of our theoretical foundations that we use and focus on dealing with practical issues towards achieving our goal of having a formally verified process modelling framework.

1.4 Our Approach: Assumptions and Outcomes

Our approach to formally verified process composition has been developed with the following set of assumptions:

1. There exists a set of atomic processes that are implementable, for example by interfacing with a variety of services including Web Services, business processes, and Human Provided Services (HPSs). Each available process can be described using a type specification of their inputs, outputs, preconditions, and effects (IOPEs). The specification allows for multiple IOPEs, both in parallel and optional (with exceptions being a common example of an optional output).

2. The methodology is agnostic to the inner working of the available processes, which are treated as black boxes. We assume the processes are well behaved and always satisfy their type specification.

3. Our argument about deadlock and livelock freedom in the composed workflows assumes the available processes always terminate.

Given these assumptions, our rigorous methodology allows for the composition of processes in a workflow. The logical core of our approach and its associated, rich theoretical background, provides guarantees of correctness for the constructed workflows. In particular, we claim the following unique benefits:

1. **Rigorous information flow:** The resource dependencies between the processes as introduced by the user are guaranteed to be enforced, and the modeller is allowed fine-grained control of the process execution order as well as which processes should execute in which cases. This minimizes the execution of unnec-
1.4. Our Approach: Assumptions and Outcomes

Essary processes and guarantees that all the necessary information and resources will be available before any step of the workflow is initiated.

2. **Systematic resource management:** The properties of our chosen logic disallow any implicit duplication or consumption (vanishing) of resources. This is particularly important as it greatly facilitates resource accounting during composition. This means that the user does not need to keep track of resources manually, especially when there is a large number of them. Therefore, we prevent “disappearing” or “magically appearing” resources, and all outcomes (including exceptions that are often forgotten about) must be handled explicitly.

3. **Type correctness during the process composition:** This guarantees the correct matching of types as processes are being composed and results in executable code that is typechecked in advance. It also provides a degree of consistency and continuity in the information flow in the sense that it guarantees that the involved resource types do not mutate during the workflow execution.

4. **Concurrent execution and deadlock and livelock freedom:** The composed workflow can be executed concurrently so that processes that independent components can be run simultaneously. At the same time, assuming the component processes always terminate, the theoretical background of our methodology guarantees deadlock and livelock freedom in our composed workflows. Concurrency is important for efficiency in a practical setting. Having deadlock and livelock freedom provides guarantees of termination for our composed workflows. However, given the limited expressivity in the workflow models (it is not possible to express loops), this result may be viewed as being relatively minor.

5. **Automated workflow deployment:** The result of our composition methodology is immediately translatable to executable code that can be used both for simulation purposes and in a production setting. This minimizes the time and cost of workflow deployment (excluding the implementation of the component processes). This is particularly important, especially in cases of workflow maintenance where minor updates in the design of the workflow take a considerable amount of time to be implemented in practice. In our case, this happens with the click of a button.

We consider these the main benefits of our process composition methodology and the main practical outcomes of our research. There are several other contributions, both
theoretical through the use of formal methods and theorem proving techniques and practical through the development of our tools and use cases. These are discussed throughout the thesis, based on the breakdown that we provide in the next section.

1.5 Achieving Our Goals - Thesis Outline

In our approach aimed at achieving our dual goal, we specify processes based on their inputs, outputs, preconditions, and effects (IOPEs) using a logic-based formalisation. Using the HOL Light theorem prover (Harrison, 1996a) and the theory of “proofs-as-processes” (Abramsky, 1994; Bellin and Scott, 1994) that links Classical Linear Logic (Girard, 1995b) and the π-calculus (Milner, 1999), we can construct compositions of processes based on logical inference. We also visualise the results as graphs and provide automated deployment capabilities (assuming the implementation of the bodies of the component processes).

We identify a set of 8 individual main components of our research. In this thesis, each chapter corresponds to one of these components. The diagram in Figure 1.2 shows the breakdown of the chapters according to general themes and the connections between them. We present the main components with a note of their respective chapters next:

1. **The formalisation of the π-calculus**, which forms the process language that describes our constructed compositions (Chapter 3).

2. **The formalisation of the proofs-as-processes paradigm**, which is the core theoretical background that allows us to create correct by construction workflows. We analyse the original proofs-as-processes paradigm, including Classic Linear Logic (CLL), the core logic of the theory, its mapping to the π-calculus, and the means to mechanise the proofs-as-processes paradigm so as to be able to perform proofs with it. (Chapter 4)

3. **The clarification of the foundations of our methodology**. This involves an analysis of previous work by Rao et al. (Rao et al., 2006), which formed the initial motivation for our research, from the standpoint of our formalised approach. (Chapter 5)

4. **The development of systematic specifications of processes within the context of proofs-as-processes**. This includes type specifications using CLL, commu-
1.5. Achieving Our Goals - Thesis Outline

5. The development of tools for the composition of logically specified processes both at the logical, theorem proving level and via a user-friendly, diagrammatic interface. (Chapter [7])

6. The development of automated deployment capabilities which allow our π-calculus workflows to be deployed as an executable system. (Chapter [8])

7. The analysis of various use-cases. We focus in particular on a use-case involving a holiday booking web agent. (Chapter [9])

8. The application of our methodology to the formal verification of collaboration patterns in healthcare. This is a direct, real-world application of our proofs-as-processes based process composition framework. We demonstrate the usefulness of our methodology in the context of healthcare and its potential use in large scale applications. (Chapter [10])

We aimed to develop each of these components in a generic, modular, and extendable way in order to maximize the applicability of our methodology but also allow a number of future extensions. We believe the resulting framework brings together a variety of theoretical developments and practical technologies and can be also be used as a basis for future developments and extensions in multiple directions.

This thesis also includes a literature review of the main research and application areas involved in the current project in Chapter [2] and concludes with a summary of the results and contributions of our work and our vision for potential further developments in Chapter [11]. Related and plans for future work will also be covered along the way, wherever contextually convenient.
Figure 1.2: Thesis organisation.
Chapter 2

Background

Our vision for the formal verification of process compositions brings to the fore two major research fields, namely process modelling and formal methods. In this chapter, we give a brief overview of the main developments in these two fields in order to set the background scene for our own research.

2.1 Process Modelling

Process modelling can be generally seen as the breaking down of a large software system or enterprise into small, independent, interacting activities. The same approach has been used in a variety of technologies, including software architecture (synthesis), multi-agent systems, concurrent process models, web services, enterprise architecture, and business process modelling. The atomic components have different names such as processes, agents, stakeholders, actions, activities, services, etc. depending on the context. In the general sense, our methodology is abstract enough to capture any of these concepts. For simplicity, we uniformly refer to system components as processes and the corresponding approach as process modelling.

The unifying root of all the various process modelling approaches is the philosophical notion of emergence (Lewes [1875]). In short, the properties of a complex system (whether software or otherwise) at the macro level emerge (arise) from simple interactions between individual, micro level components. Therefore, the appropriate composition of separately specified processes that have their own restricted sets of properties and rules, can result in a new, emerging set of properties and in capabilities that the
individual processes can not achieve on their own.

To put these concepts in context, we present the two main areas that we view as most closely related to our work, namely web services described in the next section and Business Process Modelling (BPM) described in Section 2.1.2. We believe our methodology is adaptable to deal with problems from both these areas (among others) and we demonstrate this through two specific applications presented in Chapters 9 and 10.

2.1.1 Web Services

Web services are defined as “self-contained, self-describing, modular applications that can be published, located and invoked across the Web” (Tidwell 2000) or “a software system designed to support interoperable machine-to-machine interaction over a network” (Haas and Brown 2004). They can perform various tasks of varying complexity and purpose. Research in the area has been focusing on methods for web service description, discovery, composition and execution. First a web service is developed and described with the appropriate specification. Then it is published on the web to be discovered by web service discovery tools. Once discovered, it is combined with other web services to form a composite service that can achieve a complex task. Finally, the composite service is executed to complete the cycle.

A plethora of tools, languages and protocols are under constant development for each of the steps. The most commonly used languages and protocols are the Universal Description, Discovery and Integration (UDDI) (Bellwood et al. 2003), the Web Services Description Language (WSDL) (Christensen et al. 2001), Simple Object Access Protocol (SOAP) (Box et al. 2000), and Representational State Transfer (REST) (Elkstein 2008).

- UDDI is a mechanism to facilitate Web Service discovery. It can be seen as the equivalent of a URI for Web Services.

- WSDL is a description language for Web Services that focuses on facilitating communication between services and integration with UDDI and SOAP/REST. It provides abstract descriptions of services while focusing on more concrete descriptions of the types of messages they exchange through their ports and their bindings to the communication protocols.
2.1. Process Modelling

- SOAP is a message specification that allows proper exchange of XML-encoded data. It was widely utilised as a communication protocol for the message exchanges between services, although it is gradually being replaced by the more flexible REST.

- REST is an architecture style for web services where communication is performed over the HTTP protocol. As such, it provides a more lightweight and flexible architecture compared to legacy protocols such as SOAP, and therefore makes the so called “RESTful” services easier to implement and more accessible.

There are also specification languages that include semantic information in the service description such as WSDL-S (Akkiraju et al., 2005), DAML-S (Ankolekar et al., 2002), and OWL-S (Martin et al., 2004). There have also been recent, on-going attempts to create a standard framework for the development, description and reasoning of web services. Examples are the Web Service Modeling Framework (WSMF) (Fensel and Bussler, 2002) and the METEOR-S project (Patil et al., 2004).

2.1.1.1 Service Oriented Architecture (SOA)

The development of all these web service technologies over the past years allowed a lot of flexibility in the design of modern large scale systems since they allow the composition of heterogeneous components. Based on this, the notion of Service Oriented Architecture (SOA) was developed (MacKenzie et al., 2006). In short, SOA is a software design paradigm where a software system consists of individual, loosely coupled components that provide different services. Among other things, this architecture facilitates reusability, modularity, interoperability, and maintainability. SOA can be seen as the evolution of program synthesis and it arguably revolutionised the design of large scale software systems globally.

2.1.1.2 Web Services Composition

For our research, we focus mainly on the third stage of the web services life cycle, i.e. the complex task of web services composition. It involves the appropriate combination of multiple web services in order to achieve a composite service that can perform a complex task. Since web services can be created on the fly or may fail to achieve their
described task and throw an exception, the composition system must be able to refine
the choices made dynamically. Moreover, composition needs to take into consideration
non-functional restrictions, including location, cost, and time. The complexity of the
task is compounded by the dramatic increase in available web services, as well as the
great variety of conceptual models used for the descriptions of the services.

2.1.2 Business Process Modelling

Business Process Modelling/Management (BPM) (Williams, 1967) has been used as
a major business management approach by a rapidly increasing number of companies
over the past 15 years. It involves the management of the business processes lifecycle
in an attempt to increase business effectiveness, efficiency, flexibility, and integration
with technology. It is most commonly seen within the context of Enterprise Architec-
ture (EA) (Ross et al., 2006) and as part of business optimisation.

The business process lifecycle, shown in Figure 2.1, contains the following main
stages:

1. **Design**: This involves the identification and specification of the individual pro-
cesses/components as well as the corresponding workflow that connects them
and governs their interactions.

2. **Modelling**: Unlike the notion of Process Modelling we presented so far, the Modelling stage in the context of BPM corresponds to the simulation of the designed processes and workflows in various modelled scenarios.

3. **Execution**: This stage, also referred to as **Deployment**, corresponds to the implementation of the business processes and workflows as a software system, its deployment within the organisation, and its execution as a real-world application.

4. **Monitoring**: During the execution stage, the state of the business processes is monitored in order to extract statistics and analytics and identify errors and bottlenecks.

5. **Optimization**: Making use of the results extracted in the modelling and monitoring stages, the process models are refined and optimised to further improve and enhance their functionality within the organisation.

Our formal methodology focuses on the first three stages, namely the design, modelling, and execution of process workflows, although the core benefits of formal verification are directly applied in the design stage.

It is also worth introducing the two core languages that are most commonly used in the context of BPM, namely the Business Process Model and Notation (BPMN) (Object Management Group, 2011) and the Business Process Execution Language (BPEL) (OASIS, 2007):

- **BPMN** is the most commonly used language for the design of business process workflows. It includes a large variety of concepts, including hierarchies of processes and stakeholders. This variety often becomes a source of criticism since it causes a high complexity with respect to implementation. However, it maximizes the flexibility when modelling different business scenarios. In combination with the existing graphical notation, if makes BPMN based models accessible and understandable by both technical developers and business analysts, therefore helping towards bridging the communication gap between the two.

- **BPEL** is an execution language used to map abstract business processes to concrete implementations. Its aim is to bridge the gap between the abstract specification and the implementation of a process by providing a concrete, executable model based on web services standards. As such, it focuses on handling the
various implementation issues for business processes. Even though it is not as expressive as BPMN, several attempts have been made to map abstract BPMN models to executable BPEL services, raising a lot of arguments about the differences between the two languages and, more generally, the different points of view between technical and business experts.

A number of existing companies (e.g. IBM, BonitaSoft, Appian, Casewise, etc.) focus their activities on BPM and EA. They have developed a number of tools relying on BPMN and its graphical notation. However, it is worth noting that a major role of these companies and their business experts is the analysis of the business architecture, which requires a close collaboration with the involved stakeholders in order to develop domain knowledge.

In our research, we have had the opportunity to perform similar analysis and modelling tasks in the context of the application of our methodology in the healthcare domain. We discuss this in Section 10.7.

In the next section, we present a simple example which, apart from demonstrating the properties of a process-based system, is also used throughout this thesis to demonstrate some of our results incrementally.

### 2.1.3 Motivating Example

As a simple example of a process-based system, we consider the case of a credit card payment. This process typically involves three parties or stakeholders, namely the buyer, the seller, and the bank. In the most common situation, the seller will provide the payment request to the buyer, who in turn will request a transaction from the bank. The buyer also has to supply the correct PIN or password in order to verify his identity and the validity of the transaction, and for the payment to go through.

This process can be modelled as a process-based system as follows:

- Assuming the buyer has agreed to proceed with the purchase, a service run by the seller will generate the payment request. Based on this information (and should the user choose to pay by a credit card), the `CreditCardInit` process generates a transaction order and a PIN request.

- The buyer can then use the `UserPINInput` process to respond to a PIN request.
and provide his PIN.

- The `CreditCardTransaction` process, run by the bank, processes a transaction order and a PIN. If the PIN is correct then the service completes the payment, otherwise it throws an `EX_BAD_PIN` exception.

![Diagram of the services involved in a credit card payment.](image)

Figure 2.2: Diagrams of the services involved in a credit card payment.

These three processes, represented diagrammatically in Figure 2.2, can be composed in a straightforward way so that the interactions between them are performed automatically and seamlessly. In fact, this is the usual case for everyday credit card transactions, where the user interacts with a single composite service that performs the entire payment process, rather than having to invoke the three services individually and explicitly. Given the payment request from the seller, the composite process should either perform the payment or throw an exception (in this simplified example it can only be an `EX_BAD_PIN` exception). The diagrammatic representation of the desired composite process `CreditCardPayment` is shown in Figure 2.3

![Diagram of a composite service for a credit card payment.](image)

Figure 2.3: Diagram of a composite service for a credit card payment.

Even though achieving this particular composition appears straightforward, there are a number of factors that need to be taken into consideration to ensure that the resulting service can be trusted, such as systematic resource accounting and appropriate exception handling. In our example, the `CreditCardTransaction` service must always be given the correct transaction information, and if it is further composed with some other service (e.g. one run by the seller that will complete the purchase and deliver it to the buyer) the case of a bad PIN exception must be accounted for rather than mistakenly assume that the payment will always go through.
The three available services can be composed to satisfy the specification of CreditCardPayment in a straightforward way as shown in Figure 2.4. Additionally, using a logic-based approach we can provide guarantees regarding the aforementioned factors, and, therefore, allow for a high degree of trust in the resulting composition in all cases.

Figure 2.4: Diagram of the composition of the three services involved in a credit card payment.

Having given a brief overview of the process modelling field and a small motivating example, we now proceed with setting the formal methods scene needed for our research.

2.2 Formal Methods via Theorem Proving

As mentioned in Section 1.2, formal methods generally involve techniques for software and hardware specification and verification based on mathematical logic. Broadly speaking, we can identify two main approaches to formal verification: theorem proving, which is the one we adopt for our research, and model checking. The latter involves automated methods to exhaustively check whether a model of a finite-state system complies with a given specification. It is commonly used to verify temporal properties specified using temporal logic formulas.

In our research, we attempt to exploit modern proof assistants for interactive formal verification via logical proof. This involves reasoning about a system using a logic based specification. Such specifications are often written in a language embedded within a higher-order logical framework. This will allow reasoning about both the system itself (object-level reasoning) or the language itself (meta-level reasoning).

There are two distinct types of such embedding (Boulton et al., 1992): in a shallow embedding the language constructs are written as higher-order logic predicates whereas in a deep embedding they are defined as a new distinct datatype. In the former case, the embedding is much simpler and allows direct and efficient reasoning about the algorithms based on this language using higher-order logic proof tactics. The case of a
deep embedding is more complex but allows reasoning about the language itself. For example, it allows the proof of properties such as soundness, meaning whether all the objects constructed by the language are valid, and completeness, i.e. whether every possible object of the domain can be described using this language.

Once the specification is complete, reasoning about the system is performed using the methods and tactics of a theorem prover or proof assistant (the latter usually referring more specifically to interactive tools). When trying to verify the correctness of a system within the logical framework of a theorem prover, it is important that system provides guarantees of soundness and correctness so that we are able to trust the obtained results. The LCF approach (Gordon et al., 1979) is often adopted by modern theorem proving systems for this purpose. This approach ensures every step made in a proof script is a verified logical derivation from a limited set of logical axioms.

In recent years, research in theorem provers has resulted in powerful reasoning systems which are capable of proving a huge variety of theorems in any formally specified subset of higher-order logic, either automatically or interactively. A lot of automated tactics have been implemented based on decision procedures (McLaughlin and Harrison, 2005), model elimination (Loveland, 1968), tableaux algorithms, counterexample checking, and so on. We provide a short discussion of some of the existing proof assistants in Section 2.2.1 while we focus on HOL Light as our framework of choice as described in Section 2.2.2. We conclude with a brief overview of existing efforts towards the use of formal methods for the verification of process workflows in Section 2.2.3.

### 2.2.1 Modern Theorem Provers

- The HOL family of theorem proving systems includes some of the first implementations of the LCF approach. They rely on the use of an abstract data type for theorems, so that objects of this type can only be created using the primitive logical axioms. The main available tools are divided into rules for forward reasoning and tactics for backward reasoning, whereas tacticals can be used to combine series of tactic applications together. The original HOL88 (Gordon, 1991) was implemented in Common Lisp, whereas later HOL provers used standalone implementations of the ML programming language (Paulson, 1996). Apart from HOL Light, which we describe in more detail in Section 2.2.2, one of the most
recent, advanced, and under active development HOL proof assistant is HOL4 (Slind and Norrish [2008]).

- Isabelle/HOL (Paulson [1994]) is a well known and widely used Higher Order Logic theorem prover written in Standard ML. It is by nature an interactive theorem prover, allowing the use of either a procedural proof script or the more readable, declarative Isar environment (Wenzel, 2002). However, it also includes a number of automated tactics and decision procedures such as its simplifier and its auto and blast methods. It can also cooperate soundly with external theorem proving systems via its so-called sledgehammer interface (Meng et al., 2006), thereby achieving further proof automation. Overall, it is a sound system, widely accepted by the theorem proving community. The core proof tactics developed in our framework (see Section 4.4.5.1) are inspired by the standard procedural tactics used in Isabelle/HOL.

- Coq (Bertot et al., 2004) is a well known intuitionistic theorem prover based on the Calculus of Inductive Constructions and written and compiled in OCaml. It uses LCF-style tactics for interactive proofs but also has its own decision procedures and automated methods. SSReflect (Gonthier et al., 2008), in particular, is a library of extensions to Coq aimed towards supporting the proof methodology of small-scale reflection. It provides significantly improved functionality in Coq, including improved proof layout, structure, and management. At the programming level, Coq provides its user with Ltac, a language for the creation of tactics. This can be viewed as an intermediate level between Ocaml and the Coq toplevel, with various tools such as pattern matching and goal matching available. It is possible to write tactics at the OCaml level but it is generally avoided in favour of the much more user-friendly Ltac. Similarly to the other two proof assistants, Coq allows both deep and shadow embedding of logics. The most relevant example in our case, are the embeddings of linear logic by Power et al. (Power et al., 1999) and Sadrzadeh (Sadrzadeh, 2003) which we used as a basis for our own embedding in HOL Light (see Section 4.4).

It is worth mentioning that there are other powerful theorem provers available such as PVS (Shankar et al., 1999), but we will not be discussing these here as they are not immediately relevant to our project. In addition, various automated tools for proving theorems in linear logic (the core logic used in our work) have also been developed. We discuss a few of these in Appendix A.
2.2.2 HOL Light

HOL Light (Harrison, 1996a) is a relatively recent member of the HOL family of theorem provers that was initially built in an attempt to overcome certain disadvantages of its predecessors.

The system has equality as the only primitive concept and a few primitive inference rules that form the basis of more complex rules and tactics. Built on top of these, HOL Light has its own automated methods for proofs such as the model elimination method MESON (Harrison, 1996b). Additionally, it has an array of conversion methods that allow for very efficient and fine-grained manipulation (such as rewriting or numerical reduction) of formulas. The system is based on the LCF approach, which, in practice, guarantees that any proved theorem is a logical consequence of the primitive axioms.

HOL Light is written in OCaml, which allows for a fruitful interaction at every level. This allows for easier implementation and integration of techniques and tools that can seamlessly interact with the internals and methods of HOL Light. Moreover, a library of theories such as number and set theory has already been established, proved and are available for use within HOL Light. Finally, HOL Light has an array of powerful tools, tactics, and decision procedures that can help with advanced formalization tasks such as in deep embedding and formal verification tasks.

These properties and capabilities have proven particularly useful in the context of the current work, making HOL Light a flexible framework for the development of our system. The advantages of using HOL Light and more of its capabilities will be discussed further in subsequent chapters.

2.2.3 Formal Verification of Process Workflows

A variety of formal methods, both from the theorem proving and the model checking perspectives, has been used for the verification of process workflows in the past. In particular, there have been research efforts to verify various properties of BPMN process models based on the model checker SPIN and its associated meta language Promela (Holzmann, 2004; Kherbouche et al., 2013; Yamasathien and Vatanawood, 2014). Another approach involved the use of the mathematical language and proof methods of Event-B, a framework for modelling and reasoning about systems (Abrial, 2010; Bryans and Wel, 2010). Finally, process algebras such as the π-calculus (Milner,
which is also used in the current work (see Chapter 3), and related mathematical languages such as Petri Nets (Murata 1989), have been closely associated with modelling and reasoning about process-based, distributed systems.

These efforts have made noteworthy progress towards the rigorous analysis of process workflows and offer distinct advantages, for example towards the verification of quantitative and temporal properties. Our current approach differs significantly from the static verification offered by these frameworks. Instead of analysing an existing system, we aim to develop correct-by-construction workflows from scratch. Our methodology allows the construction of an abstract, mathematical model, guarantees a number of properties about the model, and results in a deployed, executable workflow with the same properties.

Despite the difference in the general approach, we believe there can be synergies between existing tools and techniques and our logic-based workflow composition methodology, that could lead to an enhanced set of formally verified properties for the constructed workflows. However, the investigation of such synergies is beyond the scope of the current work.

2.3 Conclusion

Our research involves two main areas, namely process modelling and theorem proving. Both these areas are under constant development and their applications are used both in the academia and industry.

The process modelling approach is applied to a variety of different types of systems. Even though our methodology considers abstract processes that can be instantiated to any type of system, we focus on two particular, widely used applications, namely web services and BPM. Web services and SOAs are increasingly becoming the norm in modern software architectures as they offer increased accessibility and interoperability between heterogeneous systems. BPM, for its part, is an Enterprise Architecture approach that adds significant value to a business by breaking down its structure and enabling, among other things, optimisation, monitoring, and smooth technology integration through business process workflows. Providing a formally verified methodology for both these cases, and therefore introducing a level of trust for the correctness of the constructed workflows, offers significant advantages that are not widely available
2.3. Conclusion

as of this writing.

Theorem proving, for its part, has seen a continuous development of increasingly powerful tools for the formal verification of both hardware and software. These tools are the embodiment of matured logic-based theories that have been developed for more than 40 years, and provide a solid and flexible foundation for the formalisation of our work towards the goal of formal verification of process composition.
Chapter 3

The \(\pi\)-calculus

In this chapter, we present the \(\pi\)-calculus, a process calculus used in the proofs-as-processes paradigm, which is at the core theory of our approach. In particular, we focus on its theoretical basis, the available tools, and its embedding in HOL Light.

3.1 Introduction

The \(\pi\)-calculus is a formalism aimed at the description of concurrent processes (Milner, 1999). The name is used to show the connection to the \(\lambda\)-calculus as a minimal, abstract representation. In the \(\pi\)-calculus, processes are described atomically as independent entities. They communicate asynchronously by message passing. More specifically, this can be viewed as passing an untyped message from a process with an output port to a process with an input port of the same name by forming a common channel between the two ports.

It is worth noting that the concepts of a message, a port, and a channel in the \(\pi\)-calculus are synonymous and interchangeable, since they are all represented as variables or names. This means ports can be communicated as messages, therefore making it possible to model dynamic communication protocols.

We focus on the polyadic version of the \(\pi\)-calculus (Milner 1991). Essentially, this version allows for a vector of any number of messages (including none) to be communicated instead of just one as is the case for the monadic version. However, this is only a syntactic discrepancy, as the two versions have been shown to have equivalent
Chapter 3. The \( \pi \)-calculus

\[
P ::= x(\vec{y}).P \quad \text{(1)}
\]
\[
| \ x(\vec{y}).P \quad \text{(2)}
\]
\[
| \ P \ || \ P \quad \text{(3)}
\]
\[
| \ (\nu x)P \quad \text{(4)}
\]
\[
| \ !P \quad \text{(5)}
\]
\[
| \ 0 \quad \text{(6)}
\]
\[
| \ P + P \quad \text{(7)}
\]

(3.1)

Figure 3.1: The \( \pi \)-calculus syntax in BNF grammar.

The \( \pi \)-calculus has inspired a variety of process algebras, used to describe the communication between agents, as well as BPM languages such as BPEL, and has been employed as the means to formalise their semantics (Lucchi and Mazzara, 2007).

More importantly, in our case, it has been used to describe the correspondence of linear logic proofs to concurrent processes in the proofs-as-processes paradigm (see Chapter 4). As a result, in the context of process composition via proof, the \( \pi \)-calculus provides a language to describe the constructed composite processes.

We give a quick overview of the main elements of the \( \pi \)-calculus in Section 3.2. We then provide a brief description of some of the available, useful tools for the analysis of concurrent processes in Section 3.3. Finally, our embedding of the \( \pi \)-calculus in HOL Light is described in Section 3.4.

### 3.2 Description

Figure 3.1 shows the syntax of the \( \pi \)-calculus as a BNF grammar. In this, (1) depicts a channel \( x \) attached to process \( P \) that allows it to receive a message that will be bound to the vector of names \( \vec{y} \). Similarly, (2) depicts a process with channel \( x \) that can send a message through name \( \vec{y} \) as output. It is worth noting that the infix dots in cases (1) and (2) are often omitted for simplicity. Expression (3) describes the parallel composition
of two processes whereas (4) describes a vector of names $\vec{x}$ that is local to $P$. Finally, (5) describes a replicable process $P$, often used to construct recursive process definitions, and (6) represents the “nil” process that has no functionality. Note that in some more minimal versions, the non-deterministic choice $P + Q$ (7) between two processes $P$ and $Q$ is excluded from the syntax. It is worth remarking also that there is no explicit representation of sequential processes.

Interactions between parallel processes are represented as reductions of the $\pi$-calculus terms (similar in form to the reductions of $\lambda$-calculus). The semantics of the $\pi$-calculus have deep theoretical roots and have led to long years of research towards important concepts such as bisimulation equivalence (or bisimilarity) (see (Sangiorgi, 1996)) which are beyond the scope of the current thesis.

We should make particular note of the use of vectors of names in cases (1), (2), and (4) since this is the polyadic version of the $\pi$-calculus (Milner, 1991). The original monadic version of $\pi$-calculus only allows single names in these cases. Bellin and Scott use the polyadic $\pi$-calculus in the inference rules for the proofs-as-processes paradigm (see Section 4.3 for more details). From here on, unless otherwise stated, every reference to the $\pi$-calculus implies the polyadic version.

Reductions of $\pi$-calculus terms are defined formally using a set of rules. We will only present the most significant reduction rule that describes the interaction between two parallel processes:

$$(\ldots + x(\vec{a}).F) || (\ldots + x(\vec{b}).C) \rightarrow F[\vec{b}/\vec{a}] || C$$

(3.2)

The communication described here happens between process $C$ with output $\vec{b}$ over channel $x$ and process $F$ with input $\vec{a}$ over the same channel $x$. The processes run in parallel (denoted by the “$||$” symbol) and for their interaction $C$ sends $\vec{b}$ to $F$ over $x$ yielding $F || C$ where each free occurrence of the names in $\vec{a}$ in $F$ is replaced by the names in $\vec{b}$. It must be noted that the above interaction is only allowed if the two involved vectors $\vec{a}$ and $\vec{b}$ have the same dimension, in addition to the two interacting processes having the same port $x$.

Moreover, in addition to the reduction rules, a set of rules is defined that allow us to express structural congruence ($\equiv$) relations between processes. For example, $P + 0 \equiv 0 + P \equiv P$, $P || 0 \equiv 0 || P \equiv P$ and $!P \equiv P || !P$ are all defined as structural congruence rules. In particular, the first rule can be used to yield a simpler version of the reduction
rule (3.2) which expresses the most common interaction between π-calculus processes, by replacing the missing, arbitrary π-calculus subterms denoted by the ellipsis (…) with 0. This results in the following equation:

\[(0 + x(\vec{a}).F) || (0 + \overline{x}(\vec{b}).C) \rightarrow F[\vec{b}/\vec{a}] || C\]

Given the congruence rule \(0 + P \equiv P\), this results in the following, commonly used π-calculus reduction:

\[x(\vec{a}).F || \overline{x}(\vec{b}).C \rightarrow F[\vec{b}/\vec{a}] || C\] (3.3)

This concludes a general overview of the π-calculus, its reductions and its intuitive interpretation. In the next section, we describe some of the available tools that can be used to analyse concurrent systems described using the π-calculus.

### 3.3 Tools

There are multiple available tools that perform a variety of tasks involving π-calculus terms. On the one hand, there are tools that exploit the π-calculus semantics and established theoretical background to provide reasoning capabilities for concurrent systems, whereas other tools offer environments to visualise and simulate the behaviour of a system described using the π-calculus. We give a brief overview and examples for each type of tools in the following sections.

#### 3.3.1 Reasoning about Concurrent Systems

The π-calculus with all its theoretical background offers a variety of capabilities to reason about concurrent systems. The most well known tool with this kind of functionality is the Mobility Workbench (MWB) ([Victor and Moller, 1994](#)). The MWB, implemented in Standard ML uses model checking techniques for the manipulation and analysis of concurrent agents described using π-calculus. Its main purpose is to perform checks for open bisimulation equivalences ([Sangiorgi, 1996](#)) which roughly corresponds to checking agents for equivalent behaviour. It also allows for step-by-step, textual simulation of a π-calculus system, and the discovery of deadlocks. There are
also other tools with similar functionality such as the ABC (Briais 2005) and PiET (Mio 2006).

There has been considerable effort to formalise the $\pi$-calculus and a number of derived process algebras using logical frameworks and proof assistants. Perhaps the most noteworthy and recent work in this direction is the introduction of Psi-calculi (Bengtson et al. 2009), i.e. process algebras derived by the $\pi$-calculus using nominal datatypes and their formalisation in Isabelle (Bengtson and Parrow 2009). The aim is to formally verify various properties of the Psi-calculi and to facilitate the verification of the same properties in further, arbitrarily extended versions. Taking into consideration that process algebras are often extended to accommodate the particularities of a specific domain, the importance of such a verification effort is obvious.

3.3.2 Visualisation and Simulation

The ability to visualise concurrent systems and their simulated executions using a simple, intuitive representation can prove a valuable tool. It not only allows us to better and faster understand and analyse the behaviour of these systems, but also provides the means to present this behaviour to non-experts and easily and intuitively explain the results of complicated reduction rules.

To our knowledge, the most simple and straightforward solution is the PiVizTool (Bog and Puhlmann 2006). Written in Java, the PiVizTool can graphically represent agents described in $\pi$-calculus. It also allows a step by step, user-controlled monitoring of the interactions in a multi-agent environment. PiVizTool has proven a valuable tool not only for a better understanding of $\pi$-calculus reductions but also for checking the expected behaviour of our formalised processes (see Section 7.6.3 for example). The result from composing a number of processes may often consist of a large $\pi$-calculus term that is difficult to process on paper or even with the MWB (see Figure 7.20 for an example).

It is worth noting that, in this project, we are using our own, slightly enhanced version of the PiVizTool, in order to better accommodate our visualisation and simulation needs. Given the minor but technical nature of the changes from the original tool, we will not go into further detail here.

In a PiVizTool visualisation, an elliptical node represents a $\pi$-calculus process (agent).
Dark, directed edges (with a black circle at the receiving end) represent available π-calculus reductions, i.e. information that can be exchanged from one process to the other through a common channel. Grey edges correspond to connections between processes that are currently blocked but may eventually become available. All edges are labelled based on the information being exchanged, and a mouse hover above any edge displays a tooltip with the name of the involved channel. The user is free to select any of the available reductions as the next execution step. This may lead to new connections being formed (or others no longer being available as in the example of an optional connection), new execution steps becoming available, and new processes appearing in the graph as they become involved in subsequent steps.

To give a demonstration of a PiVizTool visualisation, let us consider a simple example of a π-calculus system:

\[
A(\text{out},x) = \overline{\text{out}}(x).A(\text{out},x) \\
B(\text{in},\text{out}) = \text{in}(x).\overline{\text{out}}(x).B(\text{in},\text{out}) \\
C(\text{in}) = \text{in}(x).C(\text{in}) \\
\text{Main()} = A(c_a,i) || B(c_a,c_b) || C(c_b)
\]

(3.4)

In this example, \(A(\text{out},x)\) is a process that continuously sends information \(x\) over channel \(\text{out}\), \(C(\text{in})\) is a process that continuously receives information through channel \(\text{in}\), and \(B(\text{in},\text{out})\) is a buffer process that continuously buffers information from channel \(\text{in}\) to channel \(\text{out}\). The \(\text{Main()}\) process composes all three in parallel by introducing channels \(c_a\) and \(c_b\) for the intermediate communication, as well as a name \(i\) that represents the exchanged information.

Note that the processes run endlessly based on their recursive definition. This is a typical way of introducing π-calculus processes in order to analyse the behaviour of a continuously running concurrent system. This may involve, for example, always online servers and clients that are continuously making requests.

The PiVizTool visualisation of the \(\text{Main}\) process is shown in Figure 3.2. More specifically, the initial state is shown in Figure 3.2a. This corresponds to the following π-calculus term (which can also be inspected in its textual form in the PiVizTool if the user chooses to do so):

\[
\overline{c_a}(i).A(c_a,i) || c_a(x).\overline{c_b}(x).B(c_a,c_b) || c_b(x).C(c_b)
\]
3.3. Tools

(a) Initial state

(b) After 1 execution step.

Figure 3.2: The PiVizTool visualisation of the Main process in example (3.4). The current state includes one available execution step, namely the communication of $i$ from $A$ to $B$. Hovering above the $i$ edge reveals the corresponding communication channel, namely $c_A$. Note that the PiVizTool has detected the connection of $B$ to $C$ through channel $c_B$, but this is currently blocked (grey edge) pending the communication between $A$ and $C$.

Clicking on the $i$ edge executes this communication and results in the graph of Figure 3.2b. The status bar of the tool provides information on the executed step:

**Step 1** - Sending from $A\#0$: $c_A(i)$ Sending to $B\#0$: $c_A(x)$

The $\#0$ tag next to $A$ and $B$ is used by the PiVizTool to ensure freshness of names. The bound name $x$ of $B$ has now been substituted by $i$ according to the reduction rules of the $\pi$-calculus. This essentially corresponds to the following $\pi$-calculus term:

$$A(c_A, i) \parallel \sigma_{c_B(i)} . B(c_A, c_B) \parallel c_B(x) . C(c_B)$$

The communication between $B$ and $C$ through channel $c_B$ is now available and can be chosen for execution. Given the recursive definitions of our processes, executing this step will return us to the initial state of Figure 3.2a for the next round of communication.

As the modelled systems scale up and the corresponding $\pi$-calculus terms dramatically increase in size, the PiVizTool greatly facilitates the visualisation of their execution and the analysis of their behaviour, especially when there is a choice between different possible executions.
In the next section, we describe our efforts towards the formalisation of the $\pi$-calculus in HOL Light, for the specific needs of formally verified process compositions using the proofs-as-processes paradigm.

### 3.4 Embedding the Polyadic $\pi$-calculus in HOL Light

The first step towards formalising the proofs-as-processes paradigm in HOL Light involves the mechanisation of the $\pi$-calculus within HOL Light. Such a task could include the formalisation of the $\pi$-calculus syntax, variable substitution, term reductions and then the theory of bisimulation. The main issue of such a formalisation would reside in dealing with variable binding and substitution as they greatly affect the complexity of the involved proofs.

There have been multiple attempts to formalise the $\pi$-calculus within a theorem prover. For example, in (Melham, 1992) the $\pi$-calculus is deeply embedded within HOL88 using a purely definitional approach. Moreover, Röckl formalises the $\pi$-calculus in Isabelle/HOL using Permutations in (Röckl, 2001) and Higher Order Abstract Syntax (HOAS) in (Röckl et al., 2001). More recently there has been an attempt towards formalising $\pi$-calculus in Isabelle/HOL using Nominal Logic (Bengtson and Parrow, 2007).

In the current work, we believe that a limited formalisation which includes the $\pi$-calculus syntax and variable substitution is sufficient to accomplish our most important tasks. We decided to follow the approach of Melham since it is not only simple and straightforward, built in a system similar to HOL Light, but it is also independent of any particular theories or tools of the theorem prover (as opposed, for example, to the work of Bengtson and Parrow that utilises the Nominal Logic theory in Isabelle/HOL).

#### 3.4.1 Syntax in HOL Light

The main issue in following Melham’s approach, though, is that it formalises the monadic $\pi$-calculus. Extending the formalisation to the polyadic one is simple with regards to the syntax and definitions but greatly complicates the involved meta-theoretic proofs. In particular, we introduced lists of names to represent vectors. Our definition of the syntax for the polyadic $\pi$-calculus terms is shown in Figure 3.3. We note that the
defined type ‘(A)Agent’ is polymorphic, with the type of the involved names as a parameter ‘A’ similarly to Melham. In our concrete applications we simply used natural numbers for names (see Section 3.4.3).

let piCalc_INDUCT,piCalc_RECURSION = define_type

"Agent = Zero
| Out A (A list) Agent
| In A (A list) Agent
| Res (A list) Agent
| Comp Agent Agent
| Plus Agent Agent
| Repl Agent";;

Figure 3.3: The definition of the syntax for polyadic \(\pi\)-calculus terms in HOL Light.

Compared to the \(\pi\)-calculus syntax given in Figure 3.1, Zero corresponds to 0, Out and In correspond to the sender and receiver processes respectively, and Res is the local binder \(\nu\). Moreover, Comp refers to the parallel composition of two processes, Plus to the non-deterministic choice, and Repl to replication of a process.

### 3.4.2 Functions about Names

We proceed to define functions \(FN\), \(BN\) and \(NAMES\) to return the list of free names, bound names and all names in a term respectively. We give the (pretty-printed) definition of \(FN\) in HOL Light as an example in Figure 3.4.

\[
\begin{align*}
(FN \ Zero & = [ ]) \land \\
(\forall x y P. \ FN \ (Out \ x \ y \ P) & = \ CONS \ x \ (APPEND \ y \ (FN \ P))) \land \\
(\forall x y P. \ FN \ (In \ x \ y \ P) & = \ CONS \ x \ ((FN \ P) \ LDIFF \ y)) \land \\
(\forall x P. \ FN \ (Res \ x \ P) & = (FN \ P) \ LDIFF \ x) \land \\
(\forall P Q. \ FN \ (Comp \ P \ Q) & = \ APPEND \ (FN \ P) \ (FN \ Q)) \land \\
(\forall P Q. \ FN \ (Plus \ P \ Q) & = \ APPEND \ (FN \ P) \ (FN \ Q)) \land \\
(\forall P. \ FN \ (Repl \ P) & = FN \ P)
\end{align*}
\]

Figure 3.4: The definition of the \(FN\) function that returns the list of free names of a \(\pi\)-calculus term in HOL Light.
The main difference from Melham’s work, which focuses explicitly on the monadic version of the \( \pi \)-calculus, is that our functions return lists of names rather than sets because we use lists of names as vectors. Therefore, returning sets would force us to convert these lists into sets within the function definition and this makes both reasoning about and evaluating the function in HOL Light surprisingly more complicated.\(^1\) However, we have formally verified that the functionality of our list-returning functions is equivalent to that of Melham’s set-returning functions. Moreover, we proved a variety of meta-theorems involving interesting properties about these functions.

### 3.4.3 Variable Substitution

The main issue involved in formalising variable substitution is the alpha-renaming of bound variables to fresh variables. Melham uses a choice function, \( ch \), that for any given name \( n \) generates a fresh name not included in a list \( l \) of given names as a parameter to variable substitution. It must be ensured that a given function always generates fresh names or, more formally, that it satisfies the following specification:

\[
\neg (\text{MEM} (ch n l) l) \quad (3.5)
\]

Notice that here too we use lists instead of sets for the same reasons as explained in Section 3.4.2. This specification is necessary for the proof of some meta-theorems. Its definition is general so as to be able to specify a choice function for any infinite type of names being used (e.g. natural numbers, strings, etc).

To give a more concrete example of such a choice function, we can define the function \( \text{NCH} \), with names expressed as natural numbers, to be the successor of the maximum of the list of given numbers, or more formally:

\[
\text{NCH} \ n \ l = \text{SUC} (\text{ITLIST} \ \text{MAX} \ n \ l)
\]

Note that the \( \text{ITLIST} \) function iterates over the list \( l \) using the \( \text{MAX} \) function to find the maximum of the list. We then prove that \( \text{NCH} \) satisfies the choice function specification (3.5), i.e. that it always produces fresh names.

\(^1\)Such evaluations are usually implemented as conversions in HOL Light that allow the reduction of a function to its value.
Following Melham’s definition, we define a substitution function $\text{SUB}$ in HOL Light as shown in Figure 3.5. The function, which has three arguments, namely a choice function, an agent and a substitution, returns the resulting agent.

\[
(\forall s. \text{SUB} \, \text{ch} \, \text{Zero} \, s = \text{Zero}) \land \\
(\forall x y P s. \text{SUB} \, \text{ch} \, (\text{Out} \, x \, y \, P) = \text{Out} \, (s \, x) \, (\text{MAP} \, s \, y) \, (\text{SUB} \, \text{ch} \, P \, s)) \land \\
(\forall x y P s. \text{SUB} \, \text{ch} \, (\text{In} \, x \, y \, P) s = \\
\quad \text{let } vs = \text{MAP} \, ((\text{FN} \, P) \, \text{LDIFF} \, y) \, \text{in} \\\n\quad \text{let } y' = \text{MAP} \, (\lambda z. \text{if} \, (\text{MEM} \, z \, vs) \, \text{then} \, (ch \, z \, vs) \, \text{else} \, z) \, y \, \text{in} \\\n\quad \text{let } s' = \text{SUBLF} \, y \, y' \, \text{in} \\
\quad \text{In} \, (s \, x) \, y' \, (\text{SUB} \, \text{ch} \, P \, (\lambda x. \text{if} \, (\text{MEM} \, x \, y) \, \text{then} \, s' \, x \, \text{else} \, s \, x))) \land \\
(\forall y P s. \text{SUB} \, \text{ch} \, (\text{Res} \, y \, P) s = \\
\quad \text{let } vs = \text{MAP} \, ((\text{FN} \, P) \, \text{LDIFF} \, y) \, \text{in} \\\n\quad \text{let } y' = \text{MAP} \, (\lambda z. \text{if} \, (\text{MEM} \, z \, vs) \, \text{then} \, (ch \, z \, vs) \, \text{else} \, z) \, y \, \text{in} \\\n\quad \text{let } s' = \text{SUBLF} \, y \, y' \, \text{in} \\
\quad \text{Res} \, y' \, (\text{SUB} \, \text{ch} \, P \, (\lambda x. \text{if} \, (\text{MEM} \, x \, y) \, \text{then} \, s' \, x \, \text{else} \, s \, x))) \land \\
(\forall P Q s. \text{SUB} \, \text{ch} \, (\text{Comp} \, P \, Q) s = \text{Comp} \, (\text{SUB} \, \text{ch} \, P \, s) \, (\text{SUB} \, \text{ch} \, Q \, s)) \land \\
(\forall P Q s. \text{SUB} \, \text{ch} \, (\text{Plus} \, P \, Q) s = \text{Plus} \, (\text{SUB} \, \text{ch} \, P \, s) \, (\text{SUB} \, \text{ch} \, Q \, s)) \land \\
(\forall P s. \text{SUB} \, \text{ch} \, (\text{Repl} \, P) s = \text{Repl} \, (\text{SUB} \, \text{ch} \, P \, s))
\]

Figure 3.5: The definition of the substitution function for $\pi$-calculus.

The most interesting cases of this definition are those of the input $\text{In}$ and local binding $\text{Res}$, because of the involvement of bound names. In theses cases, bound names are substituted (using $\text{SUBLF}$ described below) by fresh names ($y'$ - using the choice function $ch$) if they match ($\text{MEM}$) any of the names $vs$ that result from the application ($\text{MAP}$) of the substitution $s$ to the free names ($\text{FN}$) of the recursive process $P$ (excluding the list of bound names $y$). This ensures bound names are kept fresh and distinct regardless of the effect of the substitution. Note that since we are formalising the polyadic version of the $\pi$-calculus, $y$ is a list that may contain any number of names (as opposed to the monadic version where $y$ is a single name). For this reason, we have defined the substitution function $\text{SUBLF} \, k \, l$ that replaces any name from a list of names $k$ with the corresponding name from another list $l$ as follows:

\[
\text{SUBLF} \, [\, ] \, l = \lambda x. x \land \\
\text{SUBLF} \, (\text{CONS} \, h \, t) \, l = \lambda x. \text{if} \, x = h \, \text{then} \, \text{HD} \, l \, \text{else} \, \text{SUBLF} \, t \, (\text{TL} \, l) \, x
\]

A slightly more readable and usable, derived definition that can be easily proven (but
Chapter 3. The π-calculus

is not a valid HOL Light definition) is the following:

\[ SUBLF [ ] [ ] = \lambda x. x \land \]
\[ SUBLF (CONS h_1 t_1) (CONS h_2 t_2) = \lambda x. \text{if } x = h_1 \text{ then } h_2 \text{ else } SUBLF t_1 t_2 x \]

Proving meta-theorems about variable substitution in polyadic π-calculus with the given formalisations is quite tedious. It involves complex inductive proofs with several intermediate lemmas. We only proved that substitution using the identity function as a substitution function results in the same agent, or more formally:

\[ \forall ch P. \text{SUB} ch P (\lambda x. x) = P \]

This theorem is useful for the derivation of a simplified version of the Cut rule from the proofs-as-processes paradigm, which in turn allows the simplification and improved efficiency of the produced process compositions as discussed in Section 7.2.4.

Given the limited need for further metatheoretic reasoning on π-calculus for our project we did not proceed with any further proofs at this stage. However, we believe that this simple result demonstrates a potential use of our framework for such meta-theoretical reasoning tasks should it be needed in the future.

3.4.4 Extra Theorems and Tools

In order to accomplish the π-calculus formalisation described, it is also necessary to extend the existing HOL Light theories for sets and lists. For lists, we introduced two new functions, namely DEL that deletes a particular member of the list and LDIFF that calculates the difference of two lists. Around forty new theorems about sets and lists were introduced and proved. We present these definitions and the most generally useful theorems in Appendix B. We also built a few conversions and other supporting functions.

3.5 Limitations and Future Work

As previously explained, the π-calculus formalisation presented in this chapter is a simple and practical embedding and only includes the core syntax and the functions
3.5. Limitations and Future Work

that are necessary for the formalisation of the proof-as-processes paradigm described in the next chapter. As such, even though this is a robust HOL Light formalisation, it lacks flexibility and usability outside this particular context. In particular, we focus on two limitations which demonstrate these disadvantages, namely the lack of flexible handling of fresh names and the inability to define \( \pi \)-calculus agents as recursive functions.

Proper handling of fresh names is a major issue in the embedding of languages that contain name bindings (such as the binding of \( x \) in \( a(x).0 \) in the \( \pi \)-calculus). Based on Melham’s definitional approach, our substitution function shown in Figure 3.5 handles bound names explicitly. Even though this is a valid definition, it is difficult to use both at a practical level when evaluating the application of the substitution function on large \( \pi \)-calculus terms, and at a meta-theoretical level when proving theorems about substitution. A more flexible approach involves the use of Isabelle’s Nominal Logic by Bengtson and Parrow (Bengtson and Parrow, 2007). In this case, the use of nominal types for names greatly simplifies the substitution definition since name bindings and fresh names are handled in the background. To our knowledge, there is no known formalisation of a library that can perform this functionality in HOL Light, whereas introducing a similar facility would considerably improve the flexibility of the current formalisation and might be an interesting future undertaking.

The second limitation involves agent definitions. In particular, \( \pi \)-calculus agents are most commonly defined recursively as a “function” over their free names, so that they perform a function continuously (for example as servers). Having \( \pi \)-calculus agents defined as recursive function constants in HOL Light would enhance the formalisation. For example, this would be particularly useful for the mechanization of the proofs-as-processes translation of CLL exponentials as discussed in Section 4.5.1.2 since they contain recursive agents. As argued by Melham, such a feature would require considerable “special-purpose proof support” (Melham, 1992). Although recursive translations are avoided in more recent evolutions of proofs-as-processes (see Section 4.5.2), adding this feature as an extension would make our \( \pi \)-calculus formalisation more expressive and complete.

Beyond these two main limitations, our \( \pi \)-calculus formalisation would benefit from the embedding of \( \pi \)-calculus reductions, congruence rules, and bisimilarity. Even though our main focus is not on reasoning about the \( \pi \)-calculus itself, a lot of interesting properties could potentially be investigated and proven, especially in the context of
the soundness and correctness proofs of the proofs-as-processes paradigm.

### 3.6 Conclusion

In this chapter, we gave a quick overview of the $\pi$-calculus, a language used to describe concurrent systems. This is not only an essential part of the proofs-as-processes paradigm as used in our project, but it is also widely used, either in the form described here or through a more complicated, derived process algebra, for the modelling and analysis of complex concurrent systems and processes.

In Section 3.2 we reviewed the syntax and some of the fundamental concepts in the $\pi$-calculus. Given the years of research put into its semantics, a more detailed account of all the involved concepts is beyond the scope of this work. However, there is ample literature for the interested reader e.g. (Sangiorgi and Walker, 2003) and several other papers referenced throughout this chapter.

A number of available tools, such as the ones presented in Section 3.3, demonstrate the reasoning capabilities and richness of the $\pi$-calculus as a modelling language, as well as provide an environment that facilitates the analysis and simulation of the modelled systems. This is of particular importance when it comes to the simulation and analysis of the large, complicated $\pi$-calculus terms used to describe the composite processes produced by our methodology.

Finally, in Section 3.4, we reviewed our efforts towards the formalisation of the $\pi$-calculus in HOL Light. Based on previous work done by Melham, we were able to formalise the $\pi$-calculus syntax and some basic properties. This is the first, fundamental step towards the formalisation of the proofs-as-processes paradigm, which is the core of our rigorous methodology for process composition. More details on the proofs-as-processes paradigm and its formalisation in HOL Light will be given in the next chapter.
Chapter 4

The Proofs-as-processes Paradigm

The proofs-as-processes paradigm is the core theory behind our methodology. It creates a connection between logical proofs and composite processes while providing an array of verified properties. In this chapter, we analyse this theory and our efforts towards its formalisation in HOL Light. Our main aim is to develop a flexible, practical proof system for Classical Linear Logic (CLL) proofs that facilitates the extraction of the corresponding processes.

4.1 Overview

The basic idea behind the proofs-as-processes paradigm involves a mapping between Classical Linear Logic (CLL) proofs and $\pi$-calculus terms representing processes. The mapping is very similar to the Curry-Howard correspondence (Howard [1980]) between intuitionistic logic proofs and the $\lambda$-calculus. Bellin and Scott analyse this mapping by giving a corresponding $\pi$-calculus term for the conclusion of each of the CLL inference rules. As the inference rules are being used in a CLL proof, a $\pi$-calculus term that corresponds to that proof is being built based on these mappings. At the end of the proof, it is ensured that applying the possible reductions in the resulting $\pi$-calculus term corresponds to the process of cut-elimination in the proof. This means that the cut-free version of the proof corresponds to an equivalent $\pi$-calculus term that cannot be reduced further. As a result, since $\pi$-calculus reductions correspond to process communications, any $\pi$-calculus process that corresponds to a CLL proof is inherently free of livelocks and deadlocks.
We begin our presentation of the proofs-as-processes paradigm with a description of Classical Linear Logic in Section 4.2. This is followed by an introduction to process calculus proof terms and an analysis of the CLL inference rules and their correspondence to the \( \pi \)-calculus in Section 4.3. Finally, we describe our approach to the formalisation of the proofs-as-processes paradigm as a logic embedded in HOL Light in Section 4.4.

### 4.2 Classical Linear Logic

Girard proposed linear logic as a refinement to classical logic (Girard, 1995b) in which the emphasis is not merely on the truth of a statement, as in classical logic, but also on formulas that represent resources. The classical rules of contraction and weakening are not allowed in linear logic and therefore assumptions cannot be ignored or copied. For example, if a constant \( A \) is assumed twice, it is considered a distinct case than when it is assumed once. In order to achieve a proof, all assumptions must be consumed as resources. In computer science, linear logic has been used as a direct and declarative approach to reasoning about various computational models related to processes such as Petri Nets (Murata, 1989).

For the particular purpose of specifying and composing processes, we use the multiplicative additive fragment of propositional CLL without units (MALL). This version of linear logic includes linear negation, multiplicative conjunction and disjunction, and additive conjunction and disjunction. These are sufficient for the specification of a variety of processes as we review in Section 6.2. We discuss other versions of linear logic and the reasons we are not using them for this work in Section 4.5.1. From here on, unless otherwise stated, every reference to CLL corresponds to MALL.

The semantics of the CLL operators have an intuitive interpretation where propositions correspond to resources, or in our case inputs and outputs of processes. We informally discuss this interpretation next:

- Linear negation (\( \perp \)) in CLL obeys similar laws to those of classical negation. Its defining equations, as shown in Figure 4.1, demonstrate a symmetry in CLL where each operator has a dual. In the context of resources, this duality can be exploited in order to represent the information flow (Bellin and Scott, 1994). More specifically, we assume a direction (or polarity) for each connective, either
\[(A ^ {\perp}) ^ {\perp} \equiv A\]
\[(A \otimes B) ^ {\perp} \equiv A ^ {\perp} \& B ^ {\perp}\]
\[(A \oplus B) ^ {\perp} \equiv A ^ {\perp} \otimes B ^ {\perp}\]
\[(A \Rightarrow B) ^ {\perp} \equiv A ^ {\perp} \& B \]
\[(A \& B) ^ {\perp} \equiv A ^ {\perp} \oplus B ^ {\perp}\]

Figure 4.1: The equations used to define linear negation.

as an input or as an output, so that dual connectives have opposite directions. Given the symmetry of CLL, this assumption can in principle be arbitrary. We choose to treat negated literals, multiplicative disjunction (\(\Rightarrow\)), and additive conjunction (&) as inputs, and positive literals, multiplicative conjunction (\(\otimes\)), and additive disjunction (\(\oplus\)) as outputs.

This kind of distinction of polarization of the CLL connectives is not new. It has been investigated in depth, for example by Laurent who introduces a variant of linear logic named Polarized Linear Logic (LLP) (Laurent, 2002). In LLP, the \(\Rightarrow\) and \(&\) connectives are considered negative, whereas the \(\oplus\) and \(\otimes\) connectives are positive. Moreover, it is demonstrated that all provable sequents in LLP have at most one positive formula. LLP has been shown to allow clearer mappings between classical logic and CLL and a more elegant computational interpretation in linear \(\lambda\)-calculus (Laurent and Regnier, 2003). In our intuitive interpretation of CLL formulas as process specifications we adopt similar restrictions to those of LLP (see Section 6.2).

It is worth noting that in the one-sided sequent calculus version of CLL which we examine here (see below), linear negation is defined at the syntactic level and not at the logical level, i.e. there are no CLL inference rules about linear negation in the one-sided version. In short, both \(A\) and \(A ^ {\perp}\) are considered atomic formulas and the de Morgan style equations of Figure 4.1 provide a syntactic equivalence of formulas involving negation (Troelstra, 1992). This can be contrasted to both standard Propositional Logic and the two-sided sequent calculus version of CLL, as in both these cases negation is defined as a logical unary connective with its respective inference rules.
• Multiplicative conjunction or the tensor operator \((A \otimes B)\) indicates a simultaneous operation which refers to the simultaneous or parallel output of \(A\) and \(B\). In order to prove a statement \(A \otimes B\), the set of available resources (inputs) must be split in two subsets, one that can achieve \(A\) and one that can achieve \(B\). Multiplicative conjunction can be seen as the counterpart of conjunction in classical logic.

For instance, looking back at the credit card example described in Section 2.1.3, the CreditCardInit service generates a transaction order \((\text{TRANSACTION})\) and a PIN request \((\text{PIN.REQ})\) simultaneously. The output of CreditCardInit can thus be represented in CLL as \(\text{TRANSACTION} \otimes \text{PIN.REQ}\).

• Additive disjunction or the plus operator \((A \oplus B)\) is similar in concept to exclusive disjunction in classical logic and indicates that either of \(A\) or \(B\) are produced but not both. When representing processes, the additive disjunction can be used to indicate alternative outputs. It can be used, for example, to express the possibility of a process throwing an exception instead of producing the expected result. It is worth mentioning that most process composition methodologies do not take exceptions into consideration explicitly.

In our example, the CreditCardTransaction process finalizes the payment \((\text{PAYMENT})\) if the provided PIN or password was correct, otherwise it throws a bad pin exception \((\text{EX.BAD.PIN})\). This output can be represented in CLL as \(\text{PAYMENT} \oplus \text{EX.BAD.PIN}\).

• Multiplicative disjunction or the par operator \((A \triangledown B)\), has a straightforward interpretation as the dual of multiplicative conjunction. It is used to represent simultaneous or parallel input. It is worth noting that, based on the \(\triangledown\) rule as shown in Figure 4.2, the sequent \(\vdash A, B\) is logically equivalent to \(\vdash A \triangledown B\).

As an example, the CreditCardTransaction process expects 2 inputs, namely a transaction \((\text{TRANSACTION})\) and the user’s PIN number \((\text{PIN})\). This can be expressed as a parallel input \(\text{TRANSACTION}^\perp \triangledown \text{PIN}^\perp\). However, in general we choose the logically equivalent expression as separate atomic inputs \((\text{TRANSACTION}^\perp, \text{PIN}^\perp)\) in order to allow maximum flexibility for automation and code generation. This is actually imposed explicitly in our CLL process specifications (see Section 6.2).

• Additive conjunction or the with operator \((A \& B)\), can similarly be interpreted
as the dual of additive disjunction, i.e. the representation of optional input.

For example, assuming a process that can handle either of the two possible outcomes of `CreditCardTransaction`, namely `PAYMENT` and `EX_BAD_PIN`. Its optional input would be expressed in CLL as `PAYMENT ⊥ & EX_BAD_PIN ⊥`, allowing to receive either of the two resources at runtime, but not both.

Generally, a two-sided sequent calculus is used for the representation of the CLL inference rules. The left and right versions of each inference rule serve the purpose of handling a connective in the left or right hand side of the turnstile respectively. However, given the observation that \( \Gamma \vdash \Delta \) is equivalent to \( \vdash \Gamma⊥ , \Delta \) we can eliminate half of the rules by using a one-sided sequent calculus representation. This has an important impact in the automation of CLL proofs, as the number of available inference rules in the proof search is effectively halved. Moreover, Bellin and Scott use a one-sided sequent calculus representation for CLL in their work as well.

The one-sided sequent calculus versions of the inference rules for MALL are presented in Figure 4.2.

\[
\begin{align*}
\vdash A, A⊥ \\
\vdash \Gamma, A \vdash \Delta, B \quad \vdash \Gamma, A \vdash \Delta, B \\
\vdash \Gamma, A \vdash \Delta, B & \quad \vdash \Gamma, A \vdash \Delta, B \\
\vdash \Gamma, A, B & \quad \vdash \Gamma, A \vdash \Delta, B
\end{align*}
\]

\[
\begin{align*}
\vdash \Gamma, A & \vdash \Delta, B \\
\vdash \Gamma, A & \vdash \Delta, B \\
\vdash \Gamma, A & \vdash \Delta, B
\end{align*}
\]

\[
\begin{align*}
\vdash \Gamma, A & \vdash \Delta, B \\
\vdash \Gamma, A & \vdash \Delta, B \\
\vdash \Gamma, A & \vdash \Delta, B
\end{align*}
\]

\[
\begin{align*}
\vdash \Gamma, A & \vdash \Delta, B \\
\vdash \Gamma, A & \vdash \Delta, B \\
\vdash \Gamma, A & \vdash \Delta, B
\end{align*}
\]

As hinted earlier, our logic-based process specifications use a restricted set of acceptable CLL terms in order to make process specifications simpler and more intuitive and to facilitate proof automation. In particular, we only allow inputs that are atomic (e.g. \( A⊥ \)) or optional (e.g. \( A⊥ & B⊥ \)) and only a single output (in the style of LLP) that can be a disjunction (\( ⊕ \)) of conjunctions (\( ⊗ \)). We discuss process specifications and their restrictions in more detail in Section 6.2.
Chapter 4. The Proofs-as-processes Paradigm

4.3 Proof Terms and Inference Rules

Bellin and Scott attach free variables as proof terms to every CLL formula. Each of these variables corresponds to a \( \pi \)-calculus communication port/channel. Therefore, the CLL formula \( x : A \), consisting of linear proposition \( A \) and proof term \( x \), represents a \( \pi \)-calculus port \( x \) that carries information of type \( A \). Moreover, a \( \pi \)-calculus process is attached to every CLL judgement, so that the judgement \( \vdash \text{ExampleProc} :: x : A \perp, \ y : B \) represents a process named ExampleProc with an input of type \( A \) through channel \( x \) and an output of type \( B \) through channel \( y \) (see Section 6.2 for more information on process specifications using CLL).

In the proofs-as-processes paradigm, \( \pi \)-calculus terms are attached to the premises and conclusion of each CLL inference rule. The process attached to the conclusion of each inference rule is dependent on the processes attached to each of the premises of the rule and the proof terms of the involved formulas. For example, let us consider the tensor rule for CLL including the proof term annotations as shown below:

\[
\begin{align*}
\vdash F :: \vec{w} : \Gamma, \ x : A & \quad \vdash G :: \vec{u} : \Delta, \ y : B \\
\vdash (F, G)_{\vec{w} \vec{u} z} :: \vec{w} : \Gamma, \ \vec{u} : \Delta, \ z : A \otimes B
\end{align*}
\]

(4.1)

Processes \( F \) and \( G \) are attached to the two premises of the rule based on any previous proof steps. The process calculus term attached to the conclusion of the rule, also referred to as the translation of the rule to the \( \pi \)-calculus, is given by the following term:

\[
\begin{align*}
\bigotimes_{z} (F, G)_{\vec{w} \vec{u} z} \equiv (\nu x, y) (\tau(x, y) \cdot (F_{\vec{w} x} \mid\mid G_{\vec{u} y}))
\end{align*}
\]

(4.2)

We note that term (4.2) is dependent on processes \( F \) and \( G \) and also on the proof terms \( x, y, \vec{w} \) and \( \vec{u} \) of the involved formulas. The free variables found in the annotations of the formulas of a conclusion of an inference rule are ensured to be exactly the same as the free names of the corresponding \( \pi \)-calculus term for that rule, in this case \( z, \vec{w} \) and \( \vec{u} \).

The seven basic CLL inference rules with their process correspondences are shown in Figure 4.3. Before proceeding to analyse and attempt to give a practical, intuitive explanation of these correspondences, we remind the reader that the symmetry of CLL
allows for two equivalent π-calculus translations for the identity axiom and the various operators. We have chosen to translate positive atoms and the ⊗ and ⊕ operators as senders while we translate negative atoms and the □ and & operators as receivers. We examine these rules from Figure 4.3 and their translations more closely next:

### CLL inference rule

\[ \vdash \text{I}_{x,y} : y : A, x : A \perp \]

\[ \vdash F :: \overline{w} : \Gamma, x : A \vdash G :: \overline{u} : \Delta, y : B \]

\[ \vdash \overline{x,y} (F,G) _{\overline{w} \overline{u}} :: \overline{w} : \Gamma, \overline{u} : \Delta, z : A \otimes B \]

\[ \vdash \pi \text{-calculus translation} \]

\[ \pi_{x,y} \equiv (v x, y) \langle z(y) \cdot (F_{\overline{w}x} \mid G_{\overline{y}y}) \rangle \]

\[ \vdash F :: \overline{w} : \Gamma, x : A \]

\[ \vdash \boxtimes \text{L}(P) _{\overline{w}z} :: \overline{w} : \Gamma, z : A \oplus B \]

\[ \vdash \boxtimes \text{R}(Q) _{\overline{w}z} :: \overline{w} : \Gamma, z : A \oplus B \]

\[ \vdash P :: \overline{w} : \Gamma, x : A \]

\[ \vdash Q :: \overline{w} : \Gamma, y : B \]

\[ \vdash \& \text{L}(P,Q) _{\overline{w}z} :: \overline{w} : \Gamma, z : A \& B \]

\[ \vdash \& \text{R}(Q) _{\overline{w}z} :: \overline{w} : \Gamma, z : A \& B \]

\[ \vdash F :: \overline{u} : \Gamma, x : C \]

\[ \vdash \text{G} :: \overline{u} : \Delta, y : C \perp \]

\[ \vdash \text{Cut} ^{\pm} (F,G) _{\overline{u} \overline{v}} :: \overline{u} : \Gamma, \overline{v} : \Delta \]

\[ \text{Cut} ^{\pm} (F,G) _{\overline{u} \overline{v}} :: \overline{w} : \Gamma, z : A \oplus B \]

\[ \text{Cut} ^{\pm} (F,G) _{\overline{u} \overline{v}} :: \overline{w} : \Gamma, z : A \oplus B \]

\[ \vdash \pi \text{-calculus translation} \]

\[ \pi_{x,y} \equiv (v x, y) \langle z(y) \cdot (F_{\overline{w}x} \mid G_{\overline{y}y}) \rangle \]

Figure 4.3: The CLL inference rules annotated with process calculus proof terms and the corresponding π-calculus processes.

- **The identity axiom**: The identity axiom \( \vdash y : A, x : A \perp \) can be intuitively translated given the aforementioned choices to \( x(a) \cdot y(a) \cdot 0 \). The resulting process receives a message \( a \) through the channel \( x \) of the negative literal and sends the same message \( a \) through the channel of the positive atom \( y \). Such a process is referred to as an axiom
buffer.

- **The \( \otimes \) rule:** The tensor rule must intuitively correspond to a channel \( z \) that sends two messages \( x \) and \( y \) corresponding to the literals \( A \) and \( B \) (that are involved in \( F \) and \( G \) respectively) simultaneously. The given translation satisfies our intuition. It sends both \( x \) and \( y \) through channel \( z \) followed by the parallel execution of \( F \) and \( G \).

- **The \( \odot \) rule:** The par rule is the dual of the tensor rule. It is, therefore, expected that it should be able to receive two messages \( x \) and \( y \) simultaneously through a common channel \( z \). The corresponding \( \pi \)-calculus process does exactly that before invoking \( F \).

- **The \( \oplus \) rules:** The \( \oplus \) operator provides the means to ignore an argument or, consequently, a channel. In the first rule, for example, we expect to receive two names \( u \) and \( v \) corresponding to the channels for \( A \) and \( B \) respectively through a common channel \( z \). The process ignores the second name \( v \) and uses the first one \( u \) to send \( x \) before invoking \( P \). The process for the second rule is symmetric as it ignores the first name \( u \) and sends \( y \) through \( v \).

- **The \& rule:** The with rule is the dual of the \( \oplus \) rules. Therefore, the corresponding process sends two channels \( u \) and \( v \) through \( z \) and invokes either \( P \) or \( Q \) depending on which of the two channels receives input.

- **The Cut rule:** The Cut rule is perhaps the most significant one as far as process interactions are concerned. We already discussed that applying cut-elimination to the proof corresponds to performing reductions in its \( \pi \)-calculus translation. Therefore, the Cut rule corresponds to a reduction/interaction between processes \( F \) and \( G \). Additionally, the interaction will take place through the ports corresponding to the literal being cut, namely \( C \). Thus, port \( x \) will be connected to port \( y \) to form a common channel \( z \). The two processes are expected to interact through this common channel \( z \) when invoked in parallel. It is worth noting that there are no assumptions made whatsoever about which of the two services will be the receiver and which will be the sender.

As previously mentioned in Section [4.2](#), the inference rules for the four CLL units are not used in the proofs-as-processes paradigm.

It is worth noting that if the process calculus annotations are hidden from the rules as presented in Figure [4.3](#), we obtain the original CLL inference rules from Figure [4.2](#). More generally, hiding the proof terms and process translations from a CLL proof does not affect the structure and validity of the proof. We exploit this in subsequent
examples so as to present cleaner, more readable CLL proofs by freely hiding process annotations when the example focuses on the logical part of a particular proof.

### 4.4 Formalising the Proofs-as-processes Paradigm

In this section, we describe our efforts towards the formalisation of the proofs-as-processes paradigm. Embedding CLL in higher-order logic is a core requirement for the formalisation of the proofs-as-processes paradigm. There are two distinct types of such embedding (Boulton et al., 1993): in a shallow embedding the language constructs are written as higher-order logic predicates whereas in a deep embedding they are defined as a new distinct datatype. In our case, a deep embedding has several advantages, the most important of which are the following:

- It allows an easier enforcement of linearity (a shallow embedding would require explicit accounting of linear propositions and a reasoning overhead in the proofs).
- There can be a direct mapping of the sequent calculus based logic used by Bellin and Scott.
- The attachment and handling of proof terms within the syntax of the logic itself is made easier (see Sections 4.4.3 and 4.4.5.2).
- It provides capabilities for meta-level reasoning about the language itself.

Naturally, the extra flexibility of a deep embedding leads to a requirement of significantly more effort to handle deeply embedded judgements and derive proofs.

The embedding of CLL within HOL Light is based on similar work done for Coq by Power et al. (Power et al., 1999) and Sadrzadeh (Sadrzadeh, 2003). The main differences include the process calculus annotations (see 4.4.3), the transition from a two-sided sequent calculus to a one-sided version, and the use of multisets instead of lists to represent judgements (see Section 4.4.2).

In Section 4.4.1 we give an overview of the embedded syntax in HOL Light. Then, we describe two particular design choices in our embedding, namely the use of multisets to represent judgements and the embedding of proof terms as part of the logic itself in Sections 4.4.2 and 4.4.3 respectively. The formalisation of the inference rules for the
proofs-as-processes paradigm is analysed in Section 4.4.4 whereas in Section 4.4.5 we give some implementation details on our inference and proof engine for the embedded logic.

### 4.4.1 Syntax

Our formal definition for the syntax of CLL terms in HOL Light is given in Figure 4.4.

```plaintext
let linprop_INDUCT, linprop_RECURSION = define_type
   "LinProp = LinOne
    | LinBottom
    | LinPlus LinProp LinProp
    | LinWith LinProp LinProp
    | LinTimes LinProp LinProp
    | LinPar LinProp LinProp"
```

Figure 4.4: The definition of the syntax for CLL terms in HOL Light.

In Figure 4.4 “LinOne” and “LinBottom” correspond to the CLL constants 1 and \( \bot \) (see Section 4.5.1.1). Even though we do not make use of units in our logic, it is necessary to introduce 1 as the bottom object of the definition of “LinProp” as a recursive type in HOL Light. We also need to introduce \( \bot \) as the negation of 1 (see Figure 4.6). The rest of the connectives correspond to the standard CLL connectives that match their respective names as shown in Figure 4.5. Additionally, we introduce syntactic abbreviations for each of the connectives in HOL Light, also shown in Figure 4.5. We note that, for readability, from here on we will be using the standard CLL syntax as opposed to the ASCII based HOL Light syntax, even for the presentation of HOL Light statements.

Linear negation \( \bot \) is defined as a function “NEG” based on four of the equations from Figure 4.1 and their duals, as shown in Figure 4.6. We have also included the equations for the two units 1 and \( \bot \) so that “NEG” is a total function. Based on this definition, we proved the fifth equation, namely the fact that \( (A^\bot)^\bot \equiv A \). This approach made the formalisation more straightforward in HOL Light while preserving the properties of the original five equations.
4.4. Formalising the Proofs-as-processes Paradigm

<table>
<thead>
<tr>
<th>CLL connective</th>
<th>HOL Light definition</th>
<th>HOL Light syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\otimes$</td>
<td>LinTimes</td>
<td>**</td>
</tr>
<tr>
<td>$\forall$</td>
<td>LinPar</td>
<td>%</td>
</tr>
<tr>
<td>$\oplus$</td>
<td>LinPlus</td>
<td>++</td>
</tr>
<tr>
<td>$&amp;$</td>
<td>LinWith</td>
<td>&amp;</td>
</tr>
</tbody>
</table>

Figure 4.5: HOL Light definitions and abbreviations for each CLL connective.

let NEG_CLAUSES = define

(NEG 1 = $\bot$) \land
(NEG $\bot$ = 1) \land
(NEG (A $\otimes$ B) = (NEG A) $\forall$ (NEG B)) \land
(NEG (A $\forall$ B) = (NEG A) $\otimes$ (NEG B)) \land
(NEG (A $\oplus$ B) = (NEG A) & (NEG B)) \land
(NEG (A & B) = (NEG A) $\oplus$ (NEG B))

Figure 4.6: The definition of the linear negation in HOL Light.
As mentioned previously, the equations from Figure 4.1 provide a syntactic equivalence between linear formulas. Conversion of a linear formula to another syntactically equivalent one based on the given equations is accomplished in HOL Light using rewriting. In the proofs presented throughout this work rewriting is applied freely and implicitly as we take such syntactic equivalences for granted.

4.4.2 Using Multisets in Sequent Calculus Judgements

Since we aim to be faithful to the work of Bellin and Scott, we use a one-sided sequent calculus. Usually (for example in Sadrzadeh, 2003), embeddings that use sequent calculus use a list or vector of terms for either side of the turnstile ($\vdash$). Given that the order of the terms is not important, formal representations include additional inference rules that allow swapping of two terms in either side. These rules are usually referred to as the “Exchange” rules. The Exchange rule for the one-sided sequent calculus is the following:

$$\vdash \Gamma, B, A, \Delta \quad Exchange \quad \vdash \Gamma, A, B, \Delta$$

Given an existing theory of multisets in HOL Light, we decided to use multisets of terms in our formalisation. This implicitly allows exchanges in the order of terms without the need for the Exchange rule. The matching process is more complicated, but we managed to build multiset matching functions that minimize this issue (see Section 4.4.5.3).

The multiset theory in HOL Light already has many useful formalised properties. The few more needed ones were proven within a small multiset library extension. A variety of methods and tools needed in our particular tasks were also implemented. Finally, we introduced abbreviations for multiset sums and singleton multisets, in order to obtain a cleaner syntax. In this thesis, for readability, we use the $\uplus$ symbol to denote a multiset sum, whereas enumerated multisets are enclosed in curly brackets $\{\cdot\}$.

4.4.3 Sequent Calculus Embedding with Proof Terms in HOL Light

In general, HOL Light does not provide any mechanisms to allow proof annotations with proof terms or program synthesis, meaning that we had to construct such mech-
4.4. Formalising the Proofs-as-processes Paradigm

There are alternative proof assistants, such as Coq (Bertot et al., 2004), that provide such functionality for connecting proofs to computational components. However, using these would require adjusting their mechanisms to be specifically used with CLL and π-calculus which in turn would require significant effort. Implementing this functionality ourselves allows us to have full control over it and freely define all its aspects.

Given the tools available in HOL Light, the best option is to implement proof terms as part of the logic we are defining on a deep embedding rather than as external annotations. To be more specific, our defined sequent calculus utilises annotated CLL terms (as opposed to pure CLL terms) as components of sequents. The definition of these terms is the following:

\[
\text{define_type } "\text{LinTerm = LL LinProp A"} \]

Essentially, this definition annotates a CLL term of type \text{LinProp} with a π-calculus name of type \text{A}. The “LL” constructor corresponds to the type operator “:”, and is abbreviated in HOL Light as the infix syntactic operator “<>” (to avoid clashing with the HOL Light typing operator “:”). As a result, the HOL Light term \text{A<>b} corresponds to a CLL term \text{A} annotated with term \text{b}, i.e. \text{b:A}. Note that the inverted HOL Light syntax is only chosen for technical reasons.

The CLL inference rules are defined using linear consequence “|-” as a function of the following type:

\[
:\text{((num)LinTerm)multiset→ (num)Agent→ bool} \]

The (num)Agent parameter corresponds to the π-calculus translation for the given CLL judgement. The translation is, therefore, included with the judgement in a single logical term. For example, the HOL Light term \text{|- A<>a P} corresponds to the judgement \text{\vdash P::a:A}. As for the CLL syntax, for ease of readability, from here on we will use the standard sequent calculus and type theory syntax even when presenting HOL Light statements.

4.4.4 Proofs-as-processes Inference Rules

Having formally defined the syntax for CLL terms with proof terms in HOL Light, we proceed to define the CLL inference rules with their proofs-as-processes translations.
The HOL Light definition of these rules is shown in Figure 4.7.

As an example, let us review the definition of the tensor rule which, based on (4.1) and (4.2), is the following:

$$\vdash F :: \vec{w} : \Gamma, x : A \vdash G :: \vec{u} : \Delta, y : B \vdash (\nu x, y) (\pi(x, y) . (F || G))(\nu u, z : A \otimes B \otimes)$$

(4.3)

This rule is embedded in HOL Light as follows:

$$\forall x y z F G A B \Gamma \Delta. \vdash \Gamma \psi\{x : A\} (F) \land \vdash \Delta \psi\{y : B\} (G) \rightarrow \vdash \Gamma \psi\Delta \psi\{z : A \otimes B\} ((\nu x, y) (\pi(x, y) . (F || G)))$$

(4.4)

Each rule is represented as an implication in the meta-logic, which in this case is higher-order logic. We can, therefore, consider this as a shallow embedding of sequent calculus derivations. Having an extra level of depth, where derivations are described using a separate, explicit definition, would allow us to perform interesting meta-level reasoning tasks such as formalising the soundness and completeness proofs of Bellin and Scott. Expressing the sequent calculus derivations as higher-order logic implications provides a more pragmatic approach that allows easier proof construction and inference at the level of the embedded logic (see Section 4.4.5.1).

Note that all the variables are universally quantified in the meta-logic. This, combined with the fact that there is no distinction between atomic linear propositions and terms in the syntax of the rules, has an unwanted effect. Namely, the identity rule $\vdash A, A \bot$ as given by Bellin and Scott (see Figure 4.3) assumes $A$ is an atomic proposition. This is particularly important in the context of proofs-as-processes, since the translation of the rule $I_{x,y} \equiv x(a) . \bar{y}(a) . 0$ in the $\pi$-calculus corresponds to an axiom buffer that carries a single (atomic) message $a$. However, this is not reflected in the formalised rule $\forall a A x y. \vdash \{y : A \bot\} \psi\{x : A\} (x(a) . \bar{y}(a) . 0)$ where $A$ can match any CLL term. There is no straightforward way to enforce this restriction during matching in the given embedding. However, any judgement of the form $\vdash A, A \bot$ where $A$ is not atomic, more generally referred to as a buffer, can be broken down to (atomic) axiom buffers in a straightforward, decidable way (see Section 7.3). Ensuring this breakdown is always performed on non-atomic buffers before applying the identity rule in our CLL proofs should be sufficient in order to avoid any side effects.
let cll_RULES, cll_INDUCT, cll_CASES \equiv new_inductive_definition

\forall a A x y.
\vdash \{ y: A^\perp \} \uplus \{ x: A \} (y(a), \pi(a), 0)
∧ \forall x y z F G A B \Gamma \Delta.
\vdash \Gamma \uplus \{ x: A \} (F) \land \vdash \Delta \uplus \{ y: B \} (G) \rightarrow
\vdash \Gamma \uplus \Delta \uplus \{ z: A \otimes B \} ((v x, y) (\pi(x, y), (F \parallel G)))
∧ \forall x y z F G A B \Gamma.
\vdash \Gamma \uplus \{ x: A \} \uplus \{ y: B \} (F) \rightarrow
\vdash \Gamma \uplus \{ z: A \otimes B \} (z(x, y).F)
∧ \forall x z u v P A B \Gamma.
\vdash \Gamma \uplus \{ x: A \} (P) \rightarrow
\vdash \Gamma \uplus \{ z: A \otimes B \} ((v x) (z(u, v), \pi(x).P))
∧ \forall y z u v Q A B \Gamma.
\vdash \Gamma \uplus \{ y: B \} (Q) \rightarrow
\vdash \Gamma \uplus \{ z: A \otimes B \} ((v y) (z(u, v), \pi(y).Q))
∧ \forall x y z u v P Q A B \Gamma.
\vdash \Gamma \uplus \{ x: A \} (P) \land \vdash \Gamma \uplus \{ y: B \} (Q) \rightarrow
\vdash \Gamma \uplus \{ z: A \land B \} ((v u, v) (\pi(u, v). (u(x) . P + v(y) . Q)))
∧ \forall x y z F G \Gamma C \Delta.
\vdash \{ y: C^\perp \} \uplus \Delta (G) \land \vdash \Gamma \uplus \{ x: C \} (F) \rightarrow
\vdash \Gamma \uplus \Delta ((v z) (F [z/x] \parallel G[z/y]))\:\).

Figure 4.7: The definition of linear consequence in HOL Light. This introduces the CLL inference rules combined with their π-calculus translations.
Finally, we should also make note of variables $\Gamma$ and $\Delta$. These correspond to contexts $\Gamma$ and $\Delta$ in the original rule in (4.3). As such, they are multisets of linear terms (which are annotated with numerical channels) and have type “:\((\text{num})\text{LinTerm})\text{multiset}”. Note that there is no separate denotation for the proof terms within these contexts (represented as $\vec{w}$ and $\vec{u}$ in (4.3)). When the rule is being used in a proof, $\Gamma$ and $\Delta$ must be matched to all the terms of the involved judgements except those containing $A$ and $B$. The non-trivial matching algorithm is described in Section 4.4.5.3.

4.4.5 Inference in the Embedded Logic

Having obtained an embedding of the CLL inference rules in HOL Light, the next step is using them to perform inference tasks by constructing sequent calculus derivations. As an example, let us consider the commutative law of the tensor operator $\otimes$ “TimesComm”:

$$\vdash A, A \bot, B, B \bot \vdash (A \otimes B)^{\bot}, B \otimes A \quad \text{TimesComm}$$

(4.5)

The proof tree of $\text{TimesComm}$ is as follows:

$$\vdash P_b :: bb : B, nb : B^{\bot} \quad Ax \quad \vdash P_a :: aa : A, na : A^{\bot} \quad Ax \quad \vdash \otimes_y \ (P_b, P_a) :: na : A^{\bot}, nb : B^{\bot}, y : B \otimes A \quad \gamma_y$$

$$\vdash \nu_{bb,aa} \ (\otimes_y \ (P_b, P_a)) :: x : A^{\bot}, y : B \otimes A \quad \gamma_x$$

(4.6)

When the $\pi$-calculus definitions are expanded, this proof tree corresponds to the following:

$$\vdash P_b :: bb : B, nb : B^{\bot} \quad Ax \quad \vdash P_a :: aa : A, na : A^{\bot} \quad Ax \quad \vdash (\nu \ bb,aa) \ (\gamma(bb,aa) \cdot (P_b \parallel P_a)) :: na : A^{\bot}, nb : B^{\bot}, y : B \otimes A \quad \gamma_y$$

$$\vdash x(na, nb) \cdot (\nu \ bb,aa) \ (\gamma(bb,aa) \cdot (P_b \parallel P_a)) :: x : A^{\bot}, y : B \otimes A \quad \gamma_y$$

(4.7)

Note that we do not use the identity axiom to eliminate the two assumptions, because $A$ and $B$ may not be atomic (see Section 7.3 for more details).
In the following sections we give implementation details for several aspects of our inference and proof engines. More specifically, our meta-level natural deduction framework is described in Section 4.4.5.1, whereas the machinery that handles process calculus proof annotations is analysed in Section 4.4.5.2. Details of our multiset matching algorithm can be found in Section 4.4.5.3 followed by a discussion on the usage of metavariables in Section 4.4.5.4

4.4.5.1 Meta-level Natural Deduction

Since the sequent calculus is deeply embedded in HOL Light, our main approach to constructing CLL derivations is by applying the embedded rules using higher-order-logic natural deduction. Essentially, we construct sequent calculus CLL proofs at the object level by plugging rules together using natural deduction at the meta-level.

Traditional use of HOL systems, such as HOL4 and HOL Light, dictates the use of tactics for backwards reasoning and so called rules for forward reasoning (Harrison, 1996a). Tactics are applied on a goal and produce a (possibly empty) set of new sub-goals, whereas rules are functions that combine one or more theorems or assumptions to derive new facts. One of the main problems with this approach, particularly in the context of a deeply embedded logic, is that every inference rule needs to be expressed both as a new tactic that applies the rule backwards and as a new rule/function that applies the rule forwards. In our example, this would require 7 new tactics and 7 new rules/functions for the primitive inference rules of CLL, plus a new tactic and a new function for each derived rule (see Section 7.2 for some derived rules and properties that have been useful).

Instead we implemented and used an extended version of our Isabelle Light framework that enhances HOL Light with mechanisms for natural deduction style proofs (Papapanagiotou and Fleuriot, 2010). Isabelle Light is a set of tactics and tools in HOL Light that emulate the natural deduction proof style of Isabelle (Paulson, 1994). Its original aim was to enhance the user-level interaction in HOL Light by facilitating natural deduction style proofs.

In particular, Isabelle Light emulates Isabelle’s four natural deduction tactics, rule, erule, drule and frule. These enable the usage of any arbitrary HOL Light theorem in a natural deduction proof, either as a forward reasoning step (manipulating assumptions) or as a backwards reasoning step (breaking down the goal). We briefly describe
the application of each of the four tactics with any given rule $P_1 \land P_2 \land \ldots \land P_n \Longrightarrow Q$:

- The **rule** tactic unifies the conclusion of the rule $Q$ with the current subgoal. Each premise $P_i$ is then instantiated and added as a new subgoal for a total of $n$ new subgoals. The **rule** tactic is appropriate for backward reasoning.

- The **erule** tactic behaves similarly to **rule**, but also unifies $P_1$ (*major premise*) with one of the assumptions in the current subgoal. The matching assumption is deleted (eliminated) from the current subgoal and $n-1$ new subgoals are introduced. The **erule** tactic is appropriate for simultaneous forward and backward reasoning.

- The **drule** tactic unifies the major premise $P_1$ with one of the assumptions of the current subgoal which is then deleted (destroyed) and $n-1$ new subgoals are introduced. The current subgoal with $Q$ added as an assumption forms the $n$th subgoal. The **drule** tactic is appropriate for forward reasoning.

- The **frule** tactic is identical to the **drule** tactic, but does not delete the matching assumption, allowing for it to be reused in the proof if need be.

The four tactics are extended into four alternatives: **rule_tac**, **erule_tac**, **drule_tac**, and **frule_tac**. These can be used to partially instantiate a rule before applying it with the corresponding tactic. This is often useful, for example, in order to resolve ambiguities when a rule can be matched to a particular proof state in more than one way.

In addition to the new tactics, Isabelle Light includes mechanisms for metavariable manipulation and matching. It is fully integrated within HOL Light and compatible with the other HOL Light tactics and tacticals, without introducing a new interface or a layered interface.

A variety of extensions and alterations were required to suit our particular needs for the embedded language, including construction of computational components, multiset matching, and advanced use of metavariables. These are discussed in the following sections. The tactics supporting the extra functionality are marked with an “ll” prefix, indicating their specialisation for linear logic.

The extended Isabelle Light library formed a good basis for our meta-level natural deduction framework. The natural deduction tactics greatly facilitate the direct usage of embedded inference rules such as the ones for CLL. For example, consider the proof
(4.6) of TimesComm without process calculus annotations:

\[
\begin{align*}
\vdash B, B & \quad \text{Ax} \quad \vdash A, A & \quad \text{Ax} \\
\vdash A, B & \quad \text{⊥} & \vdash B & \quad \text{⊗} \\
\vdash (A \otimes B) & \quad \text{⊥} & \vdash B & \quad \text{⊗} \\
\end{align*}
\]

This proof can be accomplished in our framework either backward or forward or even by working in two directions simultaneously. Slightly simplified versions of the HOL Light proof scripts for a fully backward and a fully forward proof are shown in Figure 4.8. It is worth noting that the order in which the rules are being applied in the proof scripts follows the order of the rules in proof (4.8), from bottom to top for the backward proof and from top to bottom for the forward proof.

Figure 4.8: The interactive HOL Light scripts for the proof of TimesComm (4.6). Rule instantiations are omitted (using "...") for clarity.

Note that NEG_TIMES is the equation of linear negation that involves ⊗ (see Figure 4.1). ll_par and ll_times correspond to the CLL inference rules for ⊗ and ⊗ respectively, and by.llmeta_assumption is a tactic that uses multiset matching and metavariables to match the current (sub)goal to an assumption (see Section 4.4.5.4).

The order of the application of the rules is immediately apparent in each of the two cases. Moreover, the tactics lldrule and llrule are able to use both the ⊗ and ⊗ rules, as well as any other primitive or derived CLL inference rule, forwards and backwards respectively.

### 4.4.5.2 Handling Computational Components

The representation of proof terms as part of the embedded logic (see Section 4.4.3) raises issues regarding the handling of the computational components during the proof.
Based on whether or not the $\pi$-calculus translation is known in advance, this can be split in two problems: (i) construction of the $\pi$-calculus translation of a proof of a given goal and (ii) verification that a given $\pi$-calculus term corresponds to the translation of a particular proof.

In a usual proof term environment such as the one supported by Coq, the computational component is being handled internally while the user only has a view of the logical proof. This helps maintain a clean, logical interface, separated from any computational annotations. In our case, we initially focus on the functionality rather than the logical interface. Ultimately, we aim for the proofs to be performed via a graphical tool (see Section 6.4), thus the cleanliness of the underlying interface can be viewed as secondary.

The most straightforward way to accomplish computational construction in HOL Light proofs is by using metavariables. These are variables in a given goal whose instantiation is delayed until the proof is finished (see Section 4.4.5.4 for more details). When performing a proof within our embedded logic, the initial process calculus component is set as a metavariable. Moreover, process calculus variables in the inference rules are also considered metavariables. As the proof progresses, we construct metavariable instantiations that match the $\pi$-calculus translations of the rules being applied. At the end of the proof, applying the resulting instantiations to the initial metavariable gives us the $\pi$-calculus term resulting from that proof.

Let us attempt to construct part of proof (4.6) of TimesComm backwards. Assume we have the following subgoal:

$$\vdash P :: a : A \perp, b : B \perp, y : B \otimes A$$

We consider that the $\pi$-calculus term $P$ that corresponds to the proof of this subgoal is not yet known, and we aim to construct it during the proof. Note that, in our goal, $P, a,$ and $b$ are HOL Light metavariables. Essentially, we are trying to construct the appropriate instantiation for these variables while performing this proof backward. The first step involves the usage of the tensor rule:

$$\vdash F :: \Gamma, x : A \quad \vdash G :: \Delta, y : B \otimes$$

$$\frac{\times y}{\quad \otimes (F, G) :: \Gamma, \Delta, z : A \otimes B}$$
In order to apply this rule we need to instantiate the metavariable $P$ to $\otimes_x (\mathcal{F}, \mathcal{G})$ whereas $y$ in the goal should match $z$ in the rule. Note that the CLL terms (such as $\Gamma, \Delta, \mathcal{A}$, and $\mathcal{B}$) will be matched using our multiset matching algorithm (see Section 4.4.5.3). We should also note that our algorithm ensures all the variables in the rule being applied have fresh names in order to avoid any clashes (see Section 4.4.5.4). Variables $\mathcal{F}$, $\mathcal{G}$, $\chi$, and $y$ are introduced as fresh metavariables in the new subgoals.

The constructed proof tree is the following:

$$
\begin{align*}
\vdash \mathcal{F} :: \chi : B, \ b : B^\perp \quad &\vdash \mathcal{G} :: y : A, \ a : A^\perp \\
\otimes_x (\mathcal{F}, \mathcal{G}) :: a : A^\perp, \ b : B^\perp, \ y : B \otimes A
\end{align*}
$$

In the next couple of steps we use the two available assumptions for the two proof branches. More specifically, we use $\vdash P_b :: bb : B, \ nb : B^\perp$ for the left-hand branch and $\vdash P_a :: aa : A, \ na : A^\perp$ for the right-hand branch. Matching the left-hand branch leads to the instantiation of $\mathcal{F}$ to $P_b$, $\chi$ to $bb$, and $b$ to $nb$. Similarly, in the right-hand branch we obtain the instantiation of $\mathcal{G}$ to $P_a$, $y$ to $aa$, and $a$ to $na$. The resulting, completed proof tree is the following:

$$
\begin{align*}
\vdash P_b :: bb : B, \ nb : B^\perp \quad &\vdash P_a :: aa : A, \ na : A^\perp \\
\otimes_y (P_b, P_a) :: na : A^\perp, \ nb : B^\perp, \ y : B \otimes A
\end{align*}
$$

In the interactive theorem proving setting of HOL Light, storing all the intermediate instantiations from every proof step allows us to easily obtain the computational component of the entire proof by instantiating $P$:

$$
P = \bigotimes_y (P_b, P_a) = (\forall bb, aa). (bb, aa) \cdot (P_b || P_a)
$$

We have, therefore, managed to construct the $\pi$-calculus translation for our goal by delayed instantiation of both the initial metavariables $P$, $a$, and $b$ and the metavariables introduced by each rule application such as $\mathcal{F}$ and $\mathcal{G}$ as the proof progresses.

It should be noted that any metavariables that are not instantiated by the end of the proof are left as free variables in the resulting lemma.
For their part, computational verification proofs do not require metavariables. In those, the computational translation is already known. We note that there may exist valid alternative proofs of the same logical statement which are not be acceptable in this case because they generate a different (possibly congruent) process term.

So, in contrast to the construction example, the goal in this case would be:

\[
\vdash \bigotimes_y (P_b, P_a) :: na : A \perp, \; nb : B \perp, \; y : B \otimes A
\]

The computational translation of the tensor rule \( \bigotimes (\mathcal{F}, \mathcal{G}) \) should exactly match the process of the current goal \( \bigotimes_y (P_b, P_a) \), without introducing new metavariables, i.e. \( x, y, z, \mathcal{F}, \mathcal{G} \) are all matched to an expression in the goal.

The important thing to note here is the difference in the way processes are matched in the two types of proofs. In a construction proof, the process term in the goal (e.g. \( P \) in the above example) is matched to the process translation of the rule. In a verification proof, the process translation of the rule is matched to the given process term in the goal. In order to allow our tactics to accommodate both types of proofs, as well as take advantage of the fact that metavariables can be instantiated to anything, unification is performed between the computational components of the rule and the goal. HOL Light supports first order term unification without type instantiations. This is sufficient for our needs here, since no higher order variables are used and the types are predetermined.

### 4.4.5.3 Multiset Matching

In Section 4.4.2, we discussed the use of multisets to represent CLL judgements so as to avoid the Exchange rule. When using CLL inference rules in our meta-level natural deduction system, apart from the unification of the computational component described in the previous section, we also need to match the multisets describing the involved CLL judgements appropriately, with minimum effort from the user.
Let us return to our TimesComm example, with proof annotations hidden as in (4.8):

\[ \begin{align*}
\vdash B, B \bot & \quad \text{Ax} \\
\vdash A \bot, B \bot, B \otimes A & \quad \otimes \\
\vdash (A \otimes B) \bot, B \otimes A & \quad \gamma
\end{align*} \]

In order to accomplish this proof, we need the tensor (\(\otimes\)) rule:

\[ \vdash \Gamma, A \vdash \Delta, B \]

\[ \vdash \Gamma, \Delta, A \otimes B \otimes \]

In the case of the backward proof (see Figure 4.8a), the conclusion \(\vdash \Gamma, \Delta, A \otimes B\) of the rule must match the current goal \(\vdash A \bot, B \bot, B \otimes A\). Therefore, the tactic that applies this rule must match the multiset that corresponds to the conclusion of the rule:

\[ \Gamma \uplus \Delta \uplus \{A \otimes B\} \]

to the one that corresponds to the goal:

\[ \{A \bot\} \uplus \{B \bot\} \uplus \{B \otimes A\} \]

and more specifically match \(A\) to \(B\), \(B\) to \(A\), \(\Gamma\) to \(\{B \bot\}\), and \(\Delta\) to \(\{A \bot\}\).

The simple Isabelle Light type of matching would attempt to match the multisets based on the specific ordering, i.e. \(\Gamma\) would be matched to \(\{A \bot\}\). If we force \(\Gamma\) to be instantiated to \(\{B \bot\}\) beforehand (using \texttt{rule_tac}), the match would fail because \(\{B \bot\} \uplus \Delta \uplus \{A \otimes B\}\) does not match \(\{A \bot\} \uplus \{B \bot\} \uplus \{B \otimes A\}\) unless we incorporate commutativity and associativity of multiset sum \(\uplus\) in the matching algorithm.

Also note that, if the conclusion was ordered differently, for example \(\{B \otimes A\} \uplus \{A \bot\} \uplus \{B \bot\}\), then the algorithm would try to match \(\{A \otimes B\} \uplus \{B \bot\}\) to \(\{B \bot\}\) and fail.

Similar situations arise in forwards proofs (as in Figure 4.8b), where assumptions of the rule are matched to the current assumptions of the goal state.

We therefore implement an algorithm that properly matches CLL judgements by taking into consideration the properties of multiset sum. The algorithm splits the multisets into their elements, which are either singleton multisets or multiset variables. Then it performs the following matches:

1. First it tries to match elements of the multiset taken from the inference rule that do not contain free variables, i.e. constants and terms pre-instantiated by the user.
2. Then it tries to match elements that are not variable multisets. In our example, \{A \otimes B\} is a non-variable element that will be matched first.

3. Multiset variables (such as \(\Gamma\) and \(\Delta\) in our example) are left for last because they can match anything. If the target does not have enough elements for all such multiset variables, they are matched to the empty multiset, whereas if there are more elements left in the goal than the available variable sub-multisets, they are combined into a single multiset. For example, if our goal was \(\vdash a, b \otimes c\), then \(\Gamma\) in the tensor rule would be matched to \(\{a\}\) and \(\Delta\) to the empty multiset. If our goal was \(\vdash a, b \otimes c, d, e\), then \(\Gamma\) would be matched to \(\{a\}\) and \(\Delta\) to \(\{d, e\}\).

Rewriting using the commutative and associative properties of multiset sum, also referred to as \textit{normalization}, is used to justify these multiset matches within the tactics. More specifically, each of the three matching steps above produces an instantiation of the inference rule. Applying these instantiations and then normalizing both the rule and the target (goal or assumption) should allow them to match exactly (assuming of course such a match does exist).

4.4.5.4 Metavariables

Metavariables in a proof are variables whose instantiations are delayed until the proof is finished. They are shared among all subgoals in the proof. Metavariables in HOL Light are represented as a list of terms (variables) paired with a corresponding instantiation. We use the latter to compose all metavariable instantiations that occur during the proof.

In Isabelle Light, in a proof step involving the application of a particular inference rule, any free variables in that rule that are not matched to any subterms of the assumptions or the goal are introduced as metavariables in the new subgoals. However, metavariables are also used for the construction of the computational component, i.e. the \(\pi\)-calculus term being composed, as described in Section 4.4.5.2.

If the computational component is initially unknown, it is set as an existentially quantified variable. Then the \texttt{META EXIST TAC} tactic of HOL Light, eliminates the existential quantifier and adds the variable to the list of metavariables. This allows it to be instantiated freely in the attempt to find the correct witness.

There are two issues involved in the handling of metavariables in our proofs. As pre-
Previously mentioned, in HOL Light metavariables are globally shared by all subgoals of a proof. They are, therefore, stored beyond the scope of any single subgoal. However, the HOL Light tactics are functions that take a single subgoal as an argument and return a new goal state that includes newly introduced metavariables, metavariable instantiations, and subgoals. Given that the tactics are applied to a single subgoal, they have no access to any information on already existing metavariables. This introduces two problems in our system:

- The first one involves metavariable matching. Whenever we attempt to match an assumption to our goal, metavariables in the goal can be freely instantiated to anything that matches. However, since the assumption matching tactics do not have access to the list of metavariables, we have to input this list to the tactic explicitly. Given the fact that a lot of metavariables are involved in our proofs, this list can become fairly large and, thus, the proof script becomes cumbersome. For this reason, we have implemented the by llmeta assumption command (see Figure 4.8b) which automatically extracts the metavariables from the current goal state, then calls upon the assumption matching tactic.

- The second problem involves metavariable reuse. In each of our proof steps, the process calculus variables of the inference rule being used that are not matched to anything in the current goal are introduced as new metavariables. However, given the lack of information on the metavariables that have already been used, it is not possible to enforce freshness of any newly introduced ones. In order to ensure the uniqueness of names, the variables of the inference rule being applied at every proof step are renamed using an internal numeric counter. This counter is maintained outside the HOL Light proof state and is incorporated within our specially implemented tactics and commands. Consequently, the use of generic HOL Light tactics outside our implementation could lead to unexpected results. Moreover, this renaming can only be applied consistently during an interactive proof and, therefore, it is not possible to package our proofs in one proof command, as is typical for HOL Light proofs.

Finally, as previously mentioned, unification is used when matching two terms involving metavariables, since metavariables can be instantiated to anything as long as the terms match in the end. HOL Light allows only first-order unification without

\footnote{Note that this is simply a syntactic incompatibility at the level of the proof script and does not affect the logical validity of the proofs.}
type instantiation. This is enough for metavariables involving processes, because no higher-order variables are used in these cases and the types of π-calculus names and processes are predetermined (: num and : (num)Agent respectively). However, using higher-order variables in a CLL specification would allow us to have a more expressive specification of our goal. For example, the goal ⊢ A ⊥, f(B) corresponds to a process with input A and an output of some function f of the resource B. The function f could incorporate information about possible exceptions (e.g. f(B) = B ⊕ E where E is an exception), which may not be available from the beginning (for example because it depends on the component services). If f was set as a metavariable, this would allow us to construct it as the proof progresses without the need of specifying it from the beginning. Unfortunately, the restrictions of the unification algorithm do not allow this level of expressiveness in our CLL specifications.

4.5 Related Work

In this section, we cover a few relevant topics related to CLL and the proofs-as-processes paradigm. More specifically, we cover variations of linear logic in Section 4.5.1 and the recent evolution of the proofs-as-processes theory in Section 4.5.2.

4.5.1 Variations of Linear Logic

Typically, depending on the particular application different versions of a given logic may be used, with consequences on the associated expressivity and complexity. In particular, linear logic has a number of variations from the MALL fragment used in our work. In what follows, we focus on the most commonly used ones. In particular, we cover linear units in Section 4.5.1.1, exponentials in Section 4.5.1.2, and Intuitionistic Linear Logic in Section 4.5.1.3.

4.5.1.1 Units

Four CLL constants, namely One (1), Zero (0), Top (⊥), and Bottom (⊥ - not to be confused with linear negation ·⊥) correspond to the four units of the ⊗, ⊕, &, and ⊨ respectively. The fragment of CLL used in the proofs-as-processes paradigm is unit-free and therefore no process calculus translation has been given for these rules in the
original work. Although there may be an intuitive interpretation of the four units in the context of processes, converting this to a formal, concrete process calculus translation is not straightforward. More importantly, the CLL units were not required in any of our HOL Light reasoning tactics (see Section 7.4) or the examples and use cases examined throughout this work.

4.5.1.2 Exponentials

The exponentials or the \textit{why-not} (\(?A\)) and \textit{of-course} (or bang, \(!A\)) operators allow a controlled application of weakening and contraction in CLL. In the resource interpretation of CLL, they can be used to represent \textit{replicable resources}, i.e. resources that can be replicated or destroyed arbitrarily. As a practical example, information contained in a database record can be trivially replicated, reused, or deleted without the need of additional resources. More specifically, a \textit{replicable receiver} of type \(?A\) (symmetrically a \textit{replicable sender} of type \(!A\)) can be seen as a receiver (sender) of resources of type \(A\) that can consume (produce) an arbitrary number of such resources, including none. The exact use of these modalities becomes clearer in the corresponding CLL inference rules shown in Figure 4.9.

\[
\begin{align*}
\vdash \Gamma & \vdash \Gamma, \ ?A \wedge \vdash \Gamma, \ A \\
\vdash \Gamma & \vdash \Gamma, \ ?A, \ ?A \\
\vdash \Gamma & \vdash \Gamma, \ ?A \\
\vdash \Gamma & \vdash \Gamma, \ ?A \\
\vdash \Gamma & \vdash \Gamma, \ ?A & \vdash \ ?\Gamma, \ B \ & \vdash \ ?\Gamma, \ !B
\end{align*}
\]

Figure 4.9: The one-sided sequent calculus versions of the CLL inference rules for the exponentials.

The inference rules for the exponentials also have proofs-as-processes translations to \(\pi\)-calculus processes. These are shown in Figure 4.10 and are explained in more detail next:

- \textbf{The \(?\) rules}: The \textit{why-not} rules intuitively represent the potential use of a replicable resource:
  - The first \textit{weakening} rule demonstrates that a process that does not receive any messages of type \(A\) can be used as a receiver of replicable resource \(?A\) that consumes \(A\) exactly 0 times (\textit{weakening receiver} or \textit{garbage collector}).
Figure 4.10: The CLL inference rules for the exponentials annotated with process calculus proof terms and the corresponding π-calculus processes.

- The second dereliction rule demonstrates that a process that receives a message of type $A$ can be used as a receiver of replicable resource $?A$ that consumes $A$ exactly once (dereliction receiver).

- The third and last contraction rule demonstrates that a process that acts as a receiver of two replicable resources $?A$ can be used as a receiver of type $?A$ that accepts replicable resource $?A$ and replicates it in order to satisfy its original requirement of two $?A$ messages (contraction receiver).

The channel $z$ in the π-calculus translation of these rules is used for the handshaking with a $!A$ type of sender, as described in the $!$ rule below.

**The ! rule:** For the intuitive interpretation of the of-course rule, consider a process $Q$ that only receives replicable resources $?\Gamma$ and sends a resource of type $B$. Since all the resources received are replicable, it is possible to destroy them all and produce...
no resources, or replicate them all multiple times and produce multiple resources of
type $B$. Therefore, it is possible to construct a replicable sender of type $!B$ using $Q$.
The $\pi$-calculus translation of this rule corresponds to a process that performs a triple
channel handshake with a replicable receiver $?B$. In this handshake, it transmits the
three possible channels through which it can send resources. Based on how the receiver
$?B$ was constructed, it will use one of the three channels to obtain the corresponding
functionality from $!B$:

- If $?B$ is a weakening receiver then it will select to use channel $w$ and receive
  nothing. The $!B$ sender will function as a weakening receiver (garbage collector)
  for the rest of the resources $\Gamma$. Note that $Q$ will never be executed in this case.

- If $?B$ is a dereliction receiver then it will select to use channel $d$ and receive a
  single $B$ by a single execution of $Q$.

- If $?B$ is a contraction receiver then it will select to use channel $c$. The sender
  then functions as a contraction receiver for the $\Gamma$ resources. This will allow it
to replicate those resources in order to execute $Q$ multiple times (by a recursive
call to itself) and satisfy the two $?B$ receivers.

Due to the limitations of our current formalisation of the $\pi$-calculus (see Section 3.5),
it is not possible to formalise the translation of the of-course operator $^x_\zeta(\overrightarrow{Q}_{\vec{x}})$ because
it is defined as a recursive agent.

The alternative option of using $\pi$-calculus replication instead of recursion is purpose-
fuly avoided by Bellin and Scott (Bellin and Scott, 1994). It is worth pointing out
that modern interpretations of propositions as session types use replicating channels
instead of replicating agents (see Section 4.5.2). More specifically, a process $\overrightarrow{!x(y)}P$
consists of a replicable receiving channel $x$, i.e. a channel $x$ that may receive informa-
tion $y$ once, never, or an infinite number of times. This waives the need for recursion.
Instead, the behaviour of replicating channels is dictated by separate process calculus
reduction rules.

### 4.5.1.3 Intuitionistic Linear Logic (ILL)

The intuitionistic version of linear logic (ILL) involves sentences in a two-sided se-
quent calculus where only one conclusion is allowed on the right hand side of the turn-
stile. The main differences with CLL as described so far is the lack of non-intuitionistic
operators $\otimes$ and $\oplus$, the lack of linear negation ($\bot$), and the introduction of linear implication ($\multimap$ - read as “lolli”). The inference rules of ILL are shown in Figure 4.11.

\[
\begin{align*}
\Gamma \vdash A \quad \Delta \vdash B & \quad \Gamma, \Delta \vdash C \\
\Gamma, A \cong B & \quad \Gamma \cong C \\
\Gamma \vdash A & \quad \Gamma \vdash A \\
\Gamma, A \otimes B & \vdash C \\
\Gamma, A \multimap B & \vdash C \\
\Gamma \vdash A & \quad \Gamma \vdash A \\
\Gamma, A \otimes B & \vdash C \\
\Gamma, A \multimap B & \vdash C \\
\Gamma, A \implies B & \vdash C \\
\Gamma, A \cong B & \vdash C \\
\Gamma, A \multimap B & \vdash C
\end{align*}
\]

Figure 4.11: The inference rules of ILL.

The linear implication $A \multimap B$ can be used to describe the notion of consuming resource $A$ to produce resource $B$. This makes ILL a more widely used version of linear logic than CLL, since it provides a straightforward mechanism to reason about resources in various contexts. However, the translation of linear implication in the context of proofs-as-processes is not immediately obvious. This is demonstrated, for example, in the inconsistent interpretation given by Rao et al. (see Section 5.4.1).

The ILL expression $A \multimap B$ is actually translatable to CLL as follows:

\[
A \multimap B \equiv A \bot B
\]

Even though the expression $A \bot B$ typically violates the polarisation of the $\otimes$ connective that we described in Section 4.2, linear implication matches conceptually with our limitation of having only one positive (output) term. Investigating this connection further is left as future work though.

ILL has, in fact, been used for reasoning about session types in concurrent processes.
by Caires and Pfenning (Caires and Pfenning, 2010). This work is a recent, further step in the evolution of the theory of the proofs-as-processes paradigm, which we discuss in the next section.

4.5.2 Evolution of the Proofs-as-processes Paradigm

The theory of proofs-as-processes as described in this chapter was originally published in 1994. However, until recently the potential of using linear logic to reason about concurrent systems remained underutilised for several years. This revival is presumably due to the increased use of parallel computation, mobile processes, and concurrent heterogeneous systems at various levels (from the multiprocessors of a single device to the modern cloud infrastructures) in the last decade. The need to formally verify these systems has led researchers to revisit formal models for parallel computation and communication, including the proofs-as-processes paradigm.

One of the widely used concepts for the modelling of concurrent systems is session types, originally introduced by Honda (Honda, 1993). The purpose of these types is to provide richer semantics for the types of communicated values and session-based protocols in mobile processes, e.g. in the work of Gay et al. (Gay et al., 2003). There have been efforts to use session types as a type system for the originally untyped π-calculus (for example by Giunti and Vasconcelos (Giunti and Vasconcelos, 2010)). Reasoning about the properties of concurrent systems with session types, including attempts to ensure deadlock freedom and session fidelity, has been a major research track in concurrency theory for the past years and is an ongoing effort.

The importance of linearity of session types has led towards the use of linear logic as a formal type system for sessions. This idea has led to the evolution of the proofs-as-processes paradigm from abstractly typed processes to session typed concurrent systems. In the next sections, we give a brief overview of two major efforts in this direction: the first one, described in Section 4.5.2.1 is led by Caires, Pfenning, and Toninho, whereas the second approach, analysed in Section 4.5.2.2 was introduced by Wadler. Finally, in Section 4.5.2.3 we discuss the relevance of these two session type based theories to the current project.
4.5.2.1 Caires et al.: Session Types as Intuitionistic Linear Propositions

In their work, Caires et al. use ILL (see Section 4.5.1.3) terms to describe session types and attach them via a proofs-as-processes (or Curry-Howard) style to π-calculus channels (Caires and Pfenning, 2010). The main differences from the original proofs-as-processes paradigm are the following:

- They use ILL instead of CLL. The duality of the connectives is replaced by the duality between the left hand side and right hand side inference rules of the two-sided sequent calculus.

- The version of π-calculus used has some extensions compared to the original polyadic π-calculus. The choice operator (+) is not included, and instead a name-based, binary choice mechanism is adopted with a case(a, b) operator offering two choices and two selectors inl and inr. This enforces much stricter control over communications that involve choice. Moreover, the replication operator (!) is applied over channels (e.g. !a(x).P) rather than processes. As a result, recursion is not included.

- ILL terms represent session types. As a result, they have a different interpretation to the one we gave in Section 4.2. For example, the term A ⊗ B is interpreted as “the type of a session channel that first performs an output (sending a session channel of type A) to its partner before proceeding as specified by B” (Caires and Pfenning, 2010). This imposes an order on the two outputs as opposed to the interpretation of A ⊗ B as two simultaneous outputs in the original proofs-as-processes paradigm.

- The ILL axiom A ⊩ A is not included in the theory, i.e. it does not have a translation to a session typed process. This is particularly important in the context of process composition since the axiom is commonly used within our composition tactics (see Chapter 7).

It is important to note that a lot of the ambiguity and complexity of the original proofs-as-processes patterns is resolved in the theory of Caires et al. as a result of the differences described above. For example, the lack of input/output synchronisation (see Section 6.3) and the difficulty of formalising replication as recursion (see Section 3.5) are both resolved based on the use of linear implication and ⊗ in a way that enforces ordering constraints and the use of replicable channels (as opposed to processes) re-
Caires et al. provide proofs to demonstrate the properties of their mapping, including the strong correspondence between (their version of) the $\pi$-calculus reductions (computation) and cut-elimination steps in ILL and deadlock freedom.

Subsequent published papers describe further developments of this theory, including a version for asynchronous communication (DeYoung et al., 2012), the development of a theory of linear relations (Perez et al., 2012), a comparison to a CLL based version (Caires et al., 2012), and the use of dependent session types to describe properties about the information being communicated (Toninho et al., 2011; Pfenning et al., 2011). The latter is a very interesting concept which has the potential for allowing more expressive formal specifications of the modelled processes.

4.5.2.2 Wadler: Propositions as Sessions

In order to overcome the limitations of both the original proofs-as-processes and the work of Caires et al., Wadler recently developed a new propositions-as-sessions theory (Wadler, 2012). In it, he chooses to loosen the connection to the original $\pi$-calculus by introducing a new process calculus named CP. The syntax of CP is shown in Figure 4.12. The process calculus is defined as a direct mapping of first order CLL types to process terms. Reductions in CP are defined based on cut-elimination steps in CLL proofs. This is in contrast to the previous two theories where $\pi$-calculus reduction rules are defined separately and then the proof translations are proven to provide a tight correspondence between these rules and cut-elimination.

In the propositions-as-sessions theory, CLL terms are mapped to session types using a similar interpretation to that of Caires et al. In addition, first order CLL terms are used to allow polymorphism. The duality of input and output connectives is applied in both the CLL types and CP. It is also worth noting that the link $x \leftrightarrow y$ is used in the CLL axiom $\vdash x \leftrightarrow y :: x : A^\perp, \ y : A$ and allows bidirectional forwarding for any number of times (as opposed to the unidirectional, atomic buffer in proofs-as-processes).

Wadler also criticises the lack of an implementable programming language in the work of Caires et al. For this reason, he introduces the session typed functional language GV and a formal translation to CP.
\[ P ::= \begin{align*}
\ & x \leftrightarrow y \quad \text{link} \\
\ & \nu x : A(P | P) \quad \text{parallel composition} \\
\ & \nu x[y] . (P | P) \quad \text{output} \\
\ & x(y).P \quad \text{input} \\
\ & x[\text{inl}].P \quad \text{left selection} \\
\ & x[\text{inr}].P \quad \text{right selection} \\
\ & x.\text{case}(P,P) \quad \text{choice} \\
\ & ?x[y].P \quad \text{replicated output} \\
\ & !x(y).P \quad \text{replicated input} \\
\ & x[A].P \quad \text{send type} \\
\ & x(A).P \quad \text{receive type} \\
\ & x[].P \quad \text{empty output} \\
\ & x().P \quad \text{empty input} \\
\ & x.\text{case}() \quad \text{empty choice}
\end{align*} \]

Figure 4.12: The \( CP \) syntax in BNF grammar.

### 4.5.2.3 Relevance to the Current Work

Both of the approaches of Caires et al. and Wadler provide a systematic, formal account from a proof theoretic perspective. Their ongoing research, in which Caires et al. continue to publish papers and evolve their work, whereas Wadler is leading a large scale project to apply these ideas to industrial programming languages, is showing promising results while addressing interesting issues.

In the current project, we chose to rely on the original proofs-as-processes paradigm for two main reasons:

1. These newer theories rely on process calculi that divert from the original \( \pi \)-calculus. This would introduce a major disadvantage for the current project. The original \( \pi \)-calculus with its original syntax has been analysed and built upon for years. For example, in Section 3.3 we presented a number of useful tools for the \( \pi \)-calculus, among which the PiVizTool has proven invaluable for the simulation, empirical verification, and understanding of the constructed process compositions. Moreover, in Section 8.2.2 we introduce a library that implements \( \pi \)-calculus terms directly into code. The session typed process calculi presented in the two approaches discussed here remain at a theoretical stage and have not
been implemented. In fact, there are very few known, prototype implementations of session types with limited application in an industrial setting ([Pucella and Tov, 2008]). The focus of our project is not towards a deeply theoretical investigation of the problem, but rather the bridging of a strong theoretical background with a pragmatic, usable solution for process composition.

2. Both Caires et al. and Wadler developed their theories after we had already made significant progress towards the formalisation of the proofs-as-processes paradigm in HOL Light, and for the most part this happened in parallel with the current work. Switching our implementation to use the newer theories ran the risk of creating delays and setting back the schedule for our work. A considerable amount of time would have been required to understand, analyse, and formalise these theories to a level comparable to the one presented here for the proofs-as-processes paradigm. In retrospect, we believe our time was utilised more effectively towards developing a pragmatic, deployable solution and investigating practical use cases which will be demonstrated in subsequent chapters.

Having said this, we do anticipate useful developments of the two session types process calculi. More specifically, we expect the evaluation and further refinement of these theories, as well as the development of simulation, visualisation, and deployment tools.

Our HOL Light framework, as it currently stands, is flexible enough to accommodate modern theories involving both CLL and ILL. The composition tactics (see Chapter 7) can generally be reconstructed for any proofs-as-processes based theory with minimum effort. To demonstrate this and as part of our investigation of Wadler’s CP calculus, we attempted a formalisation of CP and the propositions-as-sessions theory in HOL Light, following the same steps as for the proofs-as-processes paradigm. Our reasoning tools (see Section 4.4.5) and process composition framework (see Chapter 7) worked out-of-the-box with minimal effort and mostly involved re-proving the necessary derived inference rules (see Section 7.2) for the new theory. It is worth noting that the only visible limitation is the fact that the current level of embedding in HOL Light cannot trivially support first order binders.

This investigation showed that our framework is flexible and adaptable to the newer proofs-as-processes based theories that are currently under development. It is possible to adapt our formalisation to use similar CLL or even ILL based theories and reuse the available reasoning tools and tactics to a large extent. There is, however, significantly
more work required to provide simulation and automated deployment functionality for these theories since they are currently at an early stage.

4.6 Conclusion and Future Work

Classical Linear Logic is a natural fit for reasoning tasks that involve information and resources, thanks to its inherent capability for systematic resource accounting. Using the Multiplicative Additive fragment (MALL) we are able to describe input and output types for processes, including simultaneous and conditional ones. Moreover, work by Abramsky, and Bellin and Scott has resulted in the development of the so-called proofs-as-processes paradigm. Based on this, there is a systematic correspondence between CLL proofs and concurrent, deadlock-free π-calculus processes. The inherent properties of CLL and the proofs-as-processes paradigm make this the perfect back-end for the formal verification of process workflows.

In this chapter, we gave an overview of CLL and the proofs-as-processes correspondence while focusing on the particular context of process specification and composition. We then discussed our approach towards the formalisation of the introduced inference rules in HOL Light. In particular, we tackle the challenging task of developing a CLL-based proof system that allows both the construction and verification of corresponding π-calculus processes. Our system relies on building CLL proof trees using natural deduction at the meta-level, exploiting metavariables, and automating the required low-level reasoning tasks such as multiset matching.

Since our formalisation is a deep embedding of the proofs-as-processes paradigm, in addition to a practical proof system, it also allows the performance of meta-level reasoning. As an example, Bellin and Scott give a deep analysis of the proof-theory of CLL and its process interpretation and, among other results, they present soundness and completeness proofs for the π-calculus mappings. Formalising these would make for an interesting future undertaking that would further solidify the correctness of our framework. Our research is focused towards a practical application rather than purely theoretical endeavours, so we believe there is room for improving the support for this kind of reasoning.

Finally, another interesting direction for future work is the generalisation of our embedding in such a way that it can easily be reused to formalise similar theories such
as Wadler’s Propositions as Sessions work. Our modular implementation already provides the flexibility to experiment with these theories, and a lot of our infrastructure can be reused in a straightforward way. Investigating the results of using these theories in the context of process specification and composition in the same way as the proofs-as-processes paradigm is used in the current work will certainly be an interesting future path for our research.
Chapter 5

Formal Analysis of Rao et al.’s Approach

The initial stages of our research were primarily motivated by the work of Rao et al. (Rao, 2004) on the composition of web services using linear logic. In our attempt to reconstruct this approach rigorously within the sound logical framework of HOL Light, we encountered a number of issues and roadblocks. Dealing with these led to a number of interesting observations, which we analyse next, and ultimately to a better understanding of process interpretations of CLL proofs within the proofs-as-processes paradigm.

5.1 Introduction

Our approach to process composition has its roots in the research area of AI planning (Hendler et al., 1990). In a pure AI planning approach, processes can be viewed as actions with specified preconditions and effects, inputs and outputs. A planner then attempts to discover the appropriate combination of actions that will lead to a goal state starting from an initial state. An example of such system for web services is SWORD (Ponnekanti and Fox, 2002) which uses rule-based planning for the composition of such services.

Theorem proving techniques follow the same philosophy. Preconditions and effects of processes are represented using logical expressions. The prover then uses these to attempt a proof of the logical expression of the goal state. The plan is finally ex-
extracted from the proof as each proof step corresponds to some interaction between the processes.

There have been multiple attempts at using a theorem prover in this context. For instance, Waldinger based his work on automated deduction and program synthesis (Waldinger, 2001). He used the theorem prover SNARK (Stickel et al., 2000) to provide proofs for service composition problems described in classical first-order logic. Lammermann worked on structural synthesis of program, a deductive approach that utilises intuitionistic propositional logic (Lämmernann, 2002). Finally, Rao et al. used propositional Linear Logic theorem proving with DAML-S based proofs in an approach based on the proofs-as-processes paradigm (Rao, 2004).

The work of Rao et al. provided us with sufficient motivation to pursue further research on the subject of formally verified process composition using a theorem prover. In the initial stages of this research, a significant amount of time was dedicated to the analysis of their work. Our aim was to formalise their approach, emulate the translation and the proof used in their main example both on paper and in HOL Light, and investigate ways to expand the capabilities and functionality of such a formalised system.

This effort led to the uncovering of several areas where we believe that the soundness and correctness of the approach of Rao et al. are compromised. In particular, the core differences between their formulation and the original proofs-as-processes theory, the fact that no formal semantics or connections to the \( \pi \)-calculus were provided for the process calculus being used, and the lack of an available implementation drastically increased the complexity of formalising and verifying the results of Rao et al.

We present our findings in more detail in the following sections. Section 5.2 includes an overview of their approach and methodology, which, to a certain extent, has been reused in this project. This is followed by a description of the process calculus introduced by Rao et al. and an analysis of the uncovered inconsistencies and differences from the standard \( \pi \)-calculus in Section 5.3. Next, in Section 5.4, we discuss the issues with ILL and their process calculus that emerged from our effort to formalise their associated correspondence in HOL Light. Finally, in Section 5.5, we introduce the notion of filters, which correspond to formal, proofs-as-processes based reconstructions of the congruence rules introduced by Rao et al.
5.2 The Work of Rao et al.

In this Section we briefly describe the work of Rao et al. on Semantic Web Services Composition using Intuitionistic Linear Logic [Rao and Su 2004, Rao et al. 2004, Rao 2004, Rao et al. 2006]. Their work involves automated composition of DAML-S web services in 3 steps: translation from the DAML-S service profiles to ILL axioms, proof of the requested service (expressed as an ILL theorem), and extraction of the composite process model from the proof. The proof is achieved using an automated ILL theorem prover.

The work of Rao et al. relies on an extension of propositional ILL with process calculus proof terms. This is based on the proofs-as-processes paradigm introduced by Abramsky (Abramsky, 1994), Bellin, and Scott (Bellin and Scott, 1994) (see Chapter 4), but also includes significant modifications of the original theory. We examine these in more detail in subsequent sections.

We begin our analysis in Section 5.2.1 with the example of ordering a ski set as described by Rao et al. and then give more details about each of the 3 stages of their approach, namely the translation in Section 5.2.2, the composition via proof in Section 5.2.3, and the extraction of the result in Section 5.2.4.

5.2.1 Example

In order to demonstrate the functionality of their framework, Rao et al. present a case of ordering a ski set [Rao et al. 2006]. In this, a core service is composed with value-adding services, i.e. services that have minor, independent functionality, such as currency or measurement conversion, that can be used as an addon to the core functionality. The core service selectSki, returns the price in US dollars of a ski set using the ski length, brand, and model as input parameters. The functionalities of various value-adding services are also illustrated diagrammatically in Figure 5.1.

Given the user’s height, weight, and skill level as well as a price limit, the requested composite service must return the price of a ski set in Norwegian Crowns (NOK). The diagram corresponding to this requested service can be found in Figure 5.2.

Some non-functional attributes such as the cost or certification required to run a service have been attached to the specifications of some of the ski services. However, these are
only presented to demonstrate the possibility of specifying non-functional attributes in Rao et al.’s framework and are ignored in the rest of their presentation provided so that the composition proof is, as stated by the authors, “simple and easy to read” (Rao et al., 2006).

### 5.2.2 Translation

The first step of the composition process is the translation of the DAML-S service profiles to ILL axioms with proof terms. An excerpt of a DAML-S service profile is shown in Figure 5.3 for illustrative purposes. Note that we have presented ILL in Section 4.5.1.3. According to Rao et al., any Semantic Web Service can be specified using the following general ILL formula:

\[
(\Delta_{cn} \otimes \Delta_{qc}) \vdash (P \otimes I \rightarrow (F \otimes O) \oplus E) \otimes \Delta_{qr}
\]  

(5.1)

In the assumption we have a multiplicative conjunction of non-functional constraints. These are split into two categories: consumable quantitative attributes ($\Delta_{cn}$) and qualitative constraints ($\Delta_{qc}$). The description of the functional parameters of the service is given by $P \otimes I \rightarrow (F \otimes O) \oplus E$ where $P$ represents the preconditions, $I$ represents the input, $O$ is the output, $F$ represents the effects and $E$ an exception of the web service. Finally, $\Delta_{qr}$ is used to describe non-functional qualitative results.

Each of the input and output parameters is extracted from the DAML-S process model
5.2. The Work of Rao et al.

<profileHierarchy:BookSelling>
  rdf:ID="Profile_Congo_BookBuying_Service">
  ...
  <profile:serviceName>Congo_BookBuying_Agent</profile:serviceName>
  ...
  <profile:input>
    <profile:ParameterDescription rdf:ID="BookTitle">
      <profile:parameterName> bookTitle </profile:parameterName>
      <profile:restrictedTo rdf:resource="&xsd;#string"/>
      <profile:refersTo rdf:resource="&congoProcess;#bookName"/>
    </profile:ParameterDescription>
  </profile:input>
  ...
  <profile:output>
    <profile:ParameterDescription rdf:ID="EReceipt">
      <profile:parameterName> EReceipt </profile:parameterName>
      <profile:restrictedTo rdf:resource="&congoProcess;#EReceipt"/>
      <profile:refersTo rdf:resource="&congoProcess;#congoBuyReceipt"/>
    </profile:ParameterDescription>
  </profile:output>
  ...
  <profile:precondition>
    <profile:ParameterDescription rdf:ID="AcctExists">
      <profile:parameterName> AcctExists </profile:parameterName>
      <profile:restrictedTo rdf:resource="&congoProcess;#AcctExists"/>
      <profile:refersTo rdf:resource="&congoProcess;#congoBuyAcctExistsPrecondition"/>
    </profile:ParameterDescription>
  </profile:precondition>
  ...
  <profile:effect>
    <profile:ParameterDescription rdf:ID="BuyEffectType">
      <profile:parameterName> BuyEffectType </profile:parameterName>
      <profile:restrictedTo rdf:resource="&congoProcess;#BuyEffectType"/>
      <profile:refersTo rdf:resource="&congoProcess;#congoBuyEffect"/>
    </profile:ParameterDescription>
  </profile:effect>
</profileHierarchy:BookSelling>

Figure 5.3: Excerpt of the DAML-S web service profile for the “Congo” book service (Ankolekar et al., 2002).
automatically and represented as $!X(Y)$ where $X$ is an ILL proposition that corresponds to the type of the parameter and $Y$ is a proof term that restricts the proposition and counts as evidence for it, following the Curry-Howard isomorphism and the proofs-as-programs paradigm. Using the more traditional type-theoretic notation, $!X(Y)$ is equivalent to $Y : !X$. Note that the of course operator “!” is used to express the fact that the input and output parameters consist of information that can be replicated. States, including preconditions and effects, are represented using simple ILL propositions without attached proof terms. The of course modality is not used in this case since once a state changes, it is no longer applicable. Finally, linear implications ($\to$) describe the data flow (consumption of inputs to produce outputs) during a web service execution and are annotated with the name of the corresponding service, even though this is not shown in the original formula (5.1) as given by Rao et al.

It is worth noting that the way the formula (5.1) is presented by Rao et al., is slightly misleading at first since it appears only one of each attribute can be specified for each service. It is only later explained that the symbols $\Delta_{cn}, \Delta_{qc}, \Delta_{qr}, P, I, F,$ and $O$ actually represent multiplicative conjunctions ($\otimes$) of resources. The case of having more than one possible types of exceptions thrown by the same service is not considered in these ILL specifications.

The available services and the request for the Ski example translated into ILL including the proof terms are shown in Figure 5.4. The proof terms are given using lower-case abbreviations corresponding to the initials of the service and each parameter name. For example, the proof term for the \textit{PRICE LIMIT} parameter of the \textit{SelectModel} service is $smp$ which stands for \textit{Select Model} \textit{Price limit}. Rao et al. claim this proof term corresponds to an ontological concept from the DAML-S Service Model that has been attached to this parameter.

\section*{5.2.3 Composition as Proof}

The composite service request is expressed in ILL using the same formula (5.1) as for the available services with the addition of the available services specifications as assumptions. In order to achieve the proof of the requested service, a set of annotated ILL inference rules are used in the spirit of the proofs-as-processes paradigm (see Figure 4.3). However, instead of the $\pi$-calculus, Rao et al. introduce their own, more complicated process calculus syntax. This is presented in their papers as the grammar
5.2. The Work of Rao et al.

Available services:

\[
\vdash PRICE\_LIMIT(smp) \otimes SKILL\_LEVEL(sms) \rightarrow_{\text{SelectModel}} BRAND(smb) \otimes MODEL(smm)
\]

\[
\vdash HEIGHT\_CM(slh) \otimes WEIGHT\_KG(slw) \rightarrow_{\text{SelectLength}} LENGTH\_CM(sll)
\]

\[
\vdash LENGTH\_CM(cic) \rightarrow_{\text{CM2INCH}} LENGTH\_IN(cii)
\]

\[
\vdash PRICE\_USD(unu) \rightarrow_{\text{USD2NOK}} PRICE\_NOK(unn)
\]

\[
\vdash LENGTH\_IN(ssl) \otimes BRAND(ssb) \otimes MODEL(ssm) \rightarrow_{\text{SelectSki}} PRICE\_USD(ssp) \oplus \text{EXCEPTION}(sse)
\]

Request:

\[
\vdash PRICE\_LIMIT \otimes SKILL\_LEVEL \otimes HEIGHT\_CM \otimes WEIGHT\_KG \rightarrow PRICE\_NOK \oplus \text{EXCEPTION}
\]

Figure 5.4: The Ski example translated into ILL with proof terms.

shown in Figure 5.5. We discuss this further in Section 5.3.

Their inference rules are presented in Figure 5.6. In our representation we have remained faithful to the figures provided by Rao et al. in their various papers, except for our use of the more traditional “::” operator as opposed to their “:” to avoid confusion. We also remind the reader that the term \( A(x) \) in Rao et al.’s notation corresponds to \( x:A \) in standard type theory notation and that ILL operators take precedence over proof annotations, i.e. for example \( A \otimes B(a,b) = (A \otimes B)(a,b) \).

Rao et al. also introduce a set of structural congruence rules for the manipulation of ILL terms that have process calculus proof terms attached. These rules, which can be viewed in Figure 5.7, appear to be axiomatically defined and not derived from the existing theory. We also note that the “Shift” rule introduced by Rao et al. (see Figure 5.6) is derived from the standard ILL rules and their structural congruence rules.

The composition proofs are achieved using an ILL theorem prover. More specifically, Rao et al. used Lolli (Hodas and Miller, 1994) in their initial versions of their system, then Forum (Miller, 1996) and in the later versions RAPS (Küngas, 2002). Part of the proof for the Ski example (without non-functional parameters as previously noted) is shown in Figure 5.8.
\[\langle \text{Process} \rangle ::= \langle \text{Inputs} \rangle . \langle \text{Process} \rangle \]
\[\quad | \langle \text{Process} \rangle . \langle \text{Outputs} \rangle \]
\[\quad | \langle \text{Process} \rangle . (\langle \text{Outputs} \rangle + \langle \text{Exception} \rangle ) \]
\[\quad | \langle \text{Channels} \rangle . \langle \text{Process} \rangle \]
\[\quad | (\nu \langle \text{Variable} \rangle ) \langle \text{Process} \rangle \]
\[\quad | \langle \text{Process} \rangle . \langle \text{Process} \rangle \]
\[\quad | \langle \text{Process} \rangle || \langle \text{Process} \rangle \]
\[\quad | \langle \text{Process} \rangle + \langle \text{Process} \rangle \]
\[\quad | \langle \text{Service} \rangle \]
\[\quad | 0 \]
\[\langle \text{Inputs} \rangle ::= (\langle \text{InputPort} \rangle, \langle \text{Inputs} \rangle) \]
\[\quad | (\langle \text{InputPort} \rangle) \]
\[\langle \text{Outputs} \rangle ::= (\langle \text{OutputPort} \rangle, \langle \text{Outputs} \rangle) \]
\[\quad | (\langle \text{OutputPort} \rangle) \]
\[\langle \text{Channels} \rangle ::= (\langle \text{Channel} \rangle, \langle \text{Channels} \rangle) \]
\[\quad | (\langle \text{Channel} \rangle + \langle \text{Channel} \rangle) \]
\[\quad | (\langle \text{Channel} \rangle) \]
\[\langle \text{Variable} \rangle ::= \langle \text{Port} \rangle \]
\[\langle \text{InputPort} \rangle ::= \langle \text{Port} \rangle \]
\[\langle \text{OutputPort} \rangle ::= \langle \text{Port} \rangle \]
\[\langle \text{Exception} \rangle ::= \langle \text{Port} \rangle \]
\[\langle \text{Port} \rangle ::= ![a-z][A-Za-z0-9\#]^* \]
\[\langle \text{Service} \rangle ::= [A-Z][A-Za-z0-9\#]^* \]
\[\langle \text{Channel} \rangle ::= \langle \text{OutputPort} \rangle \langle \text{InputPort} \rangle \]
\[\quad | (\langle \text{OutputPort} \rangle \langle \text{Variable} \rangle) \]
\[\quad | (\langle \text{Variable} \rangle \langle \text{InputPort} \rangle) \]
\[\quad | (\langle \text{Exception} \rangle \langle \text{Variable} \rangle) \]
\[\quad | (\langle \text{Variable} \rangle \langle \text{Variable} \rangle) \]

(5.2)

Figure 5.5: The grammar for Rao et al.’s process calculus which differs significantly from the \(\pi\)-calculus.
5.2. The Work of Rao et al.

Logical axiom and Cut rule

\[ A(x) \vdash (\forall x)0::A(x) \quad \text{Id} \]
\[ \Gamma \vdash P::A(a_1) \quad \Gamma', A(a_2) \vdash Q::C \quad \text{Cut} \]

Rules for propositional constants

\[ \Gamma \vdash \top \]
\[ \Gamma, A(a) \vdash P::C \quad \Gamma', A(a) \vdash Q::C \]
\[ \Gamma, A(a), B(b) \vdash P::C \quad \Gamma', A(a) \vdash Q::C \]
\[ \Gamma \vdash \bot \]

Rules for exponential !:

\[ \Gamma \vdash \Delta \]
\[ \Gamma, !A(0) \vdash \Delta \quad \text{W!} \]
\[ \Gamma, A(a) \vdash \Delta \]
\[ \Gamma, !A(0) \vdash \Delta \quad \text{L!} \]
\[ \Gamma, !A(x), !A(\langle x \rangle) \vdash \Delta \]
\[ \Gamma, !A(a) \vdash \Delta \quad \text{C!} \]

Figure 5.6: ILL inference rules combined with process calculus proof terms according to Rao et al.
Chapter 5. Formal Analysis of Rao et al.’s Approach

Commutative law:
\[ A \otimes B(a,b) \equiv B \otimes A(b,a) \]
\[ A \& B(a,b) \equiv B \& A(b,a) \]
\[ (P + Q) :: A \oplus B(a+b) \equiv (Q + P) :: B \oplus A(b+a) \]

Associative law:
\[ (A(a) \otimes B(b)) \otimes C(c) \equiv A(a) \otimes (B(b) \otimes C(c)) \equiv A \otimes B \otimes C(a,b,c) \]
\[ (A(a) \& B(b)) \& C(c) \equiv A(a) \& (B(b) \& C(c)) \equiv A \& B \& C(a,b,c) \]
\[ (A(a) \oplus B(b)) \oplus C(c) \equiv A(a) \oplus (B(b) \oplus C(c)) \equiv A \oplus B \oplus C(a+b+c) \]

Interaction law:
\[ (a_1,a_2,...,a_n)(b_1,b_2,...,b_n) \equiv (a_1b_1,a_2b_2,...,a_nb_n) \]

Replication law:
\[ !(A(a)) \equiv A(!a) \]

Figure 5.7: Structural congruence rules for ILL with process calculus proof terms as introduced by Rao et al.

5.2.4 Extraction

Once the proof is complete, the extraction of the process model as a process calculus formula appears to be trivial. As described, each of the applied inference rules is propagated as some operation in the process calculus. A concluded proof contains a single process calculus formula that is the description of the composite process. The resulting formula from the Ski example is shown in Figure 5.9.

Rao et al. propose an Upper Ontology as a general description for composite processes that allows the resulting composed services to be easily derived from the process calculus formulas and then translated to any desirable language such as DAML-S (Ankolekar et al., 2002) or BPEL4WS (Andrews et al., 2003). The resulting composite process for the Ski example in the language of the suggested Upper Ontology is given in Figure 5.10.

The approach adopted by Rao et al. initially appeared to be solid and motivational. In our first attempts in the current project, we aimed to reconstruct this work within HOL Light, formally verify its correctness, and extend it. However, our analysis from a formal methods stand-point revealed a number of issues that, we believe, were not sufficiently clarified in the work of Rao et al. These points were both at the implementation and at the theoretical level and are explained in more detail in the upcoming sections.
Figure 5.8: Part of the proof for the Ski example.
(\(\forall x_1\))(smp,sms,slh,slw).((SelectModel||SelectLength).\overline{sll}cic.CM2INCH)\(\overline{smb}\overline{sbb},\overline{smm}\overline{ssm,\overline{cissl}}\).SelectSki(sspusu + ssex_1).USD2NOK.unn + x_1

Figure 5.9: The resulting process calculus formula from the proof for the Ski example provided by Rao et al.

<hasInput smp>
<hasInput sms>
<hasInput slh>
<hasInput slw>
<hasControl sequence>
<hasSubProcess [>
  <hasControl split>
  <hasSubProcess [SelectModel]>>
<hasSubProcess [>
  <hasControl sequence>
  <hasSubProcess [SelectLength, CM2INCH]>
  <hasDataflow from sll to cic>,
 ]>
>  <hasSubProcess [SelectSki]>
<hasSubProcess [USD2NOK]>
>  <hasDataflow from smb to ssb>
<hasDataflow from smm to ssm>
<hasDataflow from cii to ssl>
<hasDataflow from smb to ssb>
<hasDataflow from ssp to unu>
<hasDataflow from sse to x1>
<hasOutput unn>
<hasException x1>
<hasVariable x1>
]

Figure 5.10: The description of the resulting composite service from the Ski example.
5.3 Issues with the Process Calculus

In our attempt to formalise the syntax of the process calculus used by Rao et al. as shown in Figure 5.5, we detected a number of what appeared to be inconsistencies.

In Rao’s PhD thesis (Rao, 2004), a grammar for the polyadic π-calculus is presented, which does not match the original polyadic π-calculus syntax. In fact, Rao et al.’s version of the calculus appears to be monadic as, throughout their work, only single names and not vectors of names are included in channels. Moreover, they introduce a sequence connective “.” which does not appear in the original π-calculus syntax. We analyse this connective further in the next section.

Rao et al. claim their introduced π-calculus is equivalent to the aforementioned version of the π-calculus with sequence. However, there are a number of features that are significantly different from the original π-calculus. These include, but are not limited to, alterations in the syntax with the introduction of exceptions as separate names and putting the senders on the right hand side of the process, composite and optional ports (see Section 5.3.3), and the distinct notion of channels and composite channels.

These differences introduced considerable complexity in our attempt to formalise and reason about this syntax. As a justification for his changes, Rao mentions that this augmented syntax strengthens the specific features of Web Services (Rao, 2004). His effort to analyse the reasoning behind this argument may be convincing, but no guarantees are given about the soundness of the resulting process calculus or its equivalence to the original π-calculus. The lack of such an equivalence may break the soundness and completeness guarantees provided by Bellin and Scott about the π-calculus translation to Linear Logic. Therefore, the properties offered by reasoning in CLL using the proofs-as-processes paradigm, such as type correctness, deadlock freedom, and systematic tracking of resources, are not guaranteed in Rao et al.’s process calculus and the constructed processes.

In our effort towards the formalisation of Rao et al.’s process calculus, we were able to reconstruct equivalent π-calculus terms with the same properties as some of the introduced features. However, we found some properties to be inconsistent with the π-calculus, a state of affair that cast further doubt on Rao et al.’s claims. We describe our results in more detail in the next few sections.
5.3.1 Encoding Sequence in π-calculus

The workflow of a DAML-S composite service is described based on its elements, which are themselves services (either atomic or composite), connected through control flow connectives including sequence, choice, loop, and split. Therefore, any composite service that is extracted from an ILL proof must be translatable to a syntax that uses these four DAML-S connectives. However, Rao et al. use process calculus formulas for the extraction of the composite service, where choice and split are explicitly defined but loops are not supported. Note that loops are normally expressed in the π-calculus through recursion and are explicitly avoided in the proofs-as-processes paradigm in order to ensure deadlock freedom.

The issue arises at the expression of sequentially composed processes in the π-calculus. Since the π-calculus is used to model concurrent processes, there is no explicit declaration of a sequence. The communication when a process $Q$ requires input that matches the output of another process $P$ is accomplished via names over their channels. The communication itself is modelled as a reduction in the π-calculus denoted by the following formula that we discussed in Section 3.2:

$$\pi\langle \vec{b} \rangle. P \parallel x\langle \vec{a} \rangle. Q \rightarrow P \parallel Q[\vec{b} / \vec{a}]$$

In the context of process workflows, the explicit description of a sequential composition is required though. Rao et al. claim to overcome this problem by using an augmented version of the π-calculus. They use the already mentioned “;” operator as a sequence connective between processes (this is in addition to the use of the dot to indicate the connection of a process to its channel as in the normal π-calculus). Then, in order to represent the interaction between two processes $P$ and $Q$ when invoked in sequence, the following formula is used:

$$(\nu y)(P.\text{output}P(y).\text{input}Q(y).Q \rightarrow P.(\text{output}P\text{input}Q).Q)$$

Note that the variable $y$ is declared as local and is omitted from the resulting form “for simplicity” (Rao, 2004). The differences between this interaction and the standard π-calculus reduction are apparent and fundamental. Important concepts such as substitution and reduction are replaced by what appears to be an arbitrary addition in the syntax.
5.3. Issues with the Process Calculus

In essence, Rao et al. define a new process calculus which is an augmented version of the $\pi$-calculus. However, as far as we can tell, no guarantees are given that this new process calculus, as well as its interaction with ILL, are sound. Indeed, any of the guarantees based on the work of Abramsky (Abramsky, 1994), Bellin and Scott (Bellin and Scott, 1994) only involve the original $\pi$-calculus and not the augmented one. In addition, no connection or systematic translation between the two process calculi is provided. Therefore, there is no indication that the augmentation of $\pi$-calculus is safe in the sense that it can actually result in a correct, type checked, deadlock free composition in all situations.

5.3.2 Channels

As mentioned in Section 3.1, in the $\pi$-calculus we often use the terms ports, channels, and messages to refer to the names being used. This terminology does not have any formal justification, nor is there a formal distinction between them. For example, we say that the process $a(b).\overline{b}(c).P$ will receive message $b$ through channel $a$ and then send message $c$ through channel $b$. Notice how $b$ is used both as a message and as the channel. In fact, the process receives information about which channel it must use to send $c$. Similarly, there is no distinction between a port and a channel. Whether a name (for example $a$) is considered an open port waiting for input or a channel connected to another process only becomes apparent through the interactions between the parallel processes, i.e. the $\pi$-calculus reductions.

In Rao’s PhD thesis (Rao, 2004), a clear distinction is made between channels and ports. This is apparent in the syntax for the process calculus being used (see Figure 5.5). The justification given for the explicit introduction of channels in the process calculus is the attempt to model processes from the service requester’s point of view. The requirement in this case, according to Rao, is that the direction of the information flow and the sequence of the processes must be explicit. It is also stated that “the channel pair is not a standard presentation in $\pi$-calculus, although it can be rewritten using $\pi$-calculus formulae”. However, there is insufficient evidence as to whether this statement is true and whether there is indeed a sound equivalent representation for channels in the $\pi$-calculus. Our doubts arise because there is no explicit representation of sequence or information flow in the $\pi$-calculus. These properties are only inferred through the interactions of otherwise parallel processes.
5.3.3 Ports

In addition to channels, Rao et al. introduce composite and optional ports. It is claimed that these are only shortcuts for actual $\pi$-calculus terms, but no evidence or any guarantees are given to support this claim. We have therefore analysed these concepts and attempted to find equivalent $\pi$-calculus terms to represent them.

- **Composite ports:** It is stated that “the port composition enables us to construct a new port that includes two parts” and that “each part is an existing port defined previously” (Rao, 2004). A composite port consisting of ports $a$ and $b$ is then represented as $(a, b)$. Our analysis of the proofs-as-processes paradigm using our formalised version (see Section 4.4) showed that composite ports can indeed be represented using the $\pi$-calculus. However, the actual representation is more complicated than a simple pair of ports and also implies certain properties that have not been explicitly defined for Rao et al.’s composite ports. Moreover, there is a distinction between the representation of composite output ports and composite input ports. We discuss these issues next.

In our attempt to understand the (otherwise hidden) semantics of a composite port, we investigate their use within ILL proofs. Based on the ILL to process calculus correspondence as given in Figure 5.6, we notice that composite ports correspond to the types $A \otimes B$ and $A \& B$. We further describe the latter in Section 5.4.2. Based on our resource interpretation of LL, the term $A \otimes B$ corresponds to a composite resource that involves both $A$ and $B$ in parallel. Note that in ILL whether or not $A \otimes B$ is an input or an output depends on its position in the judgement (see Section 5.4.2). Therefore, $A \otimes B$ corresponds to either a double input or double output of resources of type $A$ and $B$.

Typically, an agent $P$ with two output ports $a$ and $b$ that output messages $\vec{o}_a$ and $\vec{o}_b$ respectively corresponds to the $\pi$-calculus term $(\nu \vec{o}_a, \vec{o}_b)(\pi(\vec{o}_a), \bar{\vec{o}}(\vec{o}_b), P)$. However, this representation implies the ordering of the outputs, i.e. that $P$ will output $\vec{o}_a$ from $a$ first before it is able to output $\vec{o}_b$ from $b$. A process $P$ with a composite output port, i.e. using Rao et al.’s syntax the process $P.(\pi, \bar{\vec{b}})$, should be able to provide both outputs simultaneously. This is also more appropriate in the context of actual, real-world processes, where all the outputs become available simultaneously or (in different terms) in parallel.

Using our formalisation of the original proofs-as-processes paradigm, we reconstructed Rao et al.’s composite ports in $\pi$-calculus terms so that they represent the output $A \otimes B$. 
Our reconstruction shows that the two output ports \( a \) and \( b \) should be represented by separate \( \pi \)-calculus terms, namely \( (\nu \vec{\alpha}_a) (\overline{\alpha}_a).0 \) and \( (\nu \vec{\alpha}_b) (\overline{\alpha}_b).0 \). Then \( a \) and \( b \) can be put together into a composite port \( z \) as shown in the following \( \pi \)-calculus term:

\[
(\nu a, b) \left( z(a, b) . ((\nu \vec{\alpha}_a) (\overline{\alpha}_a).0) \parallel (\nu \vec{\alpha}_b) (\overline{\alpha}_b).0) \right) \parallel P
\]  

(5.5)

In a similar way, if \( a \) and \( b \) are input ports for process \( P \) in the form \( a (\vec{i}_a).0 \) and \( b (\vec{i}_b).0 \) respectively, their composition into port \( z \) results in the following term:

\[
z(a, b) . \left( a(\vec{i}_a).0 \parallel b(\vec{i}_b).0 \right) \parallel P
\]  

(5.6)

In this case, \( a \) and \( b \) are already bound by \( z \) so no local binding is necessary. Some more explanation as to how these formulas are derived based on the proofs-as-processes paradigm is given in Section 6.3.

In the case of three or more component ports, e.g. \((a, b, c)\), one may be inclined to use larger vectors of size 3 as the arguments for \( z \). However, this kind of process is incompatible with the standard proofs-as-processes patterns. We need to represent the triple composite port using pairs, namely as \((a, (b, c))\). As a result, a triple, output, composite port can be expressed as follows:

\[
(\nu a, \vec{z}) \left( z(a, \vec{z}') . ((\nu \vec{\alpha}_a) (\overline{\alpha}_a).0) \parallel \right.
\]

\[
(\nu b, c) \left( z'(b, c) . ((\nu \vec{\alpha}_b) (\overline{\alpha}_b).0) \parallel \right.
\]

\[
(\nu \vec{\alpha}_c) (\overline{\alpha}_c).0))) \parallel P
\]  

(5.7)

Similarly, the triple, input, composite port can be represented with the following term:

\[
z(a, \vec{z}') . \left( a(\vec{i}_a).0 \parallel \vec{z}'(b, c) . \left( b(\vec{i}_b).0 \parallel c(\vec{i}_c).0 \right) \right) \parallel P
\]  

(5.8)

One important difference in the functionality of Rao et al.’s ports, compared to their \( \pi \)-calculus counterparts introduced here, is the lack of ordering constraints between the execution of the port and \( P \). Using Rao et al.’s syntax, the process \((a, b).P\) will first input through port \((a, b)\) and then execute \( P \). However, it is not possible to express this restriction using the \( \pi \)-calculus. The reason and implications of this fact are further discussed in Section 6.3. This difference casts further doubt on whether Rao et al.’s
process calculus is equivalent to the \( \pi \)-calculus and carries the same properties for the process compositions constructed via LL proofs.

- **Optional ports:** Optional ports are introduced by Rao et al. similarly to composite ports. An optional port \( a + b \) corresponds to a port composed of \( a \) and \( b \) of which only one is selected externally during execution. Rao et al. use optional ports mainly to represent exceptions. A process \( P \) with an optional port that provides an option between two output ports \( a \) and \( b \) is expressed as \( P.(a + b) \). Following our proofs-as-processes based reconstruction, the same process can be expressed in \( \pi \)-calculus terms as follows:

\[
(v \ a, b) \left( z(u, v) \cdot (\overline{a}(a) \cdot (v \ \overline{o}_a)(\overline{a}(\overline{o}_a).0) + v(b) \cdot (v \ \overline{o}_b)(\overline{b}(\overline{o}_b).0) ) \right) \mid \mid P \quad (5.9)
\]

A port \( z \) with an option between two input ports \( a \) and \( b \) is dual to the above:

\[
(v \ u, v) \left( \overline{z}(u, v) \cdot (u(a) \cdot a(\overline{i}_a).0 + v(b) \cdot b(\overline{i}_b).0) \right) \mid \mid P \quad (5.10)
\]

As previously mentioned, a more detailed explanation of how these terms are derived so that they are compatible with proofs-as-processes constructions is given in Section 6.3.

For optional ports with more than two components such as \( a + b + c \), we follow the same approach as in the case of composite ports and break them down into pairs, such as \( a + (b + c) \).

- **Properties:** Given our representations of composite and optional ports we were able to verify the following two properties given by Rao et al.:

  - For composite channels, \( \overline{(a_1,b_1,c_1)}(a_2,b_2,c_2) \) is equivalent to \( \overline{(a_1a_2,b_1b_2,c_1c_2)} \). Empirically, this means that if two processes with composite channels that consist of three output ports \( a_1, b_1, c_1 \) and three input ports \( a_2, b_2, c_2 \) respectively interact with each other, then the result is given by the interaction of \( a_1 \) with \( a_2, b_1 \) with \( b_2 \) and \( c_1 \) with \( c_2 \). According to the \( \pi \)-calculus representations just described, the channel \( \overline{(a_1,b_1,c_1)}(a_2,b_2,c_2) \) corresponds to the the parallel composition of the corresponding \( \pi \)-calculus processes as follows:
\[(\nu a_1, z'_1) (\pi(a_1, z'_1) . ((\nu \vec{o}_a) (\overline{a_1} \langle a_1, z'_1 \rangle).0) ::
(\nu b_1, c_1) \left( \overline{z'_1} (b_1, c_1) . ((\nu \vec{o}_b) (\overline{b_1} \langle b_1 \vec{o}_b \rangle).0) ::
(\nu \vec{o}_c) (\overline{c_1} \langle c_1 \vec{o}_c \rangle).0) \right) \right) ::
|| z (a_2, z'_2) . (a_2 (\vec{i}_a).0 :: z'_2 (b_2, c_2) . (b_2 (\vec{i}_b).0 :: c_2 (\vec{i}_c).0)) \right)
\]

The two processes will interact via channel \(z\), giving the following result:

\[(\nu \vec{o}_a) (\overline{a_1} \langle a_1 \vec{o}_a \rangle).0 ::
(\nu b_1, c_1) \left( \overline{z'_1} (b_1, c_1) . ((\nu \vec{o}_b) (\overline{b_1} \langle b_1 \vec{o}_b \rangle).0) ::
(\nu \vec{o}_c) (\overline{c_1} \langle c_1 \vec{o}_c \rangle).0) \right) ::
|| a_1 (\vec{i}_a).0 :: z'_1 (b_2, c_2) . (b_2 (\vec{i}_b).0 :: c_2 (\vec{i}_c).0) \right)
\]

where \(a_2\) is renamed to \(a_1\) and \(z'_2\) becomes \(z'_1\) based on the reduction rule. In the next interaction, the processes will exchange message \(o_a\) via channel \(a_1\). Noting again that \(a_2\) has been substituted by \(a_1\), this corresponds to Rao et al.’s channel \(\overline{a_1}a_2\). The result then becomes:

\[(\nu b_1, c_1) \left( \overline{z'_1} (b_1, c_1) . ((\nu \vec{o}_b) (\overline{b_1} \langle b_1 \vec{o}_b \rangle).0) ::
(\nu \vec{o}_c) (\overline{c_1} \langle c_1 \vec{o}_c \rangle).0) \right) ::
|| z'_1 (b_2, c_2) . (b_2 (\vec{i}_b).0 :: c_2 (\vec{i}_c).0) \right)
\]

In the next interaction, the processes will communicate through \(z_1\) giving the following result:

\[(\nu \vec{o}_b) (\overline{b_1} \langle b_1 \vec{o}_b \rangle).0 ::
(\nu \vec{o}_c) (\overline{c_1} \langle c_1 \vec{o}_c \rangle).0 ::
|| b_1 (\vec{i}_b).0 :: c_1 (\vec{i}_c).0 \right)
\]

It is now obvious that the next two interactions will happen through \(b_1\) and \(c_1\), with \(b_2\) and \(c_2\) being substituted by their counterparts \(b_1\) and \(c_1\) respectively. These two interactions therefore form channels \(\overline{b_1}b_2\) and \(\overline{c_1}c_2\).
• For optional channels, \((a_1 + b_1)(a_2 + b_2)\) is equivalent to \(\bar{a}_1a_2 + \bar{b}_1b_2\). Empirically, this means that if two processes with optional channels \((a_1 + b_1)\) and \((a_2 + b_2)\) respectively communicate with each other the result will be either a communication between \(\bar{a}_1\) and \(a_2\) or between \(\bar{b}_1\) and \(b_2\). According to the \(\pi\)-calculus representation discussed earlier in this section, the channel \((a_1 + b_1)(a_2 + b_2)\) corresponds to the following parallel composition:

\[
\begin{align*}
&\left(\nu a_1, b_1\right)\left(z(u, v) \cdot \left(\bar{a}_1a_1\right) \cdot \left(\nu \bar{o}_a\left(\bar{a}_1\langle \bar{o}_a \rangle.0\right) + \nu \bar{o}_b\left(\bar{b}_1\langle \bar{o}_b \rangle.0\right)\right)\right) \\
&\mid | \\
&\left(\nu u, v\right)\left(z(u, v) \cdot u(a_2) . a_2\left(i_a\right).0 + v(b_2) . b_2\left(i_b\right).0\right)
\end{align*}
\]

The interaction through channel \(z\) gives the following result:

\[
\bar{a}_1\cdot \left(\nu \bar{o}_a\left(\bar{a}_1\langle \bar{o}_a \rangle.0\right) + \nu \bar{o}_b\left(\bar{b}_1\langle \bar{o}_b \rangle.0\right)\right) \\
\mid | \\
\nu (a_2) . a_2\left(i_a\right).0 + \nu (b_2) . b_2\left(i_b\right).0
\]

There are now two possible reductions, either through \(u\) or \(v\), and the choice is non-deterministic. If the interaction is through \(u\) then we obtain the following result, where \(a_2\) is substituted by \(a_1\):

\[
\left(\nu \bar{o}_a\left(\bar{a}_1\langle \bar{o}_a \rangle.0\right)\right) \mid | a_1\left(i_a\right).0
\]

If it is through channel \(v\) instead, \(b_2\) is substituted by \(b_1\) and we obtain:

\[
\left(\nu \bar{o}_b\left(\bar{b}_1\langle \bar{o}_b \rangle.0\right)\right) \mid | b_1\left(i_b\right).0
\]

Therefore, it is clear that we will either result in a communication between \(a_1\) and \(a_2\) or between \(b_1\) and \(b_2\). This justifies Rao et al.’s representation \(\bar{a}_1a_2 + \bar{b}_1b_2\).

However, it is worth noting that the + operator in this representation is not the same as the + operator in \(\pi\)-calculus, but rather corresponds to the non-deterministic choice in the \(\pi\)-calculus reduction rules. Similarly to the case of the sequence operator (see Section \(5.3.1\)), \(\pi\)-calculus reduction rules and substitutions are collapsed within simple syntax, a process which, however, is prone to errors and misinterpretations.
Even though we managed to find a justification for Rao et al.’s introduced concepts, our π-calculus interpretations show that these are still not straightforward syntax shortcuts. There is a lot of implicit information that has been withheld in the original papers by Rao et al. In order to formally verify the correspondence we presented, we would need a much more detailed account of the semantics of Rao et al.’s process calculus, and more specifically a formal description of its reduction rules.

Moreover, Rao et al. claim these concepts can be viewed as syntactic abbreviations. However, as we saw, the π-calculus representation of (for example) composite ports does not merely involve a combination of ports but the entire description of the involved processes and the possible reductions between them, whereas the composite port itself is merely represented as a single name, $z$. Additionally, imposing ordering constraints between, for example, a composite input and the rest of the process, using π-calculus is, to the best of our knowledge, not possible. Therefore, we consider Rao et al.’s concepts to be syntactic oversimplifications.

5.4 Issues with the Proof to Process Correspondence

In our attempt to formalise the work done by Rao et al., a significant amount of time was spent analysing and understanding both the proofs-as-processes paradigm and the way it has been implemented in their work (see Section 5.2). Our investigation brought to light some significant differences between the theory and the actual usage made by Rao et al. in a manner reminiscent of the differences that we highlighted between standard π-calculus and Rao et al.’s formulation in Section 5.3. Our effort was then to check whether these differences were not only reasonably justified but also sound with respect to the logical theoretical backround.

In general, we were unable to find evidence that these changes from the original theory are sound. In particular, we were interested on whether or not the property that “applying the possible reductions of a π-calculus term constructed in a proof corresponds to the process of cut-elimination in that proof” (Bellin and Scott, 1994) still holds. Bellin and Scott prove this property and the soundness and correctness of their translation using complicated proof theory for CLL involving cut-elimination proofs in proof nets (Girard, 1995a). Therefore, reconstructing such proofs for the case of Rao et al. requires some significant effort. Notably, the guarantees of correctness and the
properties of type safety and deadlock freedom of any process constructed via proof are direct consequences of these proofs.

Our analysis showed that the main differences in the implementation of Rao et al. compared to the original theory by Abramsky, Bellin, and Scott include some inconsistencies in the use of proof terms, the usage of ILL in a two-sided sequent calculus (as opposed to CLL in a one-sided sequent calculus) and an incomplete set of inference rules, and a notable change in the identity axiom translation. We analyse each of these differences and their impact next.

5.4.1 Inconsistencies in Proofs Terms

As mentioned in the previous section, Bellin and Scott attach names as proof terms to CLL formulas based on standard type theory practices and as a parallel to the Curry-Howard isomorphism. Each of these names corresponds to a process calculus name, i.e., a channel or port. For example, a proof term $x$ attached to a formula $A \otimes B$ is represented as $x : A \otimes B$ and describes the fact that the name $x$ in the corresponding $\pi$-calculus process has type $A \otimes B$. In addition, the $\pi$-calculus translation of the proof derivation of a CLL judgement is attached to the turnstile $\vdash$ of the judgement using the :: operator. For example, the proof of the judgement $\vdash P :: x : A \otimes B$ is corresponds to the $\pi$-calculus process $P$. With regard to Rao et al.’s formulation, a close look at Figures 5.6 and 5.7 shows multiple differences from the traditional type theory annotations. Weenumerate some of the most noticeable and problematic differences next:

1. The first thing is the different syntax, where the term $x: A \otimes B$ is represented as $A \otimes B (x)$.

2. Secondly, in some cases Rao et al. choose to annotate subterms of formulas. This is particularly noticeable in the congruence rules of Figure 5.7. For example in the term $A (a) \otimes B (b)$, subterms $A$ and $B$ have been individually annotated, as opposed to the annotation of the entire formula $A \otimes B$. This may have some intuitive interpretation when presented on paper, but introduces significant problems in any attempt to formalise these terms.

Indeed, annotating subterms in a CLL formula introduces a deeper connection between proof terms and logical terms. Originally in the proofs-as-processes paradigm (and in any type system for that matter) the proof term annotations can
be erased completely from the proofs without affecting the logical validity of the proof or the proof tree itself in any way. The introduction of subterm annotations, as well as congruence rules that manipulate those, reveals a deeper connection between the proof terms and the logic, as reasoning is now performed not just on ILL terms but on annotated terms, and proof annotations sometimes become an inseparable part of the logic.

For instance, it appears that $A \otimes B(a, b)$ and $A(a) \otimes B(b)$ are both valid terms in Rao et al.’s formulation. This means that the connective $\otimes$ connects both pure ILL terms and ILL terms annotated with proof terms (and possibly a combination of both). The only way to formalise such terms would be to introduce two distinct $\otimes$ operators, one for pure ILL terms and one for annotated ones. However this, in turn, would require the introduction of two sets of $\otimes$ inference rules, one for each connective. The complexity increases unnecessarily.

3. Additionally, the type of proof terms used to annotate the different ILL terms is not uniformly the same. In the proofs-as-processes paradigm, all proof terms are $\pi$-calculus names, and all proof correspondences are $\pi$-calculus processes. This is not the case in the formulation of Rao et al. When considering the process calculus introduced in Figure 5.5, it is not clear which of these introduced types is the type of the proof annotations being used. This is important information for one to be able to formalise these proof terms, but based on the wildly inconsistent annotations, we were not able to infer it.

For example, the $A$ formula in the $(R \otimes)$ rule has a (possibly) simple port $a$ attached to it as a proof term, whereas $A \otimes B$ is annotated with a composite port $(a, b)$:

$$
\Gamma \vdash P :: A(a) \quad \Gamma' \vdash Q :: B(b) \\
\Gamma, \Gamma' \vdash (P || Q) :: A \otimes B(a, b) \quad (R \otimes)
$$

Based on the $\pi$-calculus counterparts for composite ports we presented in Section 5.3.3 this may be justifiable. However, it erroneously gives the impression that part of the process calculus term is constructed within the proof terms. In our example, the composite port $(a, b)$ corresponds to the following $\pi$-calculus process:

$$
(v \ a, b) \ (\bar{z}(a, b). \ ((v \ x) \ (\bar{a}(\bar{x}).0) \ || \ (v \ u) \ (\bar{b}(\bar{u}).0)) )
$$

(5.11)
Therefore, the impression is made that the entire process is attached as a proof term to $A \otimes B$. However, looking at the corresponding $\otimes$ rule from the original proofs-as-processes paradigm in Figure 4.3, it is merely the port $z$ from process (5.11) that is attached to $A \otimes B$.

A bigger issue is raised by the annotation of linear implication. Let us review the $\Gamma, A(a) \vdash P :: B(b) \quad \Gamma \vdash A(a) \rightarrow a.p_{\overline{b}} B(b) \quad R \rightarrow$ 

In this we notice two things. Firstly, the term $A \rightarrow B$, apart from having its subterms $A$ and $B$ annotated, is itself annotated with a process rather than a (simple, composite, or any type of) port. Rao et al. use linear implication to represent the behaviour of the process that is attached to it. This is a large contradiction to the rest of the approach where the behaviour of a process is described using an entire judgement, which incidently includes but is not limited to an implication (see (5.1)). Secondly, the judgement in the conclusion of the rule is not annotated with a process, i.e. does not have a defined correspondence with the process calculus. Apart from a significant obstacle to the formalisation of this correspondence (it is not at all clear how to handle linear implication in such a formalisation), this also introduces a major inconsistency. Namely, it is not at all clear what the process calculus correspondence would be for a proof that involves the $\Gamma, A(a) \vdash P :: B(b) \quad \Gamma \vdash A(a) \rightarrow a.p_{\overline{b}} B(b) \quad R \rightarrow$

Assume we apply the $\Gamma, A(a) \vdash P :: B(b) \quad \Gamma \vdash A(a) \rightarrow a.p_{\overline{b}} B(b) \quad R \rightarrow$ rule in a forward direction to the conclusion of the above, which would by all means be a valid ILL proof step. This would result in the following:
\[ \vdash X \rightarrow_o (Y \rightarrow_o Z) \]

Using Rao et al.’s proof system, this would result in the following:

\[ \vdash X(x) \rightarrow_o S(Y(y) \rightarrow_o y.\bar{Q}.z) \]

where $S$ is the translation of this proof to the process calculus. According to the $R \rightarrow_o$ inference rule, $S$ must be of the form $a.P.b$. However, $b$ in this case is not a port, but corresponds to the processes $y.Q.\bar{z}$. In addition, it is completely unclear what $P$ would be in this case.

To summarize, proof term annotations in the formulation of Rao et al. not only differ from standard type theory style annotations in the syntax, but are also inconsistent with respect to both the type of terms being annotated (both ILL terms and their subterms can be annotated) and the types of the annotations themselves (we see examples of simple ports, composite ports, and processes being used as proof terms in different cases, as well as cases where the annotation is undetermined). These inconsistencies were immediately revealed in our attempt to formalise the correspondence of ILL to Rao et al.’s process calculus within HOL Light, which proved to be impossible even at the syntactic level.

### 5.4.2 Two-sided Sequent Calculus and ILL

Rao et al. use ILL instead of CLL for his translations. The reasoning behind this change is unclear. It appears that the use of lolli ($\rightarrow_o$) provides a somewhat more intuitive representation and interpretation of web services (even though, as such, it introduces proof term inconsistencies as we saw in the previous section). In principle, switching from CLL to ILL should not present significant challenges since the two logics share their core properties. There are, however, some issues that need to be addressed, and we found this not to be the case in the formulation of Rao et al.

Primarily, using a two-sided sequent calculus is also a significant alteration to the original theory. In particular, cut-elimination proofs differ between one-sided and two-sided representations, therefore casting further doubt on whether Rao et al.’s translation can be proven to be sound.
Chapter 5. Formal Analysis of Rao et al.’s Approach

The duality of the connectives is also lost and replaced by the duality of left and right hand side rules. Essentially, the ILL connectives, unlike the CLL ones, do not have a polarity, and the \( \otimes \) and \( \oplus \) connectives can be used to represent both inputs and outputs. Resources found in the left hand side of the turnstile (\( \vdash \)) are inputs, whereas those found in the right hand side are outputs. This, however, raises an interesting question as to the interpretation (and more formally the process calculus translation) of the \& and \( \rightarrow \) connectives (note that the \( \exists \) connective is not used in ILL).

In CLL the \textit{plus} (\( \oplus \)) and \textit{with} (\( \& \)) connectives are dual and we have translated \textit{plus} as the choice between two outputs and \textit{with} as the choice between two inputs. In ILL the right hand side rules for \textit{plus} provide optional outputs while the left hand side rule corresponds to the optional input. Naturally, the question arises: what is \textit{with} in ILL? Rao et al. refer to it as internal choice (as opposed to external choice for \textit{plus}). However, the inference rules from Figure 5.6 do not seem to clarify this distinction. The \& rules make use of composite ports, which we so far have treated as ports for parallel input and output, as well as parallel composition of processes (as opposed to a choice between them). In fact the right hand side rule \( R \& \) is very similar to the \( R \otimes \) rule which involves parallel composition and bears no relation to choice. The left hand side rules \( L \& (a) \) and \( L \& (b) \) make use of a 0 port. The semantics of this port are never given in any of the papers, neither formally nor intuitively, and it does not even appear in the syntax specification of the process calculus (see Figure 5.5). The interpretation and translation of such a port is therefore completely unknown. We believe these rules have neither been explained nor logically justified sufficiently, and moreover, are not in any way connected to the original proofs-as-processes paradigm. We consider reconstructing the \( \pi \)-calculus translation of the \& connective in ILL an interesting but also challenging and deeply proof theoretic problem.

As previously mentioned, Rao et al. make use of the \( \rightarrow \) connective to specify the behaviour of web services. However, once more, the inference rules do not provide sufficient information to fully reconstruct their process interpretation. The right hand side rule \( R \rightarrow \) seems intuitive at first, so that the left hand side of the connective is an input resource, whereas the right hand side is an output resource. For example, the term \( A \rightarrow B \) represents a process with input \( A \) and output \( B \). As with the case of proof terms described in the previous section, the issue arises in the case of nested applications of \( \rightarrow \). For example, what would the interpretation of the term \( A \rightarrow (B \rightarrow C) \) be? An even greater issue arises by the fact that Rao et al. have completely omitted the process
5.4. Issues with the Proof to Process Correspondence

calculus translation for the left hand side \( L \rightarrow \) rule. They have used this rule to derive the introduced \textit{Shift} rule. However, the \textit{Shift} rule does not consider cases where implication appears on the left hand side of the turnstile. Based on this, there are valid ILL proofs that can not be translated to Rao et al.’s process calculus.

It is worth noting that our argument here is not that process calculus interpretations of the \& and \( \rightarrow \) connectives in ILL do not exist. In fact, process-based translations of ILL have been developed by Caires et al. using session types (see Section\ref{sec:session}). We merely observe that these issues have not been addressed sufficiently by Rao et al. in any of their published papers. In conjunction with the fact that the source code for their system was never made available to the public, this prohibited any effort to formalise and mechanize their work and apply the fully formal, rigorous inference mechanisms provided by HOL Light, which would guarantee the correctness of the constructed process compositions.

5.4.3 The Identity Axiom

In this section we take a closer look at the translation of the identity axiom as given by Rao et al.:

\[
\frac{}{\Gamma \vdash (\nu x)0::A(x)} \text{Id}
\]

This has an important difference from the translation given by Bellin and Scott, which, as seen in Figure \ref{fig:identity}, is the following:

\[
\Gamma \vdash x(a).y(a).0::y:A, x:A^\perp
\]

Firstly, in Rao et al.’s rule both \( A \) and \( A^\perp \) have the same proof term. In \( \pi \)-calculus terms, this would translate into \( x(a).\bar{x}(a).0 \) instead of the Bellin and Scott’s axiom buffer \( x(a).\bar{y}(a).0 \). Rao et al.’s version introduces binding and renaming problems in the constructed process (even in the simple Ski example of Section\ref{sec:ski}).

We will not delve into the details, but the general idea is that in Rao et al.’s case, binding or renaming the input channel also binds or renames the output channel. This makes it hard to interface the corresponding process externally. For example, cutting it with another, receiving process will bind its output \( \bar{x} \) locally (see the explanation of the \textit{Cut} rule in Section\ref{sec:cut}). As a result its input \( x \) is also bound locally, thus making it impossible to access it externally. The Bellin and Scott version uses a different name \( y \)
for the input, thus keeping it independent from $x$ and free in the case where $x$ is bound locally.

In addition, Rao et al. translate the identity axiom to the null process $0$ of their process calculus. The original axiom buffer $I_{x,y}$ performs an important task of buffering information from one port to another. It also plays an important role in the handling of complex inputs and outputs (see Chapter 7). Its exact form $x(a).y(a).0$ is a product the of a deep theoretical investigation. Without delving into the proof theoretical details, Abramsky also considered other possible forms, such as a bidirectional buffers $(x(a).y(a).0 + y(a).x(a).0)$, and Bellin and Scott’s decision on this particular form facilitated their soundness and completeness proofs [Bellin and Scott 1994].

Rao et al. collapse all this effort and oversimplify the axiom buffer to a null process, since their formulation uses the same port $x$ for input and output (i.e. no “buffering” occurs). This creates serious doubts about the soundness and correctness of their correspondence, and consequently, the validity of their claims for verified web service compositions.

5.5 Structural Congruence Rules

In Section 3.2, we mentioned the set of rules in the $\pi$-calculus that define structural congruence relations between processes. In [Rao 2004], a set of congruence rules is introduced, that “consider both the LL and the process calculus” (see Figure 5.7). Two examples of such rules that we will be analysing further are the commutative and associative laws of the tensor ($\otimes$). The commutative law is defined as:

$$A \otimes B(a,b) \equiv B \otimes A(b,a)$$

whereas the associative law is defined as follows:

$$(A(a) \otimes B(b)) \otimes C(c) \equiv A(a) \otimes (B(b) \otimes C(c)) \equiv A \otimes B \otimes C(a,b,c)$$

No semantics are provided for these rules. Intuitively, the commutative law describes the fact that a process with a composite output port $(a,b)$ can reverse the involved ports, thus obtaining $(b,a)$. Similarly, the associative law provides an associativity property when there are more than two components in the same composite port. It is worth
noting that, in the associative law, propositions have been annotated individually. As we described in Section 5.4.1, this is inconsistent with respect to the usual type theory and also creates a generally unwanted syntactic connection between the logic and process calculus. For example, in the proof of the Ski example in Figure 5.8, the use of these congruence rules is presented as an inference step in the proof tree. Therefore, the inference being performed for the construction of web services composition does not only involve the logical terms, but also the process calculus terms. Consequently, it is not entirely possible to erase the process calculus terms from the proof without affecting the validity of the proof and the structure of the proof tree.

From the point of view of LL, these rules correspond to derivable properties of the logic. Both \( A \otimes B \vdash B \otimes A \) and \( (A \otimes B) \otimes C \vdash A \otimes (B \otimes C) \) are derivable from the standard rules of both ILL and CLL. However, Rao et al.’s congruence rules also involve transformations in the process calculus translation. Even though they seem simple and intuitive, no guarantees of correctness for these transformations are given. Additionally, since Rao et al. do not provide formally defined semantics for composite ports, it seems impossible to give a direct formalisation for these rules.

We note that since Bellin and Scott use standard proof annotations it is impossible to express Rao et al.’s rules in the form presented here within our proofs-as-processes formalisation. However, using our embedding it is possible to construct the \( \pi \)-calculus translations of the proofs used to formally derive the corresponding CLL properties. We present the results from this effort are presented in detail in the next sections.

These translations are actually \( \pi \)-calculus processes that rearrange the type of a port. For example, the process corresponding to the property of commutativity of \( \otimes \), namely \( A \otimes B \vdash B \otimes A \), rearranges a port that has type \( A \otimes B \) into a port of type \( B \otimes A \). This can be viewed as taking the components, in this example \( A \) and \( B \), of a composite type apart and putting them back together to form another composite type. The actual information being carried is not affected, but it is merely restructured to fit the specification of another type. Based on this description we call this kind of processes filters. Basically, the \( \pi \)-calculus translation of any CLL lemma that has no (meta-level) assumptions, is a filter.
5.5.1 The Commutative Law

The commutativity property of $\otimes$ in our annotated CLL using one-sided sequent calculus can be expressed as follows:

$$\vdash P :: x : (A \otimes B) \perp, \ y : (B \otimes A) \tag{5.12}$$

The question that we seek to answer is which process calculus term $P$ corresponds to this inference based on the proofs-as-processes paradigm.

Hiding the process calculus annotations for simplicity, the proof of the commutative law is as follows:

$$\begin{array}{c}
\vdash A \perp, B \perp \text{Id} \\
\vdash A \perp, B \perp, B \otimes A \\
\vdash (A \otimes B) \perp, B \otimes A
\end{array} \tag{5.13}$$

This proof can easily be performed within our embedded version of CLL as shown in Section 4.4.5. From the proof we can extract the instantiation for $P$:

$$P = \{ x : (i_a, i_b), (\otimes (i_b, o_b, I_i b, o_a)) \} \tag{5.14}$$

Comparing our extracted term to the version of the commutative law given by Rao et al. (see Figure 5.7) seems to give a clear indication of oversimplification on their part. The interactions of this term with other processes (see Appendix C), leads to a consistent, well-defined process that swaps the order of two arguments of type $A$ and $B$ respectively. This required us to introduce specialised lemmas that correspond to $\pi$-calculus processes with such functionality of manipulating composite types (see Section 5.5.3). In the case of Rao et al., the structure and interactions of these processes are reduced to the swap of composite port $(a, b)$ to $(b, a)$. Given the underspecified semantics of composite ports, the validity of such a swap is not verified. In fact, analysing the behaviour of $P$ gave us insight into the (formally) expected behaviour of Rao et al.’s composite ports and their $\pi$-calculus translation we presented in Section 5.3.3.
5.5.2 The Associative Law

Observations similar to those about the commutative law apply in the case of the associative law. In order to formally verify the associativity law, we followed the exact same process as for the commutative law.

The associativity property of $\otimes$ in our annotated CLL can be expressed as follows:

$$\vdash Q:: x:(A \otimes (B \otimes C))^\perp, y:((A \otimes B) \otimes C)$$  \hspace{1cm} (5.15)

Hiding the process calculus annotations for simplicity, the proof of this property is as follows:

$$\vdash A, A \perp Id \quad \vdash B, B \perp Id \quad \vdash C, C \perp Id$$

$$\vdash A, B, A \otimes B \quad \vdash C, C \perp Id$$

$$\vdash A, B, C, (A \otimes B) \otimes C$$

Reconstructing this proof within our embedded proofs-as-processes system allows us to extract in a fully rigorous fashion the appropriate $\pi$-calculus instantiation for process $Q$ that corresponds to the associativity property, as follows:

$$Q = \Pi_{x,y} \left( \Pi_{y'} \left( \Pi_{y''} \left( i_a, i_c \otimes (i_a, i_c) \right) \right) \right)$$

The complexity of this term further emphasizes the oversimplified versions of the structural congruence rules given by Rao et al.

5.5.3 Filters

As mentioned previously, from the point of view CLL, the commutative law of $\otimes$ as shown in (5.12) describes a process that swaps the order of two outputs $A$ and $B$. More generally, the proofs of such properties of the CLL connectives (e.g. commutativity,
associativity, distributivity, etc.) correspond to standalone, well-defined $\pi$-calculus processes that perform some transformation in the complex CLL type of an input or output. We call such processes filters.

These can be particularly useful when the output of a process cannot exactly match the input of another, unless an allowed (by the properties of CLL) transformation is performed. For example, a process with output $X \otimes Y$ can only communicate (via the Cut rule) with a process with input $(Y \otimes X)^\perp$ (see Section 6.2 for more details on input and output specifications using CLL) if we interleave their communication with the filter that corresponds to the commutativity law of $\otimes$.

Focusing on this particular law as an example, we note that lemma (5.12) is not usable directly for two main reasons: firstly because $A$ and $B$ are assumed to be atomic in this case and secondly because we are interested in the communication between the corresponding filter with other processes. We address these issues individually next.

### 5.5.3.1 Non-atomicity in Filters

The axiom buffer $\vdash A, A^\perp$ assumes $A$ is an atomic proposition and, therefore, a filter consisting of axiom buffers is unable to handle cases of more complicated types such as $(X \oplus Y) \otimes Z$. This demonstrated with an example in Section 7.3.

For this reason, we cannot close the branches of proof (5.13) with the identity axiom and, instead, we leave them open. This allows the proof to be completed differently for every case, so that the appropriate composite buffers that can handle each complex case individually are constructed, for example by using the BUFFER,TAC tactic presented in Section 7.3.1.

The open branches of the proof are represented as two schematic assumptions, namely $\vdash P_a :: o_a : A, \ i_a : A^\perp$ and $\vdash P_b :: o_b : B, \ i_b : B^\perp$, in the following lemma definition of the filter corresponding to the commutativity law of $\otimes$:

\[
\vdash \text{TimesComm}_x^\otimes(P_a, P_b) :: x : (A \otimes B)^\perp, \ y : (B \otimes A)
\]  

(5.17)

where $\text{TimesComm}_x^\otimes(P_a, P_b)$ is the filter as a $\pi$-calculus process.

In other words, we assume the existence of appropriate processes (non-atomic buffers) $P_a$ and $P_b$ that can handle the buffering of $A$ and $B$ respectively in any case. The proof
5.5. Structural Congruence Rules

of lemma (5.17) omitting the process calculus annotations is as follows:

\[
\begin{align*}
\vdash B, B \perp & \quad Ax \\
\vdash A, A \perp & \quad Ax \\
\vdash A \perp, B \perp, B \otimes A & \quad \otimes \\
\vdash (A \otimes B) \perp, B \otimes A & \quad \otimes
\end{align*}
\]

(5.18)

Using the proofs-as-processes paradigm, we extract the following \(\pi\)-calculus term for the filter:

\[
\begin{aligned}
\text{TimesComm}^\pi(P_a, P_b) &= \bigotimes_x \bigotimes_y (P_b, P_a) \\
&= x(i_a, i_b) \cdot (\nu o_b, o_a)(y(a, o_a) \cdot (P_b \parallel P_a))
\end{aligned}
\]

(5.19)

\(\text{TimesComm}^\pi(P_a, P_b)\) is expected to reflect the commutativity property of the tensor \(\otimes\) as a process, which, as previously mentioned, essentially corresponds to swapping the order of the two arguments of a composite output. Using the \(\pi\)-calculus representation, \(\text{TimesComm}^\pi(P_a, P_b)\) is a process that does exactly that. It receives a composite port \(x\) with arguments \(i_a\) and \(i_b\) and outputs a composite port \(y\) with arguments \(o_b\) and \(o_a\). Moreover, the input \(a\) from \(i_a\) is given as output through \(o_a\) using process \(P_a\) and the input \(b\) of \(i_b\) through to \(o_b\) using process \(P_b\). Note that in \(\text{TimesComm}^\pi(P_a, P_b)\), \(x\) and \(y\) are free names as dictated by the proofs-as-processes paradigm (since \(x\) and \(y\) are the only proof terms in the CLL judgement). Also note that \(y\) has its arguments reversed compared to \(x\) which effectively accomplishes the expected swapping behaviour.

In the case where \(A\) and \(B\) are atomic propositions, we can use axiom buffers to eliminate the assumptions. In short, the following will hold:

\[
\begin{align*}
P_a &= I_{i_a, o_a} = i_a(a) \cdot \overline{o_a}(a) \cdot 0 \\
P_b &= I_{i_b, o_b} = i_b(a) \cdot \overline{o_b}(a) \cdot 0
\end{align*}
\]

\(\text{TimesComm}^\pi(P_a, P_b) = x(i_a, i_b) \cdot (\nu o_b, o_a)(y(a, o_a) \cdot (i_b(b) \cdot \overline{o_b}(b) \cdot 0 \parallel i_a(a) \cdot \overline{o_a}(a) \cdot 0))\)

(5.20)

5.5.3.2 Filter Application via Proof

The commutativity filter as described using (5.17) is not very useful in a proof. Even from the computational point of view, the process \(\text{TimesComm}^\pi(P_a, P_b)\) is not useful
Chapter 5. Formal Analysis of Rao et al.’s Approach

on its own unless it interacts with another process $P$ that has a composite output port. We expect the result to be a new, composite process $Q$ that has the output of $P$ with its arguments swapped. A more useful rule would therefore be one that matches a particular process with a composite port and swaps its arguments using $\text{TimesComm}_x^\pi(P_a, P_b)$. This rule can be expressed as follows:

\[
\frac{\vdash \Gamma, A \otimes B}{\vdash \Gamma, B \otimes A}
\]

Using proofs-as-processes annotations and the $P_a$ and $P_b$ assumptions as before, the rule obtains the following form:

\[
\frac{\vdash P :: \vec{w} : \Gamma, a :: (A \otimes B) \quad \vdash P_a :: o_a : A, i_a : A^\perp \quad \vdash P_b :: o_b : B, i_b : B^\perp}{\vdash Q :: \vec{w} : \Gamma, b :: (B \otimes A)}
\] (5.21)

In this case we are interested in discovering the process $Q$ that corresponds to process $P$ with its arguments swapped. The proof involves the usage of the $\text{Cut}$ rule with (5.17). At the end of the proof, instantiating $Q$ gives us the following $\pi$-calculus term:

\[
Q = \text{Cut}^z (P, \text{TimesComm}_x^\pi(P_a, P_b)) = (\nu z) (P[a/z] \parallel \text{TimesComm}_x^\pi(P_a, P_b))
\] (5.22)

The interaction between process $P$ and filter $\text{TimesComm}_x^\pi(P_a, P_b)$ modelled in term (5.22) will result in the desired swapping. For the interested reader, we demonstrate this in Appendix C.

It is worth noting that other filters corresponding to associativity of $\otimes$ and other properties of the CLL connectives can easily be constructed using the same approach.

5.6 Related Work

The work of Rao et al. was an important step towards pragmatic web service composition using Linear Logic theorem proving. Although the underlying theory has evolved in the recent years (see Section 4.5.2), its practical application has not developed in the same rate.

To our knowledge, the only known approach to process composition based on the proofs-as-processes paradigm after that of Rao et al. has been proposed recently by
5.6. Related Work

Zhao. Heavily inspired by the work of Rao et al., Zhao developed an ILL based methodology for the composition of RESTful web services (Zhao, 2013). The main differences from the original approach are the use of Coq as a theorem prover to embed proofs-as-processes and perform the proofs, the adaptation of the methodology to RESTful services (as opposed to DAML-S), and the identification of two distinct composition phases.

Having a closer look at their formalisation, and given the experience from the analysis of the work of Rao et al. as presented in this chapter, one can immediately identify the inconsistencies that Zhao’s work partially inherits from Rao et al. More specifically, Zhao uses a process calculus which is much closer to the \( \pi \)-calculus than the calculus used in Rao et al., but still contains an added sequence operator (\( . \)) so that \( P.Q \) is interpreted as \( \text{execute } P \text{ then } Q \). Combined with the use of ILL (instead of CLL) without a formal mapping to the original proofs-as-processes paradigm, this raises doubts with regards to the consistency of the theory.

Even though the use of a rigorous framework such as Coq has led to a much more formal correspondence between ILL and Zhao’s extended \( \pi \)-calculus, some missteps can still be observed. Without analysing all the inference rules in detail, we can demonstrate one of the hidden inconsistencies using the following 2 rules as presented by Zhao:

\[
\Gamma \vdash P :: y : A \rightarrow G \\
\Gamma \vdash x : A \vdash P :: y : G \\
\text{Shift}
\]

\[
\Gamma \vdash y(x). P :: y : A \rightarrow G \\
\Gamma \vdash x : A \vdash P :: y : B \\
\rightarrow R
\]

Based on these rules and assuming \( \Gamma \vdash P :: y : A \rightarrow G \) we can produce the following proof:

\[
\Gamma \vdash P :: y : A \rightarrow G \\
\Gamma \vdash x : A \vdash P :: y : G \\
\text{Shift} \\
\Gamma \vdash y(x). P :: y : A \rightarrow G \\
\rightarrow R
\]

The above is in fact an alternative to a simple proof by assumption:

\[
\Gamma \vdash P :: y : A \rightarrow G \\
\text{Ax}
\]
Since the 2 proofs are equivalent without cut elimination, the 2 corresponding process calculus terms must be equivalent (based on the $\pi$-calculus congruence relations), i.e. $P \equiv y(x).P$, which is false.

This further demonstrates the danger of arbitrarily translating Linear Logic proofs as processes without a proper theoretical investigation with the associated proofs of soundness and correctness, such as the one Bellin and Scott provide for the original proofs-as-processes paradigm. Performing a formal analysis of the process calculus correspondence is essential in order to be able to guarantee the properties provided by Linear Logic (including resource accounting, type correctness, and deadlock-freedom).

To the best of our knowledge, there have not been any other attempts to analyse these proofs-as-processes based correspondences. Such discrepancies may then go by unnoticed, even when working within a formal, theorem proving framework, and compromise the validity of the claimed benefits.

### 5.7 Conclusion

In their work, Rao et al. introduced an interesting methodology for logic-based DAML-S web services composition using ILL theorem proving. It involves three simple steps, namely the translation of process specifications into ILL, the compositions of these processes towards a requested process via ILL proof, and the extraction of the resulting process described in terms of their process calculus.

Despite their connections to the proofs-as-processes theory and the $\pi$-calculus, their focus was towards better accommodation of the particular features of web services and making the process calculus specifications match the syntax of DAML-S more closely. Our attempt at formalising their work revealed that this focus was achieved at the expense of consistency and resulted in oversimplifications when compared to the original proofs-as-processes theory. Although some of the introduced concepts can be translated into valid, proofs-as-processes based $\pi$-calculus terms, others were underspecifications or overabstractions that could not be expressed within the formalised theory. These issues casted doubt on their claimed properties of soundness and correctness of the constructed process composition. Without clearly defined formal semantics, the task of verifying such claims proved impossible.
There are multiple contributions from this analysis. It highlights the importance of a proper application of the theory and of the use of formal methods to verify the properties of a system. Introducing a semi-formal or fully formal syntax does not necessarily guarantee the correctness of the constructed specifications and of any reasoning performed on them. By contrast, in our work formal methods were utilised effectively to analyse and verify the consistency and degree of rigor of the methodology.

More importantly, these results led us to reform the structure of our project. As a result, we investigated the theoretical roots of the work of Rao et al., namely the proofs-as-processes paradigm, directly and reconstructed their methodology using the original theory in its embedded form in HOL Light (see Chapter 4). Despite the observed problems in the implementation, Rao et al.’s general strategy for constructing formally verified process compositions is a valid approach and can guarantee a number of properties for the resulting composite processes.

In the next chapter, we begin the description of our take on Rao et al.’s approach based on the original proof-as-processes paradigm by introducing process specifications in CLL and the original, polyadic \( \pi \)-calculus. Our aim is to demonstrate the validity and usefulness of this approach, as well as the guaranteed properties of the end result, not only in toy examples, but also in more complicated, real-world problems.
Chapter 6

Process Specification

In this chapter, we present the means of specifying abstract process models in the context of performing composition via proof using the proofs-as-processes paradigm. In particular, we focus on specifications using logic, process calculus terms, and a diagrammatic visualisation.

6.1 Introduction

Our approach is to construct CLL specifications of abstract processes based on their inputs, outputs, preconditions, and effects. Since we are interested in process composition, we focus on describing the ways that a process interacts with other processes, rather than its specific functionality and control flow. More specifically, we focus on the types of inputs, outputs, preconditions, and effects (IOPEs) of the process as well as the channels used to communicate those.

All IOPEs are treated as resources of a specific type communicated via a π-calculus channel, either incoming or outgoing. There is no distinction made between resources describing information or objects (inputs/outputs) or changes to the state (preconditions/effects). Therefore, to avoid repetition, unless otherwise stated, we will refer to both inputs and preconditions as inputs and both outputs and effects as outputs. Unless a clear distinction is needed, exceptions are also treated as outputs.

Each of the introduced IOPEs in our system is specified as an abstract type. This is particularly important when translating process specifications to code, where the types
of IOPEs are translated to abstract Scala types (see Section 8.2.1). Essentially, this means these types can be instantiated to any concrete datatype, from simple types to complicated objects, including references to relational database tables and ontological classes.

In Section 6.2, we analyse the CLL specification of a process, which basically corresponds to the type specification of its IOPEs. Next, in Section 6.3, we discuss generated, proofs-as-processes styled specifications of atomic processes using the π-calculus. A description of a diagrammatic visualisation tool aimed at hiding the complex CLL and π-calculus specifications from the non-experts can be found in Section 6.4. We conclude this chapter with a quick breakdown of OWL-based ontologies and the potential use in our system in Section 6.5.

### 6.2 Type Specifications Using CLL

In Section 4.2, we gave an intuitive interpretation of the CLL connectives in the context of resources. This interpretation makes specifying processes in CLL a fairly straightforward task. Inputs can be represented using negated literals, or their combinations with input connectives (\& and \(\triangledown\)), whereas outputs and exceptions can be represented using positive literals, or their combinations with output connectives (⊗ and ⊕).

However, our chosen set of inference rules and our aim of having an intuitive process representation that allows for some level of proof automation, causes us to impose two restrictions on process specifications in CLL. More specifically we allow:

- Atomic (\(A^{\bot}\)) or optional (\(A^{\bot} \& B^{\bot}\)) inputs (including preconditions\(^1\)). In the general case, we represent these as \(n\) vectors of inputs \((\vec{I}_i)\). Each vector represents a set of optional inputs (or is singleton so as to represent an atomic input). Thus, a general formula for the inputs \((I)\) is:

\[
I = (\&^{\bot}_1 \vec{I}_1), (\&^{\bot}_2 \vec{I}_2), ..., (\&^{\bot}_n \vec{I}_n)
\]

Where: \(\&^{\bot}_i (a_1, a_2, ..., a_n) = a_1^{\bot} \& a_2^{\bot} \& ... \& a_n^{\bot}\).

- A single composite output \((O)\) consisting of outputs, effects, and exceptions. This output is expected to be a disjunction (⊕) of conjuncts (⊗). Each conjunct

\(^1\)Note that the “par” operator is assumed in CLL judgements, i.e. \(\vdash A, B\) and \(\vdash A \triangledown B\) are equivalent (see Section 4.2).
represents one of the possible results of the service as a simultaneous output of a set of resources. These sets are represented as vectors ($\vec{O}_i$). Thus, a general formula for the composite output is the following:

$$\vec{O} = (\bigotimes_i \vec{O}_1) \oplus (\bigotimes_i \vec{O}_2) \oplus \ldots \oplus (\bigotimes_i \vec{O}_n)$$

Where:

$$\bigotimes_i (a_1, a_2, \ldots, a_n) = a_1 \otimes a_2 \otimes \ldots \otimes a_n.$$

It is worth noting that the second restriction satisfies a result for Polarized Linear Logic (LLP - see Section 4.2) which states that at most one positive literal is sufficient to describe all provable formulas. Our restriction is based more on a pragmatic decision than this theoretical result though. It not only greatly facilitates automation (see Section 7.4), but also helps avoid the difficulty of handling processes with multiple outputs in CLL. In Section 4.3, we mentioned that the CLL Cut rule corresponds to communication between two processes. If we assume a process $F$ with two atomic outputs $A$ and $B$, i.e. $\vdash \Gamma, A, B$ and a process $G$ with two atomic inputs $A^\perp$ and $B^\perp$, i.e. $\vdash \Delta, A^\perp, B^\perp$ then it is not possible to communicate both $A$ and $B$ from $F$ to $G$, because the Cut rule involves only one literal. Instead, we express two outputs $A$ and $B$ as $A \otimes B$ and restrict process specifications to a single (composite) output.

Based on our two restrictions, a process can be specified in CLL using the following general formula:

$$\vdash I, \vec{O}$$

(6.1)

Note that the formula, as given, incorporates only the functional features of a process. It can be expanded to incorporate non-functional properties, such as cost and time, but only as a qualitative (type) specification.

Following formula (6.1), the translation of the available process for the credit card example (see Figure 2.2) and the requested composite process (Figure 2.3) are shown in Figure 6.1.

Based on the process interpretation of CLL judgements with the described restrictions, even though CLL allows for a wide range of formal statements, only a restricted subset of these actually correspond to sensible process specifications. For example, it is clear that $A^\perp$ represents an input $A$, whereas $X \otimes Y$ represents two parallel outputs $X$ and $Y$. The term $A^\perp \otimes Y$, however, is a perfectly valid CLL term that has no intuitive representation as a resource. In a similar way, not all CLL proofs correspond to intuitive process compositions. For example, we can easily produce the term $A^\perp \otimes Y$ by
Available services:

CreditCardInit: $\vdash \text{PAYMENT\_REQ}^\perp, \text{TRANSACTION} \otimes \text{PIN\_REQ}$

UserPINInput: $\vdash \text{PIN\_REQ}^\perp, \text{PIN}$

CreditCard

Transaction: $\vdash \text{TRANSACTION}^\perp, \text{PIN}^\perp, \text{PAYMENT} \oplus \text{EX\_BAD\_PIN}$

Request:

CreditCardPayment: $\vdash \text{PAYMENT\_REQ}^\perp, \text{PAYMENT} \oplus \text{EX\_BAD\_PIN}$

Figure 6.1: The available services and the request for the credit card example specified as CLL judgements.

composing two processes together, one with input $A$ and output $B$ and one with input $X$ and output $Y$ as follows:

$$
\vdash A^\perp, B \vdash X^\perp, Y \\
\vdash X^\perp, B, A^\perp \otimes Y \otimes
$$

Even though this is a sound CLL inference step, it creates a sentence which breaks our polarity restrictions and cannot be interpreted naturally as a process specification.

It is important to note that using the MALL fragment of CLL (see Sections 4.2 and 4.5.1.2) disallows the possibility of using replicable resources. The fact that some information (e.g. prices, dates, reference ids, etc.) or resources (e.g. documents that can be copied) are replicable cannot be expressed in a straightforward way in CLL without the of-course and why-not modalities. However, this lack is a cause for redundancy in the constructed workflows. If two processes share a common input which is replicable, such as a date provided by the user, it would be simpler if this input was specified only once in their parallel composition. In order to accomplish this, we introduce pre-specified processes that represent the functionality of replicating a particular type of resource at least twice. We call these processes Copy nodes, and they have the following logical specification, for any type $A$:

$$
\vdash A^\perp, A \otimes A \otimes \ldots \otimes A
$$

Although this solution is not ideal (compared to the flexibility of using the full fragment of CLL), it is sufficient for the purposes of process composition. The user may freely choose to introduce Copy nodes for any type in the composition. They therefore maintain the responsibility of deciding if a particular resource is replicable or not and
6.3 Process Calculus Specifications

avoid introducing Copy nodes for non-replicable types. It is worth noting that, when deploying the workflow as actual executable code (see Chapter 8), the user will be required to provide the functionality of each Copy node. If the involved resource is not explicitly replicable, implementing this functionality is the user’s responsibility.

With these points in mind, our CLL-based process specifications do not formally distinguish between consumable and persistent resources. For example, there is no distinction between an input parameter that describes a human actor that performs a process (such as the responsible doctor in a medical procedure) and a resource that is used up, such as blood in a transfusion procedure. If we follow the CLL interpretation strictly, each process should always consume all its inputs, including the human actor of our example! This can cause misinterpretations with respect to how persistent resources are managed in a deployed workflow.

One possible way of resolving these is by considering that the resources being consumed are in fact references to the actual persistent resources. These references are pieces of information that should in fact be consumed by the process, in the sense that the process does not maintain this information in local storage, nor is it communicated to any other process (unless explicitly specified in the workflow). The latter is an important argument when considering information provenance. In short, any reference to persistent resources should not be kept locally, but destroyed (consumed) as soon as it has served its purpose.

Having covered the logic and its usage as a process specification language, we proceed with the analysis of the corresponding process calculus specification based on the proofs-as-processes paradigm in the next section.

6.3 Process Calculus Specifications

In the previous section, we saw how a CLL specification can be used to describe the types of IOPEs of a process. The inner workings of the process remain abstract and the process is treated as a black box. However, based on the proofs-as-processes paradigm, the CLL specification has an implicit translation to the $\pi$-calculus, so that the process can communicate with others normally when composed via proof.

For example, assume an atomic process $P$ with output $x:A \otimes B$. This process must have the same (output) behaviour as a composite process $Q$ with output $x:A \otimes B$ that was
constructed using the \( \otimes \) rule on two individual (possibly atomic) processes with outputs \( ch_a:A \) and \( ch_b:B \) respectively, for some channels \( ch_a \) and \( ch_b \). For example, both \( P \) and \( Q \) have to be able to interact with another process \( R \) with input \( y:A \vdash \gamma B \). As a result, the \( \pi \)-calculus specification of (the output of) \( P \) must have the form \( ch_a \mathbin{\otimes} ch_b \) \( \langle F \rangle \mathbin{\otimes} \langle G \rangle \), where \( F \) is a process that outputs a message of type \( A \) through channel \( ch_a \) and \( G \) a process that outputs a message of type \( B \) through channel \( ch_b \). Note that if \( A \) and \( B \) are atomic, then \( F = (v \ a) \left( \overline{ch_a(a)} \cdot 0 \right) \) and \( G = (v \ b) \left( \overline{ch_b(b)} \cdot 0 \right) \), not only by intuition, but also to ensure smooth interaction with an axiom buffer such as \( I_{ch_a,z} = ch_a(a) \cdot \overline{z(a)} \cdot 0 \) in the case of \( A \).

Using the proofs-as-processes patterns to specify atomic processes in the \( \pi \)-calculus is, therefore, essential in order to ensure that the process behaves as expected within a composition constructed via CLL proof. For this reason, we have constructed a set of mappings for a systematic interpretation of CLL specifications of processes to \( \pi \)-calculus processes that ensures the expected behaviour. These are shown in Figure 6.1. Note that composite cases are mapped recursively. For example, parallel output \( z:A \otimes B \) is mapped to \( (v \ a, b) \left( \overline{z(a,b)} \cdot (A_\pi(a) \parallel B_\pi(b)) \right) \) where \( A_\pi(a) \) is the (recursive) mapping of component \( A \) over channel \( a \), the local channel that corresponds to that component in the pattern, whereas \( B_\pi(b) \) is the mapping of component \( B \) using local channel \( b \).

<table>
<thead>
<tr>
<th>Description</th>
<th>CLL term</th>
<th>( \pi )-calculus pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atomic output</td>
<td>( z:A )</td>
<td>( (v \ a) \langle \overline{z(a)} \rangle \cdot 0 )</td>
</tr>
<tr>
<td>Atomic input</td>
<td>( z:A^\perp )</td>
<td>( z(a) \cdot 0 )</td>
</tr>
<tr>
<td>Parallel output</td>
<td>( z:A \otimes B)</td>
<td>( (v \ a, b) \langle \overline{z(a,b)} \rangle \cdot (A_\pi(a) \parallel B_\pi(b)) )</td>
</tr>
<tr>
<td>Parallel input</td>
<td>( z:A^\perp \bowtie B^\perp )</td>
<td>( z(a,b) \cdot (A_\pi(a) \parallel B_\pi(b)) )</td>
</tr>
<tr>
<td>Optional output</td>
<td>( z:A \oplus B )</td>
<td>( (v \ x, y) \langle z(u,v) \rangle \cdot (u(x) \cdot (A_\pi(x)) + v(y) \cdot (B_\pi(y))) )</td>
</tr>
<tr>
<td>Optional input</td>
<td>( z:A^\perp \oplus B^\perp )</td>
<td>( (v \ u, v) \langle z(u,v) \rangle \cdot (u(x) \cdot (A_\pi(x)) + v(y) \cdot (B_\pi(y))) )</td>
</tr>
</tbody>
</table>

Table 6.1: Mappings of CLL process specifications to \( \pi \)-calculus terms based on proofs-as-processes rules. In the composite cases, we use \( X_\pi(y) \) to denote the recursive \( \pi \)-calculus mapping of a component \( y:X \) where \( y \) is the local channel corresponding to that component in the pattern.

Our introduced mappings have a clear symmetry with the proofs-as-processes translations shown in Figure 4.3. The optional output \( A \oplus B \) is perhaps the most notable case,
as it combines the translations of both $\oplus$ rules, since the choice between $A$ and $B$ is made internally by the process.

Each CLL sequent is in no way connected or dependent to any other CLL sequents in the same specification. We map this property to the $\pi$-calculus as a parallel composition ($|$) of the translations of the sequents. This has an unexpected effect in the process specification, namely the fact that the corresponding process produces its output in parallel with receiving its inputs. Usually processes are only able to produce outputs if they have received all of their inputs. However, since all CLL sequents are treated generically as the same (i.e. there is no logical distinction between input and output sequents apart from the extra-logical restrictions that we impose) it is not possible to express this input-to-output dependency in the logic.

Imposing this input-output dependency in the $\pi$-calculus translation would require synchronisation of the (possibly) multiple inputs before communicating the output. The complexity of the patterns used to describe composite inputs and outputs and the lack of a sequential composition operator in the $\pi$-calculus make describing this synchronisation in $\pi$-calculus terms without breaking consistency with the CLL specification impossible. However, it is worth noting that this synchronisation is actually imposed in the Scala code that is generated automatically when the processes are deployed (see Section 8.2.3.4).

Using the described set of mappings, the $\pi$-calculus specifications of the atomic processes in the Credit Card example (see Section 2.1.3) are shown in Figure 6.2.

It must be noted that, even though these particular specifications are valid $\pi$-calculus terms, they are not typical definitions of $\pi$-calculus agents and may seem unorthodox to process calculus experts. As an example of a significant difference, typical $\pi$-calculus agents are recursive so that they can run endlessly, representing, for example, always online servers or daemons that continuously provide a service while interacting with multiple clients. Our specified processes are designed to run once and terminate. We are modelling one-time executions of processes without taking into consideration their past or future. For example, even if our process is an always online web service, we are simply modelling a one-time interaction, even if the service persists afterwards. This point of view is also adopted at the level of a composite process: we intend to represent the appropriate connections and interactions between the component processes in order to achieve the desired result for a single run. This approach is much closer to the notion
⊢ **CreditCardInit**: $cci_{pr}: PAYMENT\_REQ$, $cci_{out}: TRANSACTION \otimes PIN\_REQ$

$\text{CreditCardInit}(cci\_out, cci\_pr) =$
\[
cci\_pr(payment\_req).0 || \\
((v cci\_out\_a, cci\_out\_b)(cci\_out(cci\_out\_a, cci\_out\_b)). \\
((v transaction)(cci\_out\_a(transaction).0)|| \\
(v pin\_req)(cci\_out\_b(pin\_req).0)))
\]

⊢ **UserPINInput**: $upi\_req: PIN\_REQ$, $upi\_out: PIN$

$\text{UserPINInput}(upi\_out, upi\_req) =$
\[
upi\_req(pin\_req).0 || (v pin)(upi\_out(pin).0)
\]

⊢ **CreditCardTransaction**: $cct\_tr: TRANSACTION$, $cct\_pin: PIN$, $cct\_out: PAYMENT \oplus EX\_BAD\_PIN$

$\text{CreditCardTransaction}(cct\_out, cct\_tr, cct\_pin) =$
\[
cct\_tr(transaction).0 || \\
cct\_pin(pin).0 || \\
((v cct\_out\_x, cct\_out\_y)(cct\_out(cct\_out\_u, cct\_out\_v). \\
(cct\_out\_u(cct\_out\_x).(v payment)(cct\_out\_x(payment).0) + \\
cct\_out\_v(cct\_out\_y).(v ex\_bad\_pin)(cct\_out\_y(ex\_bad\_pin).0)))
\]

Figure 6.2: The annotated CLL specifications of the available services for the credit card example and the corresponding $\pi$-calculus specifications.
of a workflow, where concurrency is used as means for optimisation of execution, rather than a model of a persistent, concurrent system between multiple servers and clients.

To further explain our approach, we remind the reader that the purpose of our methodology is not to create compositions of (typical) π-calculus agents, but rather to use the π-calculus as a language to describe the workflow of our verified, constructed composition of abstract processes via proof. In the case of atomic process, we merely designed our π-calculus translations so that they match the properties of the corresponding CLL specifications.

Finally, it is worth pointing out that the process calculus specifications only describe the means by which the processes communicate with their environment, which can also be referred to as the process interface. The inner workings of the process itself, or more specifically how its outputs are generated based on particular inputs, remains unspecified, and the process is still viewed as a black box.

### 6.4 Visualisation Using a Diagrammatic Interface

In the previous sections we saw how CLL can be used as an expressive language to describe processes in an abstract way and also how such CLL specifications have an implicit yet systematic translation to a more concrete π-calculus interface. However, creating CLL process specifications manually and performing CLL proof steps to develop compositions asks for some familiarity with CLL and its resource interpretation, and, more importantly, it also requires the user to have a decent level of expertise in theorem proving and the use of HOL Light. While these requirements are not intractable, it does make the methodology quite demanding for those whose main interest lies in the design of process workflows rather than performing logical inference.

During the development of our framework, we realised that we often used diagrams in order to explore the various ways of composing services and illustrate the information flow between them. This observation provided us with the motivation for developing a systematic diagrammatic approach that can abstract from the low-level text-based theorem proving. Besides, diagrammatic interfaces are commonly used as a frontend to

---

This section describes joint work with Sean Wilson who implemented the diagrammatic interface. This work was supported by EPSRC grant EP/J001058/1.
process modelling languages such as BPMN (see Section 2.1.2), and also to represent \( \pi \)-calculus agents as in the example of the PiVizTool (see Section 3.3.2). The main focus in designing our diagrammatic representation was to provide enough expressivity to allow the specification of any process, while hiding the complicated syntax and attached semantics of the underlying logic.

In our adopted diagrammatic notation, each process is represented as a rectangle with dangling edges standing for its inputs and outputs. Solid edges correspond to composite inputs/outputs whereas dashed edges correspond to optional inputs/outputs.

For example, we can represent a process \( P_a \) with inputs \( A \) and \( B \) and outputs \( X \) and \( Y \) in CLL and our graphical notation is shown below:

\[
\vdash A^\perp, B^\perp, X \otimes Y
\]

Similarly, a process \( P_b \) with inputs \( A \) and \( B \) and either an output \( X \) or an exception \( E \) is represented as follows:

\[
\vdash A^\perp, B^\perp, X \oplus E
\]

More complicated cases can be represented in the same way. For example, a process \( P_c \) with inputs \( A \) and \( B \), and outputs \( X \) and \( Y \) that may throw an exception \( E \) can be represented as shown below:

\[
\vdash A^\perp, B^\perp (X \otimes Y) \oplus E
\]

Copy processes are the only exception to the above, and are represented using a special circular node in order to make the overall appearance of the workflows cleaner and simpler. An example for a Copy process which replicates input \( A \) 3 times is shown below:
Based on this diagrammatic notation, we have constructed a GUI that allows a seamless interaction with HOL Light. The GUI not only visualises CLL statements, but also allows to perform proofs by applying mouse gestures. This functionality is demonstrated in Section 7.4.

The GUI is implemented in Java where JGraph (JGraph Ltd, 2013) is used for graph visualisation and layout. Behind the scenes, interactions with the GUI involve sending commands to an instance of HOL Light. Executed HOL Light commands communicate their results back to the GUI via JSON (Crockford, 2006) so that the state of the current proof can be visualised. As all of the reasoning takes place within HOL Light, this guarantees that any proof we complete is correct.

As an example, the diagrammatic representation of the available processes in the credit card example (see Figure 6.1) is shown in Figure 6.8.

Before beginning a composition, the user first defines the atomic processes they want to use. A toolbar button is used to bring up a dialog for adding new processes. The user then specifies the inputs and outputs, and name for the process by filling in text fields. The \( \otimes \) and \( \oplus \) operators are used to specify composite and optional outputs respectively. The HOL Light command for creating a new process is then executed in the background and the corresponding diagrammatic representation of the new process is added to the graph displayed by the GUI. We present our diagrammatic composition using this tool in the next chapter. More details about our diagrammatic tool can be found in one of our papers (Papapanagiotou et al., 2012b).

### 6.5 Ontological Classes as Types

The process specifications introduced in Section 6.2 use CLL propositions to represent abstract types. Advanced object types with richer semantics can be used to instantiate these abstract type specifications, provided a mapping can be introduced. In this
Figure 6.7: A screenshot of our diagrammatic interface. The graph shows the current state of a proof in progress. The list on the left contains a list of all the atomic and composite services defined so far.
section, we present an example of a such a rich underlying type, namely ontological classes. These provide well-defined semantics for each (originally abstract) type, and allow a new level of reasoning about the types themselves.

The Semantic Web is the W3C’s view of the next generation of the World Wide Web. In it, the existing information of the web is augmented with meaningful semantics which make it machine-understandable. As a result, the sharing of information between humans and computers is expected to be better.

Ontologies play a key part in describing information using meaningful, machine understandable semantics in the Semantic Web. They provide a representation for the conceptualization of a domain that can be understood and shared equally by people and computer applications (Gomez-Perez et al., 2004). In order to accomplish this, several logic-based ontology languages have been developed. Among these OWL (Bechhofer et al., 2004) and its newest successor OWL2 (Grau et al., 2008) are the most widely accepted and used in the community.

In a nutshell, using these languages, concepts can be defined as named classes in an “is-a” hierarchy starting from the root concept \textit{Thing}. As an example, let us con-
Consider the Pizza ontology (Drummond et al., 2007), a commonly used toy ontology. An excerpt of the defined hierarchy in the Pizza ontology is shown in Figure 6.9. In this, a MeatyPizza and a VegetarianPizza are both types of pizzas, and therefore subclasses of Pizza, which, in turn, is a subclass of Food. The Food class is a DomainConcept, a subclass of the root concept Thing. A set of properties or roles can be defined for each of these classes to further describe the corresponding concepts and the relations between them (such relations are also referred to as object properties). For example, the object property hasTopping describes the relation between a Pizza (domain) and a corresponding PizzaTopping (range). Each class can then have individuals that correspond to concrete instances of the class. For example, Scotland is an individual of the Country class. Finally, ontologies can be combined to create more complicated descriptions, for example to define concepts that combine properties from two different ontologies.

Figure 6.9: Excerpt of the class hierarchy tree for the Pizza ontology.

The logical cores of OWL and OWL2 allow for efficient reasoning such as checking for consistency and classification. The former is a check about whether an ontology is consistent based on both its definitions and individuals. For example, an ontology that includes an instance of a Pizza that is both a MeatyPizza and a VegetarianPizza when these two classes are defined as disjoint is inconsistent. The latter involves the creation of subclass relationships based on the definitions of the various classes. For example, a Pizza that only has (hasTopping) VegetarianToppings can be autom
ically classified as a VegetarianPizza (which is a subclass of Pizza). It is worth noting that many ontology queries can be reduced to either of these two tasks. These tasks are accomplished by modern, widely used ontological reasoners such as Racer (Haarslev and Mller[2001]), Pellet (Sirin et al. [2007]) and FACT++ (Tsarkov and Horrocks, 2006), using tableau-based decision procedures (Horrocks et al., 2000).

Our system uses the OWL-API (Horridge and Bechhofer, 2009) in order to parse any OWL2 ontology and make its defined concepts available to use as input or output types in the specifications of processes. In addition, an object property can be viewed as a process with its domain as input and its range as output. The OWL-API allows an external connection to the aforementioned ontological reasoners. Incorporating this functionality in our system results in a heterogeneous reasoning environment where the user can formally verify not only the type correctness of process compositions, but also the correctness of the involved ontological class specifications. In addition, some of the features of ontologies, such as their hierarchical relations and object properties, can be modelled as processes and be included as part of the constructed workflows.

The use of ontologies in our framework is demonstrated in Chapter 10.4.4, where we describe the development and use of a small medical ontology in the context of constructing formally verified healthcare collaboration patterns.

### 6.6 Conclusion and Future Work

In our process modelling and composition methodology, we adopt an abstract, two-tiered specification for processes. Firstly, we introduce CLL specifications of the information flow, i.e. type specifications for the IOPE’s of each process. Then, based on these and the proofs-as-processes correspondence, we extract \(\pi\)-calculus specifications that appropriately describe the means of communication of each process. Apart from these two parts of the specification, the implementation details of the modelled processes are unspecified as we do not need to take them into consideration during composition.

Even though the CLL and \(\pi\)-calculus specifications allow us to reason about and guarantee a number of properties of the composed workflow with particular emphasis on the information flow, the level of abstraction limits the capabilities for further reasoning beyond this, such as reasoning about the information being exchanged or quantitative
properties. The latter can be particularly important in some cases where, for example, when reasoning about time or cost is required.

One way of overcoming the relatively restricted expressiveness of CLL, is to develop mappings of the CLL abstract types to richer type systems. We discussed the possibility of mapping CLL types to ontological classes here, but other logic-based options may be worth investigating in the future. For example, there may be options to combine our specifications with those involving different logics and verification techniques, such as Hoare triples (Hoare, 1969) for verification of partial correctness (reasoning about the information itself), separation logic (Reynolds, 2002) and similar logics developed for reasoning about resource sharing and consumption in programs (Aspinall et al., 2004), and finite-state systems for model checking (reasoning about quantitative properties) is a particularly interesting topic for further work.

In a similar way, from the more practical point of view, considering an (even partial) correspondence between widely used process modelling languages such as BPMN and our CLL specifications, would add considerable value to our proposed methodology as a pragmatic solution.

Finally, it is worth noting that, working with a subset of restricted CLL judgements (based on the two restrictions discussed in Section 6.2) may facilitate the construction of specialised, automated proof tactics for process composition (see Section 7.4), but it limits the expressive power of CLL as a specification language. Although the current subset is expressive enough for the core needs of our case-studies, lifting the introduced constraints may allow for more complicated and interesting specifications, which may involve, for example, sub-processes (methods).
Chapter 7

Process Composition via Proof

Inspired by the work of Rao et al. (see Section 5.2), we have constructed a framework for formally verified process composition based on the proofs-as-processes paradigm (see Chapter 4). Having discussed process specification in the previous chapter, we now present the proof tools, including inference rules and tactics that we developed for constructing process workflows. We also analyse some of the observed properties of the constructed workflows and reconstruct the Ski example of Rao et al.

7.1 Introduction

The core premise of proof-based process composition involves the combination of \( n \) available processes specified in CLL using logical inference in order to achieve a requested CLL specification of a composite process.

For each (atomic or composite) process \( i \) we construct a CLL specification \( A_i \) of the form \( \vdash I, O \) based on the methodology described in Section 6.2. Similarly, we construct a CLL specification \( R \) of the requested composite process. We then attempt to prove \( R \) as a conjecture given \( A_1, A_2, ..., A_n \) as assumptions, i.e. the following schematic goal:

\[
\frac{A_1 \quad A_2 \quad \cdots \quad A_n}{R}
\]

Given the use of Higher Order Logic as our meta-logic (see Section 4.4), this goal can be described using the following statement:

\[
A_1 \wedge A_2 \wedge ... \wedge A_n \rightarrow R
\]
In essence, this corresponds to the construction of a CLL proof tree with $A_1, A_2, \ldots, A_n$ as assumptions (used to close its branches) and $R$ as the conclusion. If such a proof tree exists, this means that the proof/composition can be accomplished, and $R$ is a valid logical representation of a composite process that can be constructed using processes $A_1, A_2, \ldots, A_n$. Moreover, using the proofs-as-processes paradigm, we can extract the $\pi$-calculus translation of the proof, which provides a full description of the structure of the composite process $R$.

Note that the schematic goal is non-linear (i.e. it does not have the properties of CLL), meaning that an assumption $A_i$ can be used several times in order to close a number of proof branches or may not be used at all. When considering the resulting composite process $R$, this means $R$ may include $A_i$ as a component several times or $A_i$ may not be a component of $R$ at all. Essentially, we consider processes as replicable resources that can be used several times.

It is also worth noting that the requested process $R$ may not necessarily be fully specified. For example, we may not know in advance what kind of exceptions may occur in the composite process, or we may not know all the necessary inputs but only the desired output. Our system uses metavariables and unification in HOL Light to allow such flexibility in the specification of the goal (see Section 4.4.5.4). In fact, it is possible to leave the specification of the goal completely unspecified so that we can construct any composite process without any restrictions. We like to call this kind of usage of our system the “discovery mode” as, in a sense, the user is trying to discover the appropriate composition on the fly.

To start a proof, the diagrammatic interface described in Section 6.4 can be used to specify a new composite service with the addition that metavariables are allowed for as yet unknown outputs. The user must then manipulate the graph by applying actions (see Section 7.4) to create a formal graph that represents this composite service. Each action performs inference steps that modify the proof state of HOL Light and correspond to an intuitive composition of two processes. The result is communicated back to the GUI so that the corresponding graph transformation can be applied.

Our main aim in the design and development of these actions is to accommodate intuitive ways in which the user would expect to be able to compose processes. In particular, we focus on parallel, conditional, and sequential (output to matching input) composition. Based on the process interpretation of CLL proofs, each of these actions
7.2 Inference Rules for Processes

As explained in Section 6.2, not all CLL proofs correspond to sensible process specifications. This makes the task of reasoning about processes more tractable (than if we had to deal with arbitrary CLL proofs) as we only need to focus on inference steps that are valid in the context of process composition.

This observation led us to introduce some derived CLL rules that correspond to natural, valid process composition steps. These are particularly helpful when it comes to the construction of automated tactics (see Section 7.4), since they ensure the valid use of the CLL rules (to some degree). In particular, we introduce two types of derived rules. The first type, shown in Figure 7.1, are weakened or specialised versions of the original CLL inference rules that ensure a better match with the process specification restrictions. The second type, shown in Figure 7.2, are rules which describe some standard compositions of processes with buffers. We give more details for each of the
Chapter 7. Process Composition via Proof

derived rules in the next few sections.

\[
\frac{\Gamma, x:\bot, y:\bot \vdash P :: \vec{w} : \emptyset}{\Gamma, z: (A \otimes B) : \bot \vdash \text{\texttt{\&in}}} \\
\frac{\Gamma, a:\bot, b:B \vdash P :: \vec{w} : \emptyset, a:A \bot, b:B}{\Gamma, c:C \bot, b:B \vdash Q :: \vec{w} : \emptyset \vdash (P, Q) :: \vec{w} : \emptyset, x:(A \oplus C) \bot, b:B \vdash (P, Q) :: \vec{w} : \emptyset, x:(A \oplus C) \bot, y:B \oplus D} \quad \&\text{\texttt{\&proc}} \\
\frac{\Gamma, x:C \vdash F :: \vec{u} : \emptyset, x:C \bot, y:B \vdash G :: \vec{v} : \emptyset, y:B \bot \vdash \text{\texttt{Cut}}(F, G) :: \vec{u} : \emptyset, \vec{v} : \emptyset \vdash \text{\texttt{Cut}}} \\
\]

Figure 7.1: Specialised versions of CLL inference rules.

7.2.1 The \texttt{\&in} Rule

The \texttt{\&in} rule (where \texttt{in} stands for “input”) is a specialised version of the \texttt{\&} rule. As such, its derivation is a simple application of the \texttt{\&} rule:

\[
\frac{\Gamma, a\bot, b:B \vdash P :: \vec{w} : \emptyset, a:A \bot, b:B}{\Gamma, c:C \bot, b:B \vdash Q :: \vec{w} : \emptyset \vdash (P, Q) :: \vec{w} : \emptyset, x:(A \oplus C) \bot, b:B \vdash (P, Q) :: \vec{w} : \emptyset, x:(A \oplus C) \bot, y:B \oplus D} \quad \&\text{\texttt{\&proc}} \\
\frac{\Gamma, y:B \vdash \text{\texttt{\&in}}} \\
\frac{\Gamma, y:B \vdash G :: \vec{v} : \emptyset, y:B \bot \vdash \text{\texttt{Cut}}(F, G) :: \vec{u} : \emptyset, \vec{v} : \emptyset \vdash \text{\texttt{Cut}}} \\
\]

The difference between the two rules is that \texttt{\&in} connects two negated (as opposed to arbitrary) terms with \texttt{\&}. Since \texttt{\&} is an input operator, we expect our process composition proofs to only apply the \texttt{\&} rule on negated sequents, i.e. inputs. Semantically, this rule describes the fact that if a process has two inputs \(A \bot\) and \(B \bot\), then we can combine these in a composite input \((A \otimes B) \bot\).

It should be noted that \(A = (A \bot) \bot\), so it is still possible to apply \texttt{\&} on positive sequents (outputs) with the \texttt{\&in} rule. However, we ensure our constructed tactics maintain some canonicity in the proofs so that negation never appears in an output.
7.2. Inference Rules for Processes

\[ \vdash P :: \vec{w} : \Gamma, \ a : A \quad \vdash Q :: \text{buf} : B^\perp, \ b : B \quad \otimes \text{buf} R \]

\[ \vdash \otimes (P, Q) :: \vec{w} : \Gamma, \ \text{buf} : B^\perp, \ z : A \otimes B \]

\[ \vdash P :: \vec{w} : \Gamma, \ a : A \quad \vdash Q :: \text{buf} : B^\perp, \ b : B \quad \otimes \text{buf} L \]

\[ \vdash \otimes (Q, P) :: \text{buf} : B^\perp, \ \vec{w} : \Gamma, \ z : B \otimes A \]

\[ \vdash P :: \text{na} : A^\perp, \ a : B \oplus C \quad \vdash Q :: \text{buf} : B^\perp, \ b : B \quad \& \text{Lbuf} R \]

\[ \vdash \&_\text{no} \left( P, \text{L}(Q) \right) :: \text{no} : (A \oplus B)^\perp, \ a : B \oplus C \]

\[ \vdash P :: \text{na} : A^\perp, \ a : B \oplus C \quad \vdash Q :: \text{buf} : B^\perp, \ b : B \quad \& \text{Lbuf} L \]

\[ \vdash \&_\text{no} \left( \text{L}(Q), P \right) :: \text{no} : (B \oplus A)^\perp, \ a : B \oplus C \]

\[ \vdash P :: \text{na} : A^\perp, \ a : B \oplus C \quad \vdash Q :: \text{buf} : C^\perp, \ c : C \quad \& \text{Rbuf} R \]

\[ \vdash \&_\text{no} \left( P, \text{R}(Q) \right) :: \text{no} : (A \oplus C)^\perp, \ a : B \oplus C \]

\[ \vdash P :: \text{na} : A^\perp, \ a : B \oplus C \quad \vdash Q :: \text{buf} : C^\perp, \ c : C \quad \& \text{Rbuf} L \]

\[ \vdash \&_\text{no} \left( \text{R}(Q), P \right) :: \text{no} : (C \oplus A)^\perp, \ a : B \oplus C \]

Figure 7.2: Derived inference rules involving buffers.
7.2.2 The &in Rule

Similarly to the &in rule, the &in rule is a specialisation of the & rule in the context of processes. Since & is an input operator, the &in rule ensures we only connect negated sequents (inputs) with &. It is worth noting that the single output B is made explicit in this rule. This strengthens the enforcement of this rule on inputs (negated terms) only, especially taking into consideration that processes specifications only have one output. It also provides some symmetry with the &proc rule.

The derivation of &in is a simple application of the & rule as follows:

\[
\begin{align*}
\Gamma &\vdash A \bot, B \\
\Gamma &\vdash C \bot, B \\
\Gamma &\vdash (A \oplus C) \bot, B \\
\end{align*}
\]

Semantically, this rule corresponds to the construction of a conditional (if-then-else) statement involving two processes P and Q. The two processes are identical except for a single different input each (A \bot and C \bot respectively). In the resulting composite service, if A is provided as input, then P is executed, else if C is provided then Q is executed. In both cases we obtain the same type of result B (generated from either P or Q).

7.2.3 The &proc Rule

The &proc rule is a more usable version of the &in rule in the context of processes. It enables the creation of a conditional statement between processes P and Q assuming they only differ on a single input (A \bot and C \bot respectively), and on their output (B and D respectively). In the resulting composite service, if A is provided as input, then P is executed and we obtain B, else if C is provided then Q is executed and we obtain D.

The rule is derived from the &in (see Section 7.2.2) and \oplus rules as follows:

\[
\begin{align*}
\Gamma &\vdash A \bot, B \\
\Gamma &\vdash A \bot, B \oplus D \\
\Gamma &\vdash C \bot, D \\
\Gamma &\vdash C \bot, B \oplus D \\
\Gamma &\vdash (A \oplus C) \bot, B \oplus D \\
\end{align*}
\]
7.2.4 The \textit{Cut}' Rule

The \textit{Cut}' rule is a specialised version of the \textit{Cut} rule for the case where the two communicating processes are already sharing the same channel $x$. This rule, when applicable, eliminates an unnecessary introduction of a new variable $z$ and unnecessary substitutions in the two communicating processes. Therefore, the role of \textit{Cut}' is entirely related to simplification and efficiency and has no deeper interpretation or effect in the logical proof. Any of our tactics that need to use the \textit{Cut} rule, try to use \textit{Cut}' first and fall back to the normal \textit{Cut} upon failure.

The \textit{Cut}' rule is a direct consequence of the original \textit{Cut} rule and the fact that for all processes $P$ and names $x$, $P[x/x] = P$.

7.2.5 The $\otimes \text{buf}$ Rules

The $\otimes \text{buf}$ rules correspond to the parallel composition of a process $P$ with a buffer. Since buffers can either be atomic (axiom buffers) or composite, all the rules involving buffers include the buffer specification as a schematic assumption (see Section 7.3 for more details).

In order to understand the interpretation of this rule in the context of process composition, assume a process $P$ with a single input $A$ and an output $G$. Consider the case where we want $P$ to interact with another process $Q$ which has output $A \otimes B$. From this composite output, $P$ is only able to handle $A$. Therefore, $B$ must be handled by a buffer which will merely forward $B$ without changes. The composition of $P$ with such a buffer results in a composite process that is able to receive both $A$ and $B$ as inputs, and then execute $P$ to generate $G$ from $A$ while leaving $B$ unchanged, i.e. has output $G \otimes B$.

The two different versions of $\otimes \text{buf}$, namely $\otimes \text{buf} R$ and $\otimes \text{buf} L$, differentiate between adding the buffer to the right or the left of the original input respectively. This is done to avoid using the commutativity property of $\otimes$, which would result in a complicated $\pi$-calculus translation with more axiom buffers (see Section 5.5.3). Both rules can be derived by a simple application of the $\otimes$ rule. For example, $\otimes \text{buf} R$ is derived as follows:
\[
\frac{\Gamma, A \vdash B^\perp, B}{\Gamma, B^\perp, A \otimes B} \otimes
\]

### 7.2.6 The &buf Rules

The &buf rules involve a process \( P \) with a single input \( A^\perp \) and an optional output \( B \oplus C \). This process is handled as a special case, because using the &buf rules and a given buffer allows us to replace \( A^\perp \) with an optional input that involves \( A \) and either \( B \) or \( C \). Depending on whether we use \( B \) or \( C \), we use the \&Lbuf \cdot or \&Rbuf \cdot rules respectively, whereas depending on whether the option is added to the left or the right of \( A \), we use the \& \cdot bufL or \& \cdot bufR rules respectively. For example, in order to construct the optional input \((A \otimes B)^\perp\), we need to use \( B \) and add it to the right of \( A \), i.e. we need to use the \&LbufR rule.

The rules are easily derived from a combination of the &in rule and the \oplus rules. For example the \&LbufR rule is derived as follows:

\[
\frac{\Gamma, A^\perp, B \oplus C \vdash B^\perp, B \oplus C}{\Gamma, (A \oplus B)^\perp, B \oplus C} \&in \oplusR
\]

It is worth noting that this derivation is only possible because \( A \) is the only input of \( P \). Otherwise, the & rule would not be applicable.

### 7.3 Composite Buffers

In this section, we discuss the notion of composite buffers as opposed to the atomic, or axiom buffers. An axiom buffer, as explained in Section 4.3, is the \( \pi \)-calculus translation \( I_{x,y} \) of the identity rule of CLL as shown below:

\[
\frac{}{\Gamma I_{x,y}:: y:A, x:A^\perp} \quad (7.1)
\]

Note that, as mentioned in Section 4.4.4, there is an implicit restriction in (7.1) that \( A \) is an atomic proposition. The axiom buffer \( I_{x,y} \) is a process that receives a message
through channel $x$ and forwards it through another channel $y$. The $\pi$-calculus definition is the following:

$$I_{x,y} \equiv x(a).\overline{y(a)}.0 \quad (7.2)$$

More generally, we refer to any CLL judgement of the form $\vdash A \perp$, $A$ as a buffer of type $A$. If $A$ is an atomic proposition, then this matches with the identity axiom of CLL, and is therefore an atomic or axiom buffer. If $A$ is a composite term, for example in the case of $\vdash (X \otimes Y) \perp$, $X \otimes Y$, we refer to the judgement and corresponding $\pi$-calculus process as a composite buffer.

This distinction has no implications in the logic, because composite buffers can be broken down to atomic ones via proof in a straightforward way (as demonstrated by our introduced tactics in the following sections). However, the distinction is important when considering the translation to processes.

To demonstrate this, consider the case of a sequential composition $Q$ between a process $P$ with input $A$ and output $X \otimes Y$, specified as $\vdash P :: a : A \perp$, $out : X \otimes Y$ and a buffer $Buf$ of type $X \otimes Y$ specified as $\vdash Buf :: i_{buf} : (X \otimes Y) \perp$, $o_{buf} : X \otimes Y$. Based on the translation of atomic processes introduced in Section 6.3, the $\pi$-calculus definition for $P$ is the following:

$$P(a, out) =$$

$$a(aa).0 \parallel (\forall out_x, out_y)(\overline{o_{buf}(out_x, out_y)}).((\forall x)(\overline{out_x(x)}.0) \parallel (\forall y)(\overline{out_y(y)}.0))) \quad (7.3)$$

The sequential composition of $P$ and $Buf$ is a simple application of the Cut rule as shown in the following subproof:

$$\vdash P :: a : A \perp, \quad out : X \otimes Y \quad P \vdash Buf :: i_{buf} : (X \otimes Y) \perp, \quad o_{buf} : X \otimes Y \quad \vdash Q :: a : A \perp, \quad o_{buf} : X \otimes Y \quad \text{Cut} \quad (7.4)$$

Using the proofs-as-processes translation we obtain the following composite process:

$$Q(a, o_{buf}) = \text{Cut}^\pi(P, Buf)$$

$$= (\forall z)(P[z/out] \parallel Buf[z/i_{buf}]) \quad (7.5)$$

$$= (\forall z)(P(a, z) \parallel Buf(z, o_{buf}))$$

Notice that the Cut rule dictates that $P$ and $Buf$ interact through channel $z$, marked in bold.
We can obtain \( P(a, z) \) from (7.3), thus we only have \( Buf(z, o_{buf}) \) as an unknown. The \( \pi \)-calculus definition of \( Buf \) depends on how we choose to close proof (7.4).

If we ignore the implicit restriction of atomicity in the identity axiom and we treat \( Buf \) as an axiom buffer, then the proof can be completed as follows:

\[
\begin{align*}
\vdash P :: a : A \perp, \; out : X \otimes Y & \quad \text{Ax} \\
\vdash Buf :: i_{buf} : (X \otimes Y) \perp, \; o_{buf} : X \otimes Y & \quad \text{Id} \\
\vdash Q :: a : A \perp, \; o_{buf} : X \otimes Y & \quad \text{Cut}
\end{align*}
\] (7.6)

We then obtain the following translation for \( Buf \):

\[
Buf(i_{buf}, o_{buf}) = i_{buf}(x).o_{buf}(x).0
\] (7.7)

The output channel \( z \) of \( P \) which corresponds to channel \( out \) in (7.3) sends a vector of two messages \( out_x \) and \( out_y \), whereas the input channel \( z \) of \( Buf \) which corresponds to channel \( i_{buf} \) in (7.7) expects a single message \( x \). The \( \pi \)-calculus reduction for the communication between \( P \) and \( Buf \) is, therefore, not applicable in this case.

Breaking \( Buf \) down to atomic buffers solves this problem. The corrected composition proof is as follows:

\[
\begin{align*}
\vdash I_{i_x}, o_x :: i_x : X \perp, \; o_x : X & \quad \text{Id} \\
\vdash I_{i_y}, o_y :: i_y : Y \perp, \; o_y : Y & \quad \text{Id} \\
\vdash \bigotimes_{o_{buf}} (I_{i_x}, o_x, I_{i_y}, o_y) :: i_x : X \perp, \; i_y : Y \perp, \; o_{buf} : X \otimes Y & \\
\vdash P :: a : A \perp, \; out : X \otimes Y & \quad \text{Ax} \\
\vdash Buf :: i_{buf} : (X \otimes Y) \perp, \; o_{buf} : X \otimes Y & \quad \text{Cut}
\end{align*}
\] (7.8)

From the proofs-as-processes translation, we obtain the appropriate \( \pi \)-calculus term for \( Buf \), which is the following:

\[
Buf(i_{buf}, o_{buf}) = \bigotimes_{o_{buf}} \left( \bigotimes \left( I_{i_x}, o_x, I_{i_y}, o_y \right) \right) \\
= i_{buf}(i_x, i_y).o_x.o_y.(o_{buf}(o_x, o_y).(i_x(x).o_x(x).0 \; || \; i_y(y).o_y(y).0))
\] (7.9)

In this case, in the resulting composition of (7.5), the message vector size of the input channel \( z \) of \( Buf \) which corresponds to channel \( i_{buf} \) in (7.9) now matches the vector...
7.3. Composite Buffers

size of $out$ in $P$ as seen in (7.3), so the $\pi$-calculus reduction and the corresponding interaction are possible.

Using buffers is very common in our composition proofs. Whenever a process is unable to handle a particular resource, we compose it with a buffer which enables the input of the resource and forwards it unchanged. Some examples were presented in our derived inference rules involving buffers in Section [7.2]. In those cases, we keep the required buffer as an extra premise in the rule and do not discharge it using the identity axiom, precisely to avoid the situation described above.

It is important to note that the distinction between information that is handled by a process and information that is buffered through is also made in the diagrammatic visualisation. More specifically, grey edges are used to represent buffered resources as opposed to the solid black edges representing connections with the actual processes.

In order to derive composite buffers and properly discharge the corresponding premises in our rules, we have constructed two automated tactics, namely $\text{BUFFER\_TAC}$ and $\text{PARBUF\_TAC}$, which we describe in the next two sections. These tactics are an important part of our high level composition tactics described in Section [7.4] and are needed to support the diagrammatic composition of processes.

7.3.1 BUFFER\_TAC

We introduce $\text{BUFFER\_TAC}$ as a HOL Light tactic that proves any CLL judgement of the form $\vdash A \perp, A$, i.e. any atomic or composite buffer. Given our chosen CLL polarisation and restrictions (see Section [6.2]), $A$ can either be atomic, or composite of the form $X \otimes Y$ or $X \oplus Y$, where $X$ and $Y$ can also be atomic or composite of the same forms. Based on this, $\text{BUFFER\_TAC}$ is implemented as a backward, recursive tactic, which follows a simple algorithm:

1. If $A$ is atomic then use the identity axiom:

   \[
   \vdash A \perp, A \quad Id
   \]

2. If $A$ is of the form $X \otimes Y$ then we use the $\exists in$ rule followed by the $\otimes$ rule. This breaks down the given buffer of type $X \otimes Y$ to two new, possibly composite buffers of type $X$ and $Y$ respectively. We use $\text{BUFFER\_TAC}$ recursively to prove
these buffers, as shown below:

\[
\begin{align*}
\text{BUFFER_TAC} & \quad \text{BUFFER_TAC} \\
\vdash X^\perp, X & \quad \vdash Y^\perp, Y \\
\vdash X^\perp, Y^\perp, X \otimes Y & \quad \vdash (X \otimes Y)^\perp, X \otimes Y \\
\end{align*}
\]

It should be noted that some extra automation is required to perform the correct context splitting between \(X^\perp\) and \(Y^\perp\) in the application of the \(\otimes\) rule. This effectively allows \text{BUFFER_TAC} to also prove judgements of the form \(\vdash (Y \otimes X)^\perp, X \otimes Y\).

3. If \(A\) is of the form \(X \oplus Y\) then we use the \&\text{in} rule followed by the \(\oplus\) rules. Similarly to the \(\otimes\) case, this breaks down the given buffer to two new buffers which are proven, in turn, by a recursive call to \text{BUFFER_TAC}:

\[
\begin{align*}
\text{BUFFER_TAC} & \quad \text{BUFFER_TAC} \\
\vdash X^\perp, X & \quad \vdash Y^\perp, Y \\
\vdash X^\perp, X \oplus Y & \quad \vdash Y^\perp, X \oplus Y \\
\vdash (X \oplus Y)^\perp, X \oplus Y & \quad \vdash (X \oplus Y)^\perp, X \oplus Y \\
\end{align*}
\]

As for the automated context splitting in the \(X \otimes Y\) case, we have built some automation in order to determine which of the two \(\oplus\) rules is appropriate to use. This effectively allows \text{BUFFER_TAC} to also prove judgements of the form \(\vdash (Y \oplus X)^\perp, X \oplus Y\).

This algorithm allows the automatic construction of the appropriate \(\pi\)-calculus buffers in each case.

For example, the automatic proof performed by \text{BUFFER_TAC} for the composite buffer \(\vdash (A \oplus (B \otimes C))^\perp, A \oplus (B \otimes C)\) is the following:

\[
\begin{align*}
\vdash A^\perp, A & \quad \text{Id} \\
\vdash A^\perp, A \oplus (B \otimes C) & \quad \text{Id} \\
\vdash (A \oplus (B \otimes C))^\perp, A \oplus (B \otimes C) & \quad \text{Id} \\
\vdash B^\perp, B & \quad \text{Id} \\
\vdash B^\perp, C^\perp, B \otimes C & \quad \text{Id} \\
\vdash (B \otimes C)^\perp, B \otimes C & \quad \text{Id} \\
\vdash (B \otimes C)^\perp, A \oplus (B \otimes C) & \quad \text{Id} \\
\vdash (A \oplus (B \otimes C))^\perp, A \oplus (B \otimes C) & \quad \text{Id} \\
\end{align*}
\]

Notice the use of the identity axiom 3 times, since we are breaking down the composite buffer into 3 axiom buffers of type \(A, B,\) and \(C\) respectively.
The constructed process calculus term of this proof is the following, where \( \text{in} \) and \( \text{out} \) are the input and output channels of the buffer respectively:

\[
\begin{align*}
&\overset{x_0,y_0}{\text{in}} \left( \overset{x_1}{L} \left( I_{x_0,x_1} \right) \right) \overset{y_2}{\text{out}} \left( \overset{x_4,y_4}{\otimes} \left( \overset{x_5,y_5}{\nu} \left( I_{x_4,x_5} \right) \right) \right) \\
&\text{This corresponds to the following } \pi\text{-calculus process:}
\end{align*}
\]

\[
(v \ u, v) \ (\overline{i}m(u, v) \cdot (u(x_0) \cdot (v \ x_1) \cdot \ (out(u,v) \cdot \nu(x_1) \cdot x_0(a) \cdot \overline{x_1}(a) \cdot 0) + v(y_0)) \\
(v \ y_2) \ (out(u,v) \cdot \nu(y_2) \cdot y_0(x_4,y_4) \cdot (v \ x_5,y_5) \ (\overline{y_2}(x_5,y_5) \cdot (x_4(b) \cdot \overline{x_5}(b) \cdot 0 \ || \ y_4(c) \cdot \overline{x_5}(c) \cdot 0))))
\] (7.10)

The complexity of this term is noteworthy and provides a good reason for minimizing the use of buffers in our proofs.

### 7.3.2 PARBUF_TAC

The PARBUF_TAC is used to prove the specialised set of buffers of the form \( \vdash \Gamma, \ O \) where \( \Gamma = A_1^\perp, A_2^\perp, \ldots, A_n^\perp \) and \( O = A_1 \otimes A_2 \otimes \ldots \otimes A_n \) for any number \( n \) of either atomic or optional inputs \( A_i \). These correspond to parallel compositions of atomic or optional buffers, i.e. they have several atomic or optional inputs \( A_i^\perp \) which are then forwarded in parallel (i.e. using only \( \otimes \)) in the output \( O \). We call this particular type of processes parallel buffers of type \( O \).

For example, \( \vdash A^\perp, (B \oplus C)^\perp, D^\perp, A \otimes (B \oplus C) \otimes D \) is a parallel buffer provable by PARBUF_TAC, whereas \( \vdash (X \otimes Y)^\perp, X \otimes Y \) is not because \( (X \otimes Y)^\perp \) is neither atomic nor optional. Parallel buffers appear often when dealing with buffered resources of optional cases in our automated tactics (see Section 7.4).

The PARBUF_TAC tactic is similar to BUFFER_TAC, but treats the parallel case of \( O = X \otimes Y \) (case 2 above) differently. In this case, since \( X \) is atomic or optional, we use the \( \otimes \) rule to split our goal in two buffers, one of type \( X \) and a parallel buffer of type \( Y \). We prove the \( X \) buffer using BUFFER_TAC and the \( Y \) buffer using PARBUF_TAC recursively, as shown in the following proof, where \( \Gamma \) is the multiset of all inputs that correspond to the conjuncts of \( Y \) (or simply \( Y^\perp \) if \( Y \) is atomic):

\[
\begin{array}{c}
\text{BUFFER_TAC} \quad \text{PARBUF_TAC} \\
\vdash X^\perp, X \quad \vdash \Gamma, Y \\
\vdash X^\perp, \Gamma, X \otimes Y
\end{array}
\]
As an example, the proof performed by PARBUF_TAC for parallel buffer \( \vdash A^\perp, (B \oplus C)^\perp, D^\perp, A \otimes (B \oplus C) \otimes D \) is the following:

\[
\begin{array}{lll}
\text{BUFFER_TAC} & \vdash (B \oplus C)^\perp, B \oplus C & \vdash D^\perp, D \\
\vdash A^\perp, A & \vdash (B \oplus C)^\perp, D^\perp, (B \oplus C) \otimes D \\
\vdash A^\perp, (B \oplus C)^\perp, D^\perp, A \otimes (B \oplus C) \otimes D \\
\end{array}
\]

Breaking down the steps performed by BUFFER_TAC, the full proof is the following:

\[
\begin{array}{lll}
\vdash B^\perp, B & \vdash C^\perp, C & \vdash D^\perp, D \\
\vdash B^\perp, B \oplus C & \vdash C^\perp, B \oplus C & \vdash D^\perp, (B \oplus C) \otimes D \\
\vdash A^\perp, A & \vdash (B \oplus C)^\perp, D^\perp, A \otimes (B \oplus C) \otimes D \\
\end{array}
\]

For the sake of completeness, the process calculus term extracted from this proof is the one shown below, where \( out \) is the output channel, whereas \( i_d, i_{bc}, \) and \( i_d \) are the input channels for \( A, B \oplus C, \) and \( D \) respectively:

\[
\begin{align*}
\bigotimes_{\text{out}} & (I_{i_d,x_0,y_0} \bigotimes_{\text{bc}} (L_{x_2,y_2}(I_{i_{bc},x_2}) \cdot R_{x_1}(I_{i_{bc},y_3})), I_{i_d,y_1})
\end{align*}
\]

This corresponds to the following \( \pi \)-calculus process:

\[
(\nu x_0,y_0) (\text{out}(x_0,y_0), (i_d(a) \cdot x_0(a)).0) \parallel \]

\[
(\nu x_1,y_1) (\overline{y_0}(x_1,y_1),(\nu u,v)(i_{bc}(u,v).u(x_2)).(\nu u,v)(i_{bc}(u,v).u(x_2)).(\nu u,v)(i_{bc}(u,v).u(x_2)).(\nu u,v)(i_{bc}(u,v).u(x_2))).0) + v(y_2) .
\]

\[
(\nu y_3)(x_1(u,v).\overline{y_3}(x_3).x_2(b).\overline{x_3}(b).0) + v(y_2) . \]

\[
(\nu y_3)(x_1(u,v).\overline{y_3}(x_3).x_2(b).\overline{x_3}(b).0) . y_3(c).0))) \parallel
\]

\[
i_d(d \cdot \overline{y_1}(d).0))) \quad (7.11)
\]

This completes our detailed analysis of the main support tactics set used to implement the high level, process composition actions that we describe next.

### 7.4 Process Composition Actions

As previously mentioned in Section 7.1, the core part of our composition methodology involves finding a CLL proof that the requested composite process is achievable, using
7.4. Process Composition Actions

the available processes as assumptions. MALL entailment is PSPACE-complete (Lincoln et al., 1992). Therefore, the possibility of constructing fully automated, effective proof procedures is questionable.

Besides, given the context of process composition, some control over the proof is desirable. For example, the ability to make conscious decisions about using one process (assumption) over another equivalent one or about the particular structure of the proof and, therefore, the corresponding composition is desirable. Such decisions can be affected by reasons outside the logical specification, such as non-functional qualities (e.g. location, cost, efficiency, etc.) or personal preference.

We therefore focus a small set of generic actions that a user needs when interactively composing processes. Each of these actions corresponds to a composition step, i.e. to some intuitive combination of two (atomic or composite) processes. For each of these steps we have constructed a custom, high level, fully-automated, HOL Light proof tactic that applies a number of CLL inference steps in order to generate the verified result at the logical level, thereby ensuring the action’s correctness. By using these actions the user has the freedom to make practical, process related decisions without worrying about the tedious process of applying a large number of primitive CLL rules since these are now automatically applied by each action.

An interface to trigger these actions can easily be implemented using mouse gestures on top of our diagrammatic notation (see Section 6.4). This results in an intuitive, purely graphical interface for process composition that emphasizes the information flow between the composed processes while completely hiding the underlying proof construction from the user. As such, the interface is usable and intuitive even without knowledge of CLL or theorem proving. To accomplish this, there is a natural tradeoff between having a high level interface and the flexibility of performing arbitrary, low level CLL proof steps. For this reason, we attempt to make the high level tactics as powerful as possible by minimizing the number of cases that can only be solved with low level proof steps.

We have considered 3 generic actions for the composition of two processes \( P \) and \( Q \):

- **Parallel** composition with the TENSOR action, where \( P \) and \( Q \) will be executed in parallel.
- **Optional** or conditional composition with the WITH action, where, depending on the input, either \( P \) or \( Q \) will be executed (as an equivalent to an “if-then-else”
Chapter 7. Process Composition via Proof

• **Sequential** composition with the JOIN action, where at least some of the outputs of \( P \) are connected to inputs of \( Q \).

The result of each composition step is a new (composite) process that can be further composed with other services. In the next sections, we describe the 3 tactics, including their implementation and visualisation, in more details.

### 7.4.1 The TENSOR Action

The TENSOR action corresponds to the parallel composition of two processes. This is particularly useful in cases where each of the components of a composite output of a processes needs to be handled by a different process. Composing these handlers in parallel allows all the involved outputs to be handled simultaneously.

At the diagrammatic level, the user can accomplish the TENSOR action with the simple gesture that involves clicking on a process and then right-clicking on another. In the interface this creates a new outer box, representing the parallel composition of the original processes into a new, composite process, whose inputs and outputs correspond to those of the original processes. Note that this new box can be collapsed, thus hiding its components, in order to make the diagram more concise.

In the CLL representation the TENSOR action can be easily verified based on the appropriate application of the tensor (\( \otimes \)) inference rule. The diagrammatic representation and proof tree for a simple example involving the TENSOR action on two services \( P \) and \( Q \) are given Fig. 7.3.

### 7.4.2 The WITH Action

The concept of the WITH action is analogous to the TENSOR action. It corresponds to the optional composition of two processes. This type of composition is useful in cases where each of the components of an optional output of a process needs to be handled by a different process.

For example, assume a process \( S \) has an optional output \( A \oplus C \) where \( C \) is an exception. We want \( A \) to be handled by some process \( P \), such as the one specified by \( \vdash A^\perp, B^\perp, X \),
while another process \( Q \) specified as \( \vdash C \perp, D \perp, Y \) plays the role of the exception handler for exception \( C \). For this to happen, we need to compose \( P \) and \( Q \) together using the \textsc{with} action so that \( A \oplus C \) from \( S \) can be dealt with in one go.

More generally, given a process \( \mathcal{P} \) defined as \( \vdash \Gamma_{\mathcal{P}}, A \perp, X \) and a process \( \mathcal{Q} \) defined as \( \vdash \Gamma_{\mathcal{Q}}, C \perp, Y \), where \( \Gamma_{\mathcal{P}} \) and \( \Gamma_{\mathcal{Q}} \) are arbitrary (possibly empty) multisets of inputs, their optional composition on inputs \( A \) and \( C \) can be viewed as the construction of an \textit{if-then-else} statement where if \( A \) is provided then \( \mathcal{P} \) will be executed, else if \( C \) is provided then \( \mathcal{Q} \) will be executed.

At the diagrammatic level, the user accomplishes the \textsc{with} action that will compose \( \mathcal{P} \) and \( \mathcal{Q} \) by clicking on input \( A \) of \( \mathcal{P} \) and then right-clicking on input \( C \) of \( \mathcal{Q} \). In the interface, a new collapsible, dashed, outer box is then created, representing a new composed process where either \( \mathcal{P} \) or \( \mathcal{Q} \) will be executed (contrast this to the solid outer box of the \textsc{tensor} action which, as explained before, consists of processes that run in parallel).

Given the processes \( P \) and \( Q \) from the initial example, i.e. where \( \Gamma_{\mathcal{P}} = \{B \perp\} \) and \( \Gamma_{\mathcal{Q}} = \{D \perp\} \), the diagrammatic representation of the application of the \textsc{with} action and the corresponding generated proof tree are shown in Figure 7.4.

The verification of the \textsc{with} action is more complicated than for \textsc{tensor}. It relies on the use of the \& rule of CLL (hence the name of the action) and more specifically the \&\textsc{proc} rule (see Section 7.2.3):

\[
\vdash \Gamma, A \perp, X \quad \vdash \Gamma, C \perp, Y \\
\vdash \Gamma, (A \oplus C) \perp, X \otimes Y \quad \&\textsc{proc}
\]
Figure 7.4: The proof tree and diagrammatic representation of the \textbf{WITH} action on input \( A \) of process \( P \) and input \( C \) of process \( Q \). A new outer box is created with optional input \( (A \oplus C) \) allowing either \( P \) or \( Q \) to be executed.
The particularity of this rule is that the context $\Gamma$, i.e. all the inputs except the ones involved in the WITH action, must be the same for both the involved processes. In the process interpretation, this means we need to account for unused resources from other inputs. In the example above, $P$ has apart from input $A\bot$ another input $B\bot$ which is missing from $Q$. The service formed by the conditional composition of $P$ and $Q$ will also have to have this input. Therefore, if exception $C$ occurs and $B$ is provided by another source, $B$ will not be consumed (since $P$ will not be invoked). Thus, it needs to be buffered through together with the output $Y$ of $Q$ that will handle $C$.

At the level of the logic, in order to apply the $\&$ proc rule to processes $P$ and $Q$, we need to minimally adjust the contexts $\Gamma^P$ and $\Gamma^Q$ of the two processes so that they end up being the same $\Gamma = \Gamma^P \cup \Gamma^Q$. Notice that we use multiset union ($\cup$) instead of multiset sum ($\oplus$) so that common inputs of $P$ and $Q$ are shared in the resulting composition.

In order to accomplish this for process $P$, we calculate the multiset of “missing” inputs $\Gamma^c_P = \Gamma^Q \setminus \Gamma^P$ so that $\Gamma = \Gamma^P \oplus \Gamma^c_P$. In the example of Figure 7.4, we obtain $\Gamma^c_P = \{D\}$. We then construct a parallel buffer (see Section 7.3.2) of type $\otimes_i \Gamma^c_P$ where $\otimes_i \{a_1, a_2, ..., a_n\} = a_1^\bot \otimes a_2^\bot \otimes ... \otimes a_n^\bot$ (notice the use of negation to convert every input $a_i$ to an output). In the example, this corresponds to the atomic buffer $\vdash D^\bot$, $D$. We remind the reader that parallel buffers can easily be proven/constructed using PARBUF_TAC. The parallel composition between this buffer and the process $P$ results in the following process:

$$\vdash \Gamma^P, \Gamma^c_P, A^\bot, X \otimes (\otimes_i \Gamma^c_P)$$

Symmetrically, we obtain the following process from $Q$:

$$\vdash \Gamma^Q, \Gamma^c_Q, C^\bot, Y \otimes (\otimes_i \Gamma^c_Q)$$

The $\&$ proc rule is now applicable because $\Gamma = \Gamma^P \cup \Gamma^c_P = \Gamma^Q \cup \Gamma^c_Q$. The final result is the following process:

$$\vdash \Gamma, (A \oplus C)^\bot, \left( X \otimes (\otimes_i \Gamma^c_P) \right) \oplus \left( Y \otimes (\otimes_i \Gamma^c_Q) \right)$$

Notice that the output $X$ of $P$ has now been paired with the buffered resources $\Gamma^c_P$ that
are only used by \( Q \) and are not consumed by \( P \), and symmetrically \( Y \) has been paired with the unused inputs of \( P \).

The corresponding proof for our example of \( P \) and \( Q \) is shown in Figure 7.4c.

This result and the complexity of the way it is constructed in CLL are representative of the unique quality of CLL which is able to provide systematic management of resources and, more specifically, a systematic accounting of unused resources.

### 7.4.3 The JOIN Action

The JOIN action is perhaps the most intuitive one when composing processes, but also the most complicated one to handle at a high level in CLL because of all the different possible cases. It reflects the connection of two processes in sequence, i.e. where (some of) the outputs of a service are connected to the corresponding inputs of another. At its core, the JOIN action relies on the use of the Cut rule in CLL:

\[
\vdash \Gamma, A \vdash \Delta, A^\perp, X \quad \text{Cut}
\]

In this, a process \( P \) with specification \( \vdash \Gamma, A \), i.e. with some context (multiset of inputs) \( \Gamma \) and output \( A \), is connected in sequence with a process \( Q \) with specification \( \vdash \Delta, A^\perp, X \), i.e. with an input \( A \), output \( X \), and (possibly) more inputs in context \( \Delta \). Notice the variation from the Cut rule as given in Figure 4.2. In this case, we make the output \( X \) of \( Q \) explicit since, based on the restrictions introduced in Section 6.2, \( Q \) has exactly one output.

Diagrammatically, the user can accomplish this by clicking on an edge that corresponds to one of the outputs of \( P \) and then right-clicking on an edge that corresponds to a matching input of \( Q \). The JOIN action then attempts to connect the output of \( P \) to the inputs of \( Q \) maximally. Depending on the complexity of \( A \) (which may not be atomic), the functionality of the JOIN action may become fairly complicated in order to produce a result that is intuitively expected by the user. In what follows, we present different cases for \( A \), the technical details of how each case is handled, and a visualised example for each case.
7.4.3.1 Atomic Output

In the simple case where $A$ is atomic, a straightforward use of the *Cut* rule is sufficient to connect the two processes.

For example, consider the $\text{JOIN}$ action between process $P$ specified by $\vdash A \perp, B \perp, X$ and process $Q$ specified by $\vdash X \perp, Z$. The result paired with the constructed proof tree is shown in Figure 7.5.

![Diagram of JOIN action](image)

$$
\begin{align*}
\vdash A \perp, B \perp, X & \quad \vdash X \perp, Z \\
\vdash A \perp, B \perp, Z & \quad \text{Cut}
\end{align*}
$$

Figure 7.5: The JOIN action between service $P$ with an atomic output and service $Q$ with a matching input.

Note that this approach works more generally for any non-atomic $A$ as long as an input of type $A \perp$ exists in $Q$.

7.4.3.2 Parallel Output

If $A$ is a parallel output, such as $B \otimes C$, then we need to construct an input of type $(B \otimes C) \perp$ for $Q$ so that we can apply the *Cut* rule as described above. We consider the following cases:

- If $Q$ has both inputs $B \perp$ and $C \perp$, i.e. $Q$ is of the form $\vdash \Delta, B \perp, C \perp, X$, then we can use the $\gamma \text{in}$ rule (see Section 7.2.1) to combine the two inputs as follows:

  $$
  \begin{align*}
  \vdash \Delta, B \perp, C \perp, X & \quad Q \\
  \vdash \Delta, (B \otimes C) \perp, X & \quad \gamma \text{in}
  \end{align*}
  $$
An example of the \textbf{JOIN} action between process $P$ specified by $\vdash A^\bot, B^\bot, X \otimes Y$ and process $Q$ specified by $\vdash X^\bot, Y^\bot, Z$ using our diagrammatic interface and the corresponding proof tree are shown in Figure 7.6.

\begin{align*}
\frac{\vdash A^\bot, B^\bot, X \otimes Y}{\vdash A^\bot, B^\bot, Z} & \quad \frac{\vdash X^\bot, Y^\bot, Z}{\vdash (X \otimes Y)^\bot, Z} & \text{\textit{Join}} \\
\frac{}{\vdash \Delta, B^\bot, X \otimes C} & \quad \frac{}{\vdash \Delta, B^\bot, C^\bot, X} \quad \frac{\vdash C^\bot, C}{\vdash \Delta, B^\bot, X \otimes C} & \text{\textit{Cut}}
\end{align*}

(c) Proof Tree

Figure 7.6: The JOIN action between service $P$ with a composite output and service $Q$ with two matching inputs.

- If $Q$ has only one of the two inputs, for example $B^\bot$, i.e. $Q$ is of the form $\vdash \Delta, B^\bot, X$ and $C^\bot \not\in \Delta$, then $Q$ is not capable of handling the second resource $C$. Instead, $C$ must be forwarded by a buffer. In this case we use the $\otimes \text{buf} R$ rule (see Section 7.2.5) as follows:

\begin{align*}
\frac{}{\vdash \Delta, B^\bot, C^\bot, X} & \quad \frac{\vdash C^\bot, C}{\vdash \Delta, B^\bot, X \otimes C} & \text{\textit{buf} R}
\end{align*}

Notice the use of \textsc{Buffer_Tac} (see Section 7.3.1) to prove and discharge the subgoal involving the introduced buffer of arbitrarily complex type $C$. We can then use the $\gamma \text{in}$ rule in the same way as in the previous case to construct an input of type $(B \otimes C)^\bot$. Symmetrically, if $Q$ had an input $C^\bot$ (i.e. the right-hand side of $B \otimes C$) and no input $B^\bot$ we would use the $\otimes \text{buf} L$ rule (in order to buffer the left-hand side of $B \otimes C$).

An example of the \textbf{JOIN} action between process $P$ specified by $\vdash A^\bot, B^\bot, X \otimes Y$ and process $Q$ specified by $\vdash X^\bot, C^\bot, Z$ using our diagrammatic interface and the corresponding proof tree are shown in Figure 7.7.
### 7.4. Process Composition Actions

#### 7.4.3.3 Optional Output

If $A$ is an optional output, such as $B \oplus C$, then, similarly to the parallel case, we need to construct an input of type $(B \oplus C)^\perp$ for process $Q$. We assume process $Q$ can handle $B$, i.e. $Q$ is of the form $\vdash \Delta, B^\perp, X$, which means the second option $C$ needs to be handled by buffers. To accomplish this, we create an optional composition between $Q$ and a parallel buffer of type $\otimes_i^\perp (\Delta \uplus C)^\perp$ where $\otimes_i^\perp \{a_1, a_2, ..., a_n\} = a_1^\perp \otimes a_2^\perp \otimes ... \otimes a_n^\perp$ (notice the use of negation to convert inputs to outputs) using the $\& \ proc$ rule (see Section 7.2.3) as follows:

$$
\frac{
\vdash \Delta, B^\perp, X \quad \vdash \Delta, C^\perp, \otimes_i^\perp (\Delta \uplus C^\perp)
}{
\vdash \Delta, (B \oplus C)^\perp, X \oplus \left( \otimes_i^\perp (\Delta \uplus C^\perp) \right)} \quad \& \ proc
$$

(7.12)

Notice the use of $\texttt{PARBUF\_TAC}$ (see Section 7.3.2) to prove and discharge the subgoal involving the introduced parallel buffer. The $\& \ proc$ rule can also be used symmetrically to construct input $(B \oplus C)^\perp$ when $Q$ has an input $C^\perp$.

Similarly to the $\texttt{WITH}$ tactic, the particular structure of the CLL rules (here in partic-
ular the &proc rule) ensures the systematic management of unused resources. More specifically, among the two optional inputs $B$ and $C$, if $C$ is received then process $Q$ will never be executed. As a result, any extra resources that are provided as part of the context $\Delta$ of $Q$ will remain unused and need to be buffered together with $C$. This the reason behind the type $\otimes_i^{\perp}(\Delta \oplus C^{\perp})$ of the constructed buffer as opposed to a simpler, perhaps more intuitive type $C$.

An example of the JOIN action between process $P$ specified by $\vdash A^{\perp}, B^{\perp}, X \oplus E$ and process $Q$ specified by $\vdash X^{\perp}, C^{\perp}, Z$ using our diagrammatic interface and the corresponding proof tree are shown in Figure 7.8.

Based on the functionality described above, it is interesting to consider the case where $\Delta = \{\}$, i.e. where $Q$ only has one input $B$ (or, symmetrically, only one input $C$). This simplifies the constructed proof tree (7.12) as follows:

Figure 7.8: The JOIN action between service $P$ with an optional output and service $Q$ with a matching input.
The proof tree can be further simplified in specific cases of the output \( X \) of \( Q \). More specifically, we consider the following 2 special cases of \( X \):

- If \( X = C \) then the proof construction \( (7.13) \) produces the composition \( \vdash (B \oplus C)\perp, C \perp C \). Given the idempotency of the \( \oplus \) operator, i.e. \( \vdash (C \oplus C)\perp, C \), we can simplify this to \( \vdash (B \oplus C)\perp, C \) by using the &in rule (see Section 7.2.2) instead of \&proc as shown below:

\[
\frac{\vdash B\perp, C \quad \vdash C\perp, C \quad \&\text{in}}{\vdash (B \oplus C)\perp, C}
\]

The example of the JOIN action between process \( P \) specified by \( \vdash A\perp, B\perp, X \oplus Y \) and process \( Q \) specified by \( \vdash X\perp, Y \) is presented in Figure 7.9.

- If \( X = D \oplus C \) (or symmetrically \( X = C \oplus D \)) for some \( D \) then the constructed proof can also be simplified. Proof \( (7.13) \) will produce the composition \( \vdash (B \oplus C)\perp, D \oplus C \perp C \) which can be simplified to \( \vdash (B \oplus C)\perp, D \perp C \). The simplification

---

**Figure 7.9:** The JOIN action between service \( P \) with an optional output \( X \oplus Y \) and service \( Q \) that only has \( X \) as input and \( Y \) as output.
can be accomplished using one of the &buf rules (see Section [7.2.6]). In this particular case, we need to add C to the right of input B and C already exists in the right hand side of the output D ⊕ C. Therefore we will use the &RbufR rule as follows:

\[
\frac{}{B^\perp, D \oplus C} \rightarrow \frac{C^\perp, C}{} & \text{RbufR}
\]

The example of the JOIN action between process P specified by \( \vdash A^\perp, B^\perp, X \oplus Y \) and process Q specified by \( \vdash X^\perp, C \oplus Y \) is presented in Figure 7.10.

![Diagram](image)

Figure 7.10: The JOIN action between service P with an optional output X ⊕ Y and service Q that only has X as input and C ⊕ Y as output.

Our JOIN tactic first checks for the two special cases presented above before attempting the more general proof strategy for an optional output.

### 7.4.3.4 Composite Output

Finally, the output A of P can be a complex combination of multiple parallel and/or optional outputs. In that case, we apply the above proof strategies in a recursive, bottom-up way.

For example, if \( A = B \otimes (C \oplus D) \) and Q has an input \( C^\perp \), then we will first use the proof strategy from Section [7.4.3.3] to construct the optional input \((C \oplus D)^\perp\) and then
the proof strategy from Section 7.4.3.2 to construct $(B \otimes (C \oplus D))^\perp$.

7.5 Workflow Properties

The structure of the process workflows constructed using our introduced, logic-based composition access demonstrates some very interesting properties. In this section, we discuss some of these properties that emerge when considering compositions of more than two processes. In Section 7.5.1 we discuss compositionality and introduce the notion of a composition strategy. Next, in Section 7.5.2 we give an example to demonstrate the concurrent nature of the workflows, whereas in Section 7.5.3 we discuss the execution of conditional workflows.

7.5.1 Compositionality and Composition Strategies

The three composition actions that we described in the previous section, TENSOR, WITH, and JOIN, provide the three fundamental ways of composing two processes in parallel, conditionally, and sequentially respectively, and construct a new composite process. The involved components can have any logical specification and there is no distinction between atomic or composite processes. This allows a large degree of compositionality and several levels of granularity in the process model. We can construct high level process workflows where each component is a black box that can be further decomposed to another workflow with multiple components, which in turn can have complicated structures and so on.

The general composition methodology is to connect component processes (which we view as atomic, black boxes even if they are composite) as building blocks using the three actions. Each action constructs a new building block that can be combined with others using the same actions. We continue building increasingly complicated compositions until we reach our preset goal or in the case of the discovery mode until we are satisfied with the constructed workflow (see Section 7.1). The result can then be packaged and stored as a new available process.

Although this procedure may seem intuitive and straightforward, it has some important implications to the structure of the process workflows that we are constructing. More specifically, the order of application of the actions, or what we refer to as composition
strategy, implicitly affects the structure of the composition. Even though the visualised workflow may even appear the same for two different strategies in our current implementation, meaning that the information flow in the composite process will follow the same pattern, the structure, efficiency, and order of execution of the two constructed workflows may differ.

To demonstrate this, we revisit the credit card example as described in Section 2.1.3. We introduced the 3 available processes for this example, namely CreditCardInit, UserPINInput, and CreditCardTransaction, using our diagrammatic notation in Section 6.4. These are shown below in Figure 7.11a.

The three processes can be composed using two different strategies: we can either use the JOIN tactic to join the output PIN of UserPINInput with the PIN input of CreditCardTransaction and then the output PIN_REQ of CreditCardInit with the PIN_REQ input of (the now connected) UserPINInput (Strategy A) or, alternatively, we could join PIN_REQ first and then PIN (Strategy B). We demonstrate the two strategies graphically in Figure 7.11. Both strategies result in a verified composition that satisfies the requested specification and have the same final diagrammatic representation, as shown in Figure 7.11d.

However, proof-wise the two strategies do not have identical results. The order of application of actions changes the structure of the underlying CLL proof that is being constructed automatically. The proofs, shown in Figure 7.12, while equivalent are not identical.

Each of these two proofs yields a verified π-calculus term that conforms to the requested specification of the composite process. The two π-calculus terms extracted from the proofs are shown in Figure 7.13. The structure of each proof affects the π-calculus term being constructed. For example, Strategy B uses the CLL identity axiom for the TRANSACTION output of CreditCardInit. This introduces an extra buffer, shown in bold, in the resulting π-calculus term.

Therefore, following different composition strategies affects the structure and, to a certain degree, the efficiency of the resulting workflow.
Figure 7.11: The two composition strategies for the credit card example visualised diagrammatically. Both strategies have the same initial state and the same result.
component services CreditCardInit, UserPINInput, and CreditCardTransaction are abbreviated as CCI, UPI, and CCT respectively.

Figure 7.12: The CLL proof scripts constructed by the CreditCardPayment composition using Strategy A (7.14) and Strategy B (7.15).

(7.15a)

(7.15b)

(7.14)
\[ \text{CreditCardPayment}(\text{in, out}) \equiv \]
\[ (\nu z') (\text{CreditCardInit}(z', \text{in}) \parallel z'(\text{ctt}_{tr}, \text{upi}_{req}). \]
\[ (\nu z) (\text{UserPINInput}(z, \text{upi}_{req}) \parallel \text{CreditCardTransaction}(\text{out}, \text{ctt}_{tr}, z)) ) \] (7.16)

(a) Strategy A

\[ \text{CreditCardPayment}(\text{in, out}) \equiv \]
\[ (\nu z') ( (\nu z) (\text{CreditCardInit}(z, \text{in}) \parallel z(\text{ibuf}, \text{upi}_{req}). \]
\[ (\nu o_{buf}, \text{upi}_{out}) (\text{o}_{buf}(o_{buf}, \text{upi}_{out}). (\]
\[ \text{ibuf}(\text{tr}) \cdot \text{obuf}(\text{tr}) \cdot 0 \ || \ \text{UserPINInput}(\text{upi}_{out}, \text{upi}_{req})) ) ) \parallel \]
\[ z'(\text{ctt}_{tr}, \text{ctt}_{pin}).\text{CreditCardTransaction}(\text{out}, \text{ctt}_{tr}, \text{ctt}_{pin}) \] (7.17)

(b) Strategy B

Figure 7.13: The resulting \( \pi \)-calculus terms for the CreditCardPayment composition for each of the two possible strategies. Marked in bold is the extra axiom buffer introduced in Strategy B.
7.5.2 Concurrency

Using the \( \pi \)-calculus as the language to describe our composed workflows results in a naturally concurrent execution. In fact, as mentioned in Section 6.3, there is no explicit synchronisation in the \( \pi \)-calculus specification of the processes, which essentially means the entire workflow can be executed in parallel, completely asynchronously. The proofs-as-processes paradigm guarantees that the workflow execution will never block either with a deadlock (two or more processes waiting for input from each other) or livelock (two or more processes endlessly feeding information to each other). This is a natural consequence of the paradigm, since process interactions correspond to \( \pi \)-calculus reductions, and performing such reductions in turn corresponds to the terminating procedure of cut eliminations in CLL proofs.

Taking a step back from the introduced \( \pi \)-calculus specifications of the component processes, we consider a more pragmatic approach where these processes receive all their inputs before providing their outputs (input synchronisation) and the output may take a considerable amount of time to be produced. Note that these conditions cannot be strictly imposed on the \( \pi \)-calculus terms, but are, in fact, imposed in the generated executable code (see Section 8.2.3.4).

Even with these restrictions, workflows constructed based on the proofs-as-processes paradigm maintain a high degree of concurrency in their execution and never block unless one of the component processes blocks internally. Ordering constraints are only implicitly imposed between processes that are composed in sequence because of the input synchronisation occurring within the receiving process. This is primarily attributed to the fact that the \( \pi \)-calculus reductions that correspond to the workflow execution enforce a synchronisation of communication channels rather than a synchronisation of information.

Let us demonstrate this through an example. Consider process \( P \) with specification \( \vdash X \perp, A \otimes B \), process \( Q \) with specification \( \vdash A \perp, C \), and process \( R \) with specification \( \vdash B \perp, D \), i.e. \( P \) has two outputs \( A \) and \( B \) which are each handled by \( Q \) and \( R \) respectively. One strategy for composing these processes is to use the JOIN action between \( P \) and \( Q \) in order to connect \( A \) and then a second JOIN action between the result and \( R \) in order to connect \( B \). The result of the first JOIN action is a process with specification \( \vdash X \perp, C \otimes B \):
7.5. Workflow Properties

\[
\begin{array}{c}
\vdash X^\perp, A \otimes B \\
\vdash A^\perp, C \\
\vdash B^\perp, B \\
\vdash X^\perp, A \otimes B \\
\vdash A^\perp, C \otimes B \\
\vdash (A \otimes B)^\perp, C \otimes B \\
\vdash X^\perp, C \otimes B \\
\end{array}
\]

(7.18)

The action introduces a buffer of type \( B \) because \( \mathcal{Q} \) can only handle \( A \), and thus output \( B \) of \( \mathcal{P} \) is buffered through. The output \( C \) of \( \mathcal{Q} \) is now “synchronised” (via the \( \otimes \) rule) with the output \( B \) of the buffer.

Similarly, the second \( \text{JOIN} \) action introduces a buffer of type \( C \) because \( \mathcal{R} \) can only handle \( B \). The input \( B^\perp \) of \( \mathcal{R} \) is “synchronised” (via the \( \in \) rule) with the unput \( C^\perp \) of the buffer:

\[
\begin{array}{c}
\vdash C^\perp, C \\
\vdash B^\perp, D \\
\vdash C^\perp, B^\perp, C \otimes D \\
\vdash (C \otimes B)^\perp, C \otimes D \\
\vdash X^\perp, C \otimes D \\
\end{array}
\]

(7.19)

Therefore, based on the constructed “synchronisations”, process \( \mathcal{Q} \) (in conjunction with the buffer of type \( B \)) will be executed before process \( \mathcal{R} \) in order to communicate an output of type \( C \otimes B \). However in practice, process \( \mathcal{R} \) may still execute before \( \mathcal{Q} \) finishes!

This becomes clearer if we inspect the corresponding \( \pi \)-calculus term. The process translation of proof (7.18) generated by the first \( \text{JOIN} \) action is the following process \( \mathcal{S} \):

\[
\mathcal{S} = \text{Cut}^z \left( \mathcal{P}, \bigotimes_{o_b} (\mathcal{Q}, I_{b,o_b} : o_q, o_b : (Q, I_{b,o_b})) \right) \quad (7.20)
\]

where \( i_q \) and \( o_q \) the input and output channels of \( \mathcal{Q} \) respectively, \( I_{b,o_b} \) the axiom buffer of type \( B \), and the rest of the channels are newly introduced in the proof. This corresponds to the following \( \pi \)-calculus term:

\[
\mathcal{S} = (\nu z) \left( \mathcal{P} [z/o_p] \parallel z^d_b (i_q, i_b) . (\nu o_q, o_b) (\mathcal{Q} \parallel i_b (b) . o_b (b) . 0) \right) [z/z^d_b] \quad (7.21)
\]
Assuming the communication with $P$ (corresponding to the *Cut* step in the proof) is executed, process $S$ will be reduced to the following term:

$$S = (\nu o_q, o_b) \left( o_q \langle o_q, o_b \rangle . (Q \parallel i_b \langle b \rangle . \overline{a_b} \langle b \rangle . 0) \right)$$  (7.22)

The process translation of proof (7.19) generated by the second $\textsf{JOIN}$ action has the following form:

$$\text{Cut}^z \left( S, Z_i, i_r, \left( o_c, o_r \parallel \bigotimes_{z_h} (I_{i_c, o_r} \parallel R) \right) \right)$$  (7.23)

where $o_s$ the output channel of $S$, $i_r$ and $o_r$ the input and output channels of $R$ respectively, $I_{i_c, o_r}$ the axiom buffer of type $C$, and the rest of the channels are newly introduced in the proof.

This is equivalent to the following $\pi$-calculus term:

$$(\nu z) \left( S[z/o_s] \parallel Z_i \parallel Z_h (i_c, i_r) . (\nu o_c, o_r) \left( \overline{o_q} \langle o_q, o_r \rangle . (i_c \langle c \rangle . \overline{a_c} \langle c \rangle . 0 \parallel R) \right \parallel \right) [z/z_h]$$  (7.24)

Replacing with $S$ from (7.22) and applying substitutions we obtain:

$$(\nu z) \left( (\nu o_q, o_b) \left( Z \langle o_q, o_b \rangle . (Q \parallel i_b \langle b \rangle . \overline{a_b} \langle b \rangle . 0) \right) \right \parallel Z (i_c, i_r) . (\nu o_c, o_r) \left( \overline{o_q} \langle o_q, o_r \rangle . (i_c \langle c \rangle . \overline{a_c} \langle c \rangle . 0 \parallel R) \right \parallel \right)$$  (7.25)

Notice that the two parts from the two sub-proofs can now communicate through channel $z$ (corresponding to the *Cut* step in proof (7.19)). This will synchronize channels $o_q$ and $o_b$, i.e. the output channel of $Q$ with the output channel of the buffer of type $B$. This, however, does not enforce $Q$ to be executed yet. This is because it is the communication channels that are being synchronized and not the information itself. The reduction gives us the following result:

$$Q \parallel i_b \langle b \rangle . \overline{a_b} \langle b \rangle . 0 \parallel (\nu o_c, o_r) \left( \overline{o_q} \langle o_q, o_r \rangle . (o_q \langle c \rangle . \overline{a_c} \langle c \rangle . 0 \parallel R[ o_b/i_r]) \right \parallel \right)$$  (7.26)

Pending the synchronization of output channels $o_c$ and $o_r$ of the $C$ buffer and $R$ respectively (which will occur when the output $C \otimes D$ is communicated to another receiving
process), \( Q \) will be able to communicate with the \( C \) buffer, and the \( B \) buffer will be able to communicate with \( R \) independently. In fact since buffers do not affect the carried information but merely forward it through a different channel, \( Q \) and \( R \) are essentially executed in parallel, with \( Q \) the first to be initiated.

Therefore, we observe that the \( \pi \)-calculus terms generated by the proofs-as-processes paradigm only synchronise the involved communication channels independently of the execution of the individual process components. This allows the workflow to be executed concurrently and asynchronously, without unnecessary dependencies or ordering constraints between the component processes, and independently from the composition strategy used to generate the composition.

### 7.5.3 Conditional Execution

The execution of conditional workflows created by our proofs-as-processes based approach via the \( \text{WITH} \) (see Section 7.4.2) and \( \text{JOIN} \) actions (see Section 7.4.3.3) ensures that no processes will be executed in vain. Only the processes that correspond to the part of the workflow that satisfies the runtime conditions will be executed. In fact, based on the composition strategy used, the user has control over which processes will be executed under each condition. We review the two cases of constructing conditional compositions separately next.

**Conditional via the \( \text{WITH} \) action.** For the case of a conditional composition using the \( \text{WITH} \) action, let us review the example composition from Figure 7.4 which composes process \( P \) specified by \( \vdash A^\perp, B^\perp, X \) and process \( Q \) specified by \( \vdash C^\perp, D^\perp, Y \). The result is a process \( S \) with specification \( \vdash (A \oplus C)^\perp, B^\perp, D^\perp, (X \otimes D) \oplus (Y \otimes B) \).

As previously explained in Section 7.4.2 this corresponds to a statement of the form if \( A \) then \( P \) else if \( C \) then \( Q \). As a result we expect either \( P \) or \( Q \) to be executed, but never both.

The process calculus term for \( S \) generated by the proof in Figure 7.4 is the following:

\[
S = \bigwedge_{\alpha_x} L_a \left( \bigotimes_{\alpha_d} (P, I_{id,od}) \right) \bigg|_{\alpha_y} \bigg|_{\alpha_d} R_y \left( \bigotimes_{\alpha_d} (Q, I_{ib,ob}) \right)
\]

(7.27)

where \( i_a \) the input channel for type \( A \) of \( P \), \( \alpha_x \) the output channel for type \( X \) of \( P \), \( i_c \) the input channel for type \( C \) of \( Q \), \( \alpha_y \) the output channel for type \( Y \) of \( Q \), \( I_{id,od} \) and \( I_{ib,ob} \) the axiom buffers for types \( D \) and \( B \) respectively, \( \alpha_d \) the output channel for the
resulting process $S$, and the rest of the channels are newly introduced in the proof. This corresponds to the following $\pi$-calculus term:

$$
S = (\nu u,v)\left( \frac{z_d}{\nu z} (u,v). (u\langle i_a \rangle). \\
(\nu z_d^a)\left( o_s (u,v). \overline{z_d^a}. (\nu o_x, o_d)\left( \frac{z_d}{\nu z} (o_x, o_d). (P \parallel i_d\langle d \rangle. \overline{x_d\langle d \rangle.0}) \right) + v\langle i_c \rangle. \\
(\nu z_b^o)\left( o_s (u,v). \overline{z_b^o}. (\nu o_y, o_b)\left( \frac{z_b}{\nu z} (o_y, o_b). (Q \parallel i_b\langle d \rangle. \overline{x_b\langle d \rangle.0}) \right) \right) \right) \right) \right) \right) \right) \right) \right)
$$

(7.28)

In short, process $P$ is composed with a $D$ buffer to form subterm $\frac{z_d}{\nu z} (o_x, o_d)\left( (P, I_{i_d\langle d \rangle}) \right)$ of (7.27) and process $Q$ with a $B$ buffer for subterm $\frac{z_b^o}{\nu z} (o_y, o_b)\left( (Q, I_{i_b\langle d \rangle}) \right)$ of (7.27). These subterms are then composed with the $\&$ rule to form $S$. This rule creates a $\pi$-calculus sum between the two symmetrical terms as shown in (7.28) (second line).

According to the reduction rules of the $\pi$-calculus (see Section 3.2), only one of the two summands will be reduced, whereas the other will disappear without being executed. As a result, only $P$ or $Q$ will be executed depending on the input received through channel $z_d^a$, but never both.

- **Conditional via the JOIN action.** For the case of a conditional composition constructed via the JOIN action, we revisit the example from Figure 7.8 where we join process $P$ specified by $\bot A^\bot$, $B^\bot$, $X \oplus E$ with process $Q$ specified by $\bot X^\bot$, $C^\bot$, $Z$ to form composite process $S$ specified by $\bot A^\bot$, $B^\bot$, $C^\bot$, $Z \oplus (C \otimes E)$. Since $Q$ can only handle $X$, in the case where $P$ produces $E$, $Q$ should not be executed. This can be seen by inspecting the process calculus term for $S$ generated by the proof in Figure 7.8 which is the following:

$$
S = \text{Cut}^z \left( \frac{z_d}{\nu z} (o_x) \left( \frac{z_b}{\nu z} (o_y) \left( L(Q), \overline{z_b^o}, (I_{i_c\langle d \rangle}, I_{i_e\langle e \rangle}) \right) \right) \right) \right)
$$

(7.29)

where $o_p$ the output channel of $P$, $i_x$ the input channel for type $X$ of $Q$, $o_z$ the output channel for type $Z$ of $Q$, $I_{i_c\langle d \rangle}$ and $I_{i_e\langle e \rangle}$ the axiom buffers for types $C$ and $E$ respectively, $o_s$ the output channel for the resulting process $S$, and the rest of the channels are newly introduced in the proof.

Notice the similarity of subterm $\frac{z_d}{\nu z} (o_x) \left( \frac{z_b}{\nu z} (o_y) \left( L(Q), \overline{x_b^o}, (I_{i_c\langle d \rangle}, I_{i_e\langle e \rangle}) \right) \right)$ with (7.27). Similarly to that case, the use of the $\&$ rule creates a $\pi$-calculus sum. As a result either
7.5. Workflow Properties

\[ o_T L_Q (Q) \text{ or } R_{\alpha_T} \left( \alpha_{c_o}, I_{c_i, o_i} \right) \]

will be executed, but not both, i.e. either \( Q \) will be executed or the two buffers \( I_{c_i, o_i} \) and \( I_{c_i, o_i} \) depending on the input through channel \( z_x \). This ensures \( Q \) will never be executed if \( E \) is provided by \( P \). Moreover, the unused resource \( C \) given to \( Q \) is also buffered in the final output, together with the unhandled \( E \).

As obvious as this observation may seem (\( Q \) could not possibly execute without all its inputs), things become slightly more complicated when considering the provider of \( C \). Assume a process \( T \) specified by \( \perp Y \downarrow, C \). This process can be used to provide \( C \) for \( Q \). In fact, it can be composed with \( S \) in a straightforward way using a \( \text{JOIN} \) action. The result of this action is shown in Figure 7.14.

The important question to ask is in which cases will \( T \) be executed at runtime. More specifically, if \( P \) provides \( E \) and \( Q \) is not executed, should \( T \) be executed or not? Since \( Q \) is never executed in this case, resource \( C \) will not be required, therefore the execution of \( T \) may or may not be necessary.

A good practical example of such a case is one where \( T \) is the \text{UserPINInput} process from the credit card example (see Figure 6.1) and \( C \) is the user’s PIN. On the one hand, if something went wrong during the transaction and an exception was thrown, we do not want the system to unnecessarily request the user’s PIN number. On the other hand, if the resulting process is further composed with an exception handler for exception \( E \) such that, given the user’s PIN can still proceed and complete the transaction, then we do \textbf{require} \text{UserPINInput} (\( T \)) to be executed.

In this situation, the user can control the behaviour of the workflow and whether \( T \) will be executed or not by exploiting compositionality and using the appropriate composition strategy (see Section 7.5.1). We remind the reader that every usage of a composition action can be viewed as the construction of a new composite process involving two components.

In the example of Figure 7.14, \( T \) is joined with \( S \). Therefore, \( T \) will execute in parallel with \( S \) and independently of the output of \( P \). This can easily be shown by investigating the process calculus term corresponding to the entire proof, which is the following:

\[
\text{Cut}^\zeta \left( T, \text{Cut}^\zeta \left( P, \tilde{\bigotimes}_z^c \left( L_Q (Q), \tilde{R}_{\alpha_T} \left( \alpha_{c_o}, I_{c_i, o_i} \right) \right) \right) \right) \tag{7.30}
\]

where \( o_T \) the output channel for type \( C \) of \( T \) and \( i_q \) the input channel for type \( C^\perp \) of \( Q \).
Chapter 7. Process Composition via Proof

(a) Before

(b) After

\[ \vdash Y \perp, C \]
\[ \vdash A \perp, B \perp, C \perp, Z \oplus (C \otimes E) \]

Cut

(c) Proof Tree

Figure 7.14: The JOIN action between service T and conditional composite service S.

In contrast, we can use the JOIN action between T and Q before joining process P to Q. This will create a new process Q’ specified by \( \vdash X \perp, Y \perp, Z \) as shown in Figure 7.15.

We can then use the JOIN action between P and Q’. The action will perform the same steps as the ones shown in Figure 7.8 but replacing C with Y and Q with Q’. The resulting process S’ will be specified by \( \vdash A \perp, B \perp, Y \perp, Z \oplus (Y \otimes E) \) and the corresponding process calculus term is the following:

\[
\text{Cut}^z \left( P, \bigotimes_{\alpha} L \left( \text{Cut}^z \left( T, Q \right) \right), \bigotimes_{\alpha} R \left( \bigotimes_{i_{c,0_e}} \left( i_{c,0_e} \right) \right) \right) \tag{7.31}
\]

Based on our previous analysis of the constructed π-calculus term (7.27), by replacing Q with process Q’ = Cut^z (T, Q) we can observe that if P provides an exception E then process Q’, i.e. the composition of both T and Q, will never be executed.

To sum up, the user can control the execution of a composite workflow by choosing the appropriate composition strategy while keeping in mind the compositionality of the result of each action performed.
7.5. Workflow Properties

(a) Before

(b) After

\[ \vdash Y \perp, C \quad \vdash X \perp, C \perp, Z \quad Q \]

\[ \vdash X \perp, Y \perp, Z \] \quad \text{Cut}

(c) Proof Tree

Figure 7.15: The JOIN action between process $T$ and process $Q$. 
7.6 The Ski Example Revisited

In this chapter so far, we have described our methodology to specify and compose processes using CLL theorem proving and the proofs-as-processes paradigm. Based on this, we revisit the Ski example as originally presented by Rao (see Section 5.2.1), in order to describe how the specification and composition is accomplished within the proofs-as-processes system.

We begin by describing the CLL and $\pi$-calculus specifications of the component services of the Ski example in Section 7.6.1. Then, in Section 7.6.2 we demonstrate the proof corresponding to the specification of the requested process GetSki, whereas the extracted $\pi$-calculus term is discussed in Section 7.6.3.

7.6.1 Specification

Based on the typing specifications of processes in CLL described in Section 6.2, we have created CLL specifications of the available services and the requested service in the Ski example. Each of these, paired to the corresponding diagrammatic representation, are presented in Figure 7.16. They can be contrasted to the specifications given by Rao et al. as shown in Figure 5.4.

We have also introduced abstract $\pi$-calculus specifications of the available services using the methodology described in Section 6.3. Moreover, we introduce $\pi$-calculus terms for a Request and a Response process. These simulate the user input to and the expected output from the resulting composition GetSki, respectively. The parallel composition of the Request and Response services and the derived composite service GetSki (see Figure 7.20) is introduced as the Main process to complete our model. All these specifications are shown in Figure 7.17.

7.6.2 Proof

As previously mentioned in Section 7.1, the GetSki composition can be constructed by proving the requested specification using the available services as assumptions (see Figure 7.16). Using the JOIN action described in Section 7.4.3 is sufficient to construct a sequential workflow for GetSki. In this case, we follow a right-to-left strategy which typically produces fewer buffers. The constructed proof is shown in Figure 7.18.
Available services:

\[ \text{SelectLength}(slh, slw, sll) :: slh:HEIGHT\_CM, \ slw:WEIGHT\_KG, \ sll:LENGTH\_CM } \]

\[ \text{Cm2Inch}(cic, cii) :: cic:LENGTH\_CM, \ cii:LENGTH\_IN } \]

\[ \text{Usd2Nok}(unu, unn) :: unu:PRICE\_USD, \ unn:PRICE\_NOK } \]

\[ \text{SelectModel}(smp, sms, smo) :: smp:PRICE\_LIMIT, \ sms:SKILL\_LEVEL, \ smo:BRAND \otimes \text{MODEL} } \]

\[ \text{SelectSki}(ssb, ssms, ssl, sso) :: ssb:BRAND, \ ssms:MODEL, \ ssl:LENGTH\_IN, \ sso:PRICE\_USD \otimes \text{EXCEPTION} } \]

Request:

\[ \text{GetSki}(x, y, z, w, t) :: x:PRICE\_LIMIT, \ y:SKILL\_LEVEL, \ z:HEIGHT\_CM, \ w:WEIGHT\_KG, \ t:PRICE\_NOK \oplus \text{EXCEPTION} } \]

Figure 7.16: The CLL specification and corresponding diagrammatic representation of the available services and the requested service for the Ski example.

\[
\text{SelectLength}(slh, slw, sll) = slh(hc).0 \parallel slw(wk).0 \parallel (v \ 1c)(sll(1c).0)
\]

\[
\text{Cm2Inch}(cic, cii) = cic(li).0 \parallel (v li)(cii(li).0)
\]

\[
\text{Usd2Nok}(unu, unn) = unu(pu).0 \parallel (v pn)(\text{unu}(pn).0)
\]

\[
\text{SelectModel}(smp, sms, smo) = smp(pl).0 \parallel sms(sl).0 \parallel (v smo_a, smo_b)(smp\langle smo_a, smo_b\rangle.(v br)(smp\langle br\rangle.0) \parallel (v mo)(smp\langle mo\rangle.0))
\]

\[
\text{SelectSki}(ssb, ssms, ssl, sso) = ssb(br).0 \parallel ssms(mo).0 \parallel ssl(li).0 \parallel (v sso_x, sso_y)(sso\langle sso_u, sso_v\rangle.(sso_x\langle sso_u\rangle.(v pu)(sso_y\langle pu\rangle.0) + sso_y\langle sso_y\rangle.(v exe)(sso_y\langle exe\rangle.0)))
\]

\[
\text{Request}(slh', smp', sms', slw') = (v hc)(slh'(hc).0) \parallel (v pl)(smp'(pl).0) \parallel (v sl)(sms'(sl).0) \parallel (v wk)(slw'(wk).0)
\]

\[
\text{Response}(t) = (v t_u, t_v)(t_u(t_x, t_v).t_x(t_x(pn).0 + t_v(t_y, t_y(exe).0)))
\]

\[
\text{Main()} = \text{Request}(slh, smp, sms, slw) \parallel \text{Response}(t) \parallel \text{GetSki}(smp, sms, slh, slw, t)
\]

Figure 7.17: The available services and the Request, Response, and Main processes for the Ski example defined as \( \pi \)-calculus processes.
Figure 7.18: The CLL proof of the requested service GetSki in the Ski example.
7.6.3 Result

The workflow diagram corresponding to our proof for the Ski example is presented in Figure 7.19 whereas the \( \pi \)-calculus term that is constructed based on the proofs-as-processes paradigm is shown in Figure 7.20. The latter is easily comparable to the result produced by Rao et al. from Figure 5.9, i.e. the following:

\[
(v_{x1})(smp, sms, slh, slw).((SelectModel || SelectLength).\overline{\text{cic}}.CM2INCH)
\]
\[
.(smbssb, smmssm, \overline{\text{cissl}}).SelectSki(ss\overline{\text{pnu}} + ss\overline{\text{ex}}_1).USD2NOK.unn + x_1
\]  (7.36)

The latter appears to be more readable and gives a clearer, immediate picture of the involved interactions between the component services. We remind the reader that, as discussed in Chapter 5, Rao’s syntax lacks formal semantics and, therefore, it is impossible to verify the correctness of their result. In contrast, our \( \pi \)-calculus result from Figure 7.20 has formally defined semantics and can be immediately executed following the formal rules for \( \pi \)-calculus reductions.

\[
\text{GetSki}(smp, sms, slh, slw, t) = (v z)((v z')(SelectLength(slh, slw, z')) ||
\text{Cm2Inch}(z', z)) ||(v \text{sso})(v \text{sмо})(SelectModel(smp, sms, smo) ||
\text{sмо}(ssb, ssm).SelSki(ssb, ssm, z, sso)) || (v u, v)(\overline{\text{сро}}(u, v).
(u(up).v unn)(y(up, vp).\overline{\text{р}}(unn).Usd2Nok(unu, unn)) +
\quad v(c).v d)(y(uq, vq).\overline{\text{c}}(d).c(ex).\overline{\text{d}}(ex).0))))
\]  (7.37)

When observing the constructed \( \pi \)-calculus term, it is immediately apparent that, even for a small example of 5 simple components, the complexity of the \( \pi \)-calculus term is
fairly high and prohibits any attempt to verify its functionality on paper. Instead, we used the PiVizTool (see Section 3.3.2) to visualise the interactions that occur within the composite process. Four consecutive snapshots (out of a total of 15) from the resulting visualisation are shown in Figure 7.21. As previously explained, each edge in the PiVizTool visualisation represents a possible interaction between two agents. The grey edges represent interactions that are currently blocked whereas the black edges represent interactions that can occur immediately on the next execution step. Each snapshot is the result of applying one \( \pi \)-calculus reduction in the previous state.

For example, in snapshot 1 of Figure 7.21 the SelectLength process interacts with Cm2Inch by sending a length\_inch message, whereas in snapshot 2, Cm2Inch sends length\_inch to SelSki. Essentially, this corresponds to the conversion of the output of the SelectLength service from centimeters to inches via the Cm2Inch process. Also note the two options sso\_x and sso\_y for SelSki. These correspond to the two possible outcomes, i.e. the price in USD of a ski set or an exception. The possibility of choosing either of the two options allows us to simulate and investigate different scenarios.

The PiVizTool visualisation provides a low level simulation of the constructed composition. It can play an important role towards empirically verifying that our result satisfies the requested process. However, it also reveals some of the inner workings of the \( \pi \)-calculus, such as mechanics corresponding to input/output synchronisation and agent negotiation. These may be redundant and distracting for a user who is interested in process modelling and information flow and has no knowledge of the proofs-as-processes paradigm. Moreover in the PiVizTool simulation, due to the limitations described in Section 6.3, it is possible for a process to send its output before receiving all the inputs. A simulation that is better adapted to the context of processes and demonstrates the verified information flow and execution sequence can be easily obtained by the realisation of the \( \pi \)-calculus result as an executable module. We describe this procedure in the next chapter.

### 7.7 Conclusion and Future Work

The main aim of the work described in this chapter, is the development of a rigorous methodology for process composition. In the context of proofs-as-processes, our
7.7. Conclusion and Future Work

Figure 7.21: Consecutive snapshots of the Ski example π-calculus result taken from PiVizTool.
approach relies on the proof of the formal specification of the requested composite process using the specifications of the available component processes. This proof is facilitated by a number of derived inference rules and tactics that are tailored to the particular task of process composition.

In particular, we derived a number of inference rules that respect the polarization of the CLL connectives and the restrictions of CLL process specifications. Another group of inference rules, in combination with our developed automated tactics, facilitate the handling of buffered resources, which is a common theme in CLL proofs given the inherent accounting of resources.

Using these tools, we introduce the actions $\text{TENSOR}$, $\text{WITH}$, and $\text{JOIN}$, which correspond to parallel, conditional, and sequential composition respectively. These require a significant amount of complicated automation, especially in the case of $\text{JOIN}$, in order to ensure a sound, consistent, and intuitive result in any possible case.

Further investigation of a number of examples, including the reconstruction of Rao et al.’s ski purchasing example, reveals a number of interesting properties. For example, we touched upon compositionality, concurrent execution, and conditional execution. The concurrent nature of the constructed $\pi$-calculus terms allows for an efficient execution, where independent processes can be executed simultaneously. In contrast, conditional compositions allow the user to control the order of process execution in the workflow so that each process is only executed under the conditions where this is necessary.

Our intuitive, tailored process composition actions are complemented in a straightforward way by a diagrammatic user interface, thus making logic-based process composition available to non-experts in theorem proving or logic.

Apart from achieving our practical goals, we believe this work more generally demonstrates the challenges that need to be tackled in order to bridge the gap between the interesting and solid results of a theoretical concept, in our case the proofs-as-processes paradigm, and a pragmatic solution that can be used in real-world situations, in this case process composition. The development of the necessary rules, tools, and procedures requires knowledge and thorough understanding of the theory being used, the needs of the particular application, and, most importantly, how concepts, specifications, changes, inference steps, and proof strategies in the logic affect the structure, properties, and execution of the end result.
Based on this, we believe there is ample room for further improvement of the process composition tools described here. For example, various specialised cases of process compositions may lead to new types of composition steps. This, in turn, may lead to adjustments in the existing implementation, or the requirement for more useful inference rules and composition actions.

Another area of possible improvements, which we are already considering in ongoing side projects, is the diagrammatic interface. For example, finding better ways to clearly present buffered resources and conditional or ordering constraints between processes is a ongoing effort in the more general goal of providing an intuitive, user-friendly, mobile interface.

Our transition from theory to practice is completed in the next chapter, where we discuss how the $\pi$-calculus end result can be deployed as an executable software system.
Chapter 8

Automated Workflow Deployment

So far we have described mostly theoretical aspects of our approach to process composition, both from the logical and process algebraic points of view. Using logical specifications of processes, we construct formally verified workflows described using $\pi$-calculus terms, where component processes are treated as black boxes (see Section 6.3). Now we tackle the challenging task of converting these $\pi$-calculus terms into concrete executable code that aims to preserve the verified properties of the original workflow.

8.1 Overview

Drawing from all the information encoded in the process algebraic terms, our aim is to create a pragmatic, deployable, and scalable solution that can be used in a variety of real-world systems in such a way that the properties verified in the CLL proofs are directly applied.

We have chosen Scala (Odersky 2013) as the programming language for this task. It is a modern, hybrid object oriented and functional language, and as such, combines concepts from both these programming styles. As a pure object oriented language, every value in Scala is an object. Data and object types are described by classes or traits. A trait is a partially abstract class, i.e. a class where some members or methods may be abstractly specified and only concretely instantiated in classes or objects that extend the trait. This concept improves upon the notion of interfaces used in Java and, apart from allowing abstract specifications, it also facilitates multiple inheritance.
Moreover, Scala classes and traits form a hierarchy which begins from the universal superclass `Any` and ends in the bottom type (universal subclass) `Nothing`. Finally, as a functional language, Scala supports a number of elements from functional programming including type inference, currying, anonymous functions, pattern matching, and many others.

Overall, Scala offers some important benefits over other programming languages for this particular task:

1. It is a scalable language based on the Java Virtual Machine and, thus, with the potential to deploy solutions in multiple environments, from tablets and mobile phones to large computer networks.

2. Scala’s strong connections with Java allow the use of any Java class within Scala code. Thus, we can exploit the wide range of existing Java libraries in order to accommodate modern, widely used processes, connect to databases and other data sources, attach ontological classes to types (e.g. using the OWL API [Horridge and Bechhofer, 2011]), and, using the toolset provided by Java Enterprise Edition [Jendrock et al., 2010], construct workflows of web services and business processes.

3. Finally, Scala’s PILIB library [Cremet and Odersky, 2004] offers a direct translation of $\pi$-calculus terms to Scala code, which is convenient for our task at hand (see Section 8.2.2).

Since our efforts focus on automatically generating code from $\pi$-calculus compositions that have been constructed by the proofs-as-processes framework, the kind of $\pi$-calculus terms that our procedure needs to handle is limited to specific patterns, namely those involved in atomic and composite process specifications in CLL. Thus our task is slightly easier than having to translate any possible $\pi$-calculus term automatically into code.

We begin by describing the translation of core elements of CLL and the $\pi$-calculus as a preliminary step in Section 8.2. Next, in Section 8.3 we analyse the general architecture of the deployed components (including both atomic and composite processes). The details of our code generation procedure from proofs-as-processes based specifications are given in Section 8.4. Finally, we recap in Section 8.5 with a quick overview of the credit card example and the automatically generated Scala code for the `CreditCardPayment` workflow.
8.2 Preliminaries

In this section, we describe how some of the core elements of the CLL and π-calculus specifications used in our framework are translated into Scala code. Based on these fundamental translations, we can build the more complicated procedures in order to achieve automated workflow deployment as described in subsequent sections.

8.2.1 From CLL Types to Scala Types

In Section 6.2, we explained how CLL terms are used to describe the types of the inputs and outputs of a process based on the resource interpretation of CLL given in Section 4.2.

Each proposition in CLL is an abstract type specification of the corresponding resource. Since we can only express simple types in CLL, no further information about the type can be incorporated in its specification. As a result, the only natural translation of a CLL type in Scala is the introduction of an abstract type. We then rely on the user to determine the corresponding concrete Scala type instantiation for each CLL type. This instantiation may either be a primitive type or any complicated Scala class or trait or Java class or interface. This allows the flexibility of attaching complicated data structures, including relational database tables and ontological classes, to each CLL specified type.

For example, the CreditCardTransaction process (see Figure 6.1) has an input of type $PIN^\bot$. This is translated to an abstract type PIN in Scala. The user can then either implement this as a simple integer, or create a custom class that only allows 4-digit natural numbers. Similarly, the TRANSACTION type may be implemented as a complicated datatype that includes all the necessary information about the specific transaction taking place, or as a reference to a database entry with the relevant information.

Apart from CLL propositions, a translation of the CLL connectives is also necessary. More specifically we need to express parallel resources, i.e. translate the $\otimes$ and $\gamma$ operators and optional resources, i.e. the $\oplus$ and $\&$ operators. Parallel type $A \otimes B$ is translated to type Pair$[A,B]$ in Scala. Pairs in Scala allow the combination of two types A and B in one, which matches the behaviour of the binary operators $\otimes$ and
Exactly. Similarly, optional type \( A \oplus B \) is translated to Scala type \( \text{Either}[A,B] \). Instances of the \( \text{Either}[A,B] \) type can either be instances of the \( \text{Left}(x) \) object where \( x \) is of type \( A \) or of the \( \text{Right}(y) \) object where \( y \) is of type \( B \). This matches the exclusive disjunction behaviour of the binary operators \( \oplus \) and \( \& \) exactly.

For example, the CreditCardTransaction process has output type \( \text{PAYMENT} \oplus \text{EX\_BAD\_PIN} \) which is translated to the Scala type \( \text{Either}[\text{PAYMENT,EX\_BAD\_PIN}] \), where \( \text{PAYMENT} \) and \( \text{EX\_BAD\_PIN} \) are abstract types as described above. As a results, CreditCardTransaction can only provide an output \( \text{Left}(P) \) where \( P \) is of type \( \text{PAYMENT} \) or an output \( \text{Right}(E) \) where \( E \) is of type \( \text{EX\_BAD\_PIN} \). Once a resource of type \( \text{Either}[\text{PAYMENT,EX\_BAD\_PIN}] \) is received by some other process, simple pattern matching can be used to determine whether the output was of type \( \text{Left} \) or \( \text{Right} \) and therefore whether the payment was successful or a bad pin exception was thrown.

### 8.2.2 From \( \pi \)-calculus to Executable Code: The PiLIB Library

Even though the \( \pi \)-calculus provides abstract models of processes or agents, its semantics are well defined and conceptually well understood by the process calculi community. It provides a concurrent model of execution where independent agents without shared memory communicate with each other via message passing. It is also worth noting that the \( \pi \)-calculus names (i.e. messages and channels) are by nature untyped. Therefore, messages can contain any type of information and channels may be carrying any type of messages, including other channels.

In modern, concrete, distributed system implementations, message passing is more commonly used for the integration of loosely coupled components of large distributed systems, for example using Message Oriented Middleware (MOM) (Curry, 2005). In that case, messages usually follow a strict, predetermined protocol and their types are strictly determined by associated metadata. Since these systems do not have formal semantics, whether or not there is an association between them and the \( \pi \)-calculus has not been investigated yet.

In contrast, most modern, widely used, smaller scale, concurrent programming language libraries rely on shared memory management, often through semaphores or other types of locks, e.g. as described by (Wellings, 2004). There have been a few attempts to create concurrency libraries or languages based on a \( \pi \)-calculus model (Gi-
“PILib is a library written in Scala that implements the concurrency constructs of the \(\pi\)-calculus” (Cremet and Odersky, 2004). It exploits Scala features to mimic the \(\pi\)-calculus syntax and map its semantics to the underlying Java Virtual Machine. Originally PIlib was designed to become the main concurrency library for Scala. However, it was abandoned in favour of the actor-based library Akka (Typesafe Inc, 2013), although it is still available for use in the standard Scala distribution.

Table 8.1 summarizes the mapping of \(\pi\)-calculus primitives to PIlib syntax. We give a brief overview of each primitive in the following sections and refer the interested reader to the related paper (Cremet and Odersky, 2004) for a more detailed description.

<table>
<thead>
<tr>
<th>Description</th>
<th>(\pi)-calculus</th>
<th>PIlib</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel/name</td>
<td>(a)</td>
<td>(a : \text{Chan}{A})</td>
</tr>
<tr>
<td>Input</td>
<td>(a(x).P)</td>
<td>(a \ast { x \Rightarrow P })</td>
</tr>
<tr>
<td>Output</td>
<td>(\overline{a}(y).P)</td>
<td>(a(y) \ast P)</td>
</tr>
<tr>
<td>Parallel</td>
<td>(P \parallel Q)</td>
<td>spawn (&lt; P \mid Q &gt;)</td>
</tr>
<tr>
<td>Sum</td>
<td>(P + Q)</td>
<td>choice (P, Q)</td>
</tr>
</tbody>
</table>

Table 8.1: Mapping of \(\pi\)-calculus syntax to PIlib.

### 8.2.2.1 Channels/Names

As previously noted, \(\pi\)-calculus names are untyped. This allows for channels to carry other channels and messages to have arbitrary content. In contrast, Scala functions are statically and strongly typed. The gap between the two is bridged using polymorphism. The type of a channel \(a\) in PIlib is defined as \(\text{Chan}\{A\}\) where \(A\) is the type of message being carried by \(a\). Recursive typing allows the possibility for \(A\) to be an instance of type \(\text{Chan}\{B\}\), i.e. a channel carrying a message of type \(B\). In that case, \(a\) would have type \(\text{Chan}\{\text{Chan}\{B\}\}\), i.e. a channel carrying channels that carry messages of type \(B\).

Even though this allows the user to recreate any possible \(\pi\)-calculus term, it is still required that channel types are predetermined. This is particularly important in an automated code generation procedure.
8.2.2.2 Input

Input in PiLIB is defined as a channel function. Note that the "\(\ast\)" operator is used in PiLIB to represent the "\(\cdot\)" symbol of \(\pi\)-calculus. It essentially retrieves an input message \(x\) from channel \(a\) and then applies the anonymous function \(\{ x \mapsto P \}\) to it.

8.2.2.3 Output

Output in PiLIB is also defined as a channel function. To be more precise, the channel type \(\text{Chan}[A]\) inherits the function trait, i.e. a PiLIB channel is itself a function. The application \(a(y)\) of the channel \(a\) (as a function) on a message \(y\) accomplishes the output of that message. The "\(\ast\)" operator (implemented differently than the "\(\ast\)" operator of the input syntax) ensures \(P\) will be executed after the message is sent.

8.2.2.4 Parallel

The \texttt{spawn} object and its accompanying operators "\(<\", "\|\)”, and "\(>\)" in PiLIB is actually syntactic sugar for the \texttt{scala.concurrent.ops.spawn} function, which boils down to primitive JVM thread spawning. This ensures that parallel processes will be actually executed in parallel, to the extent that the standard JVM multithreading library can accomplish this.

8.2.2.5 Summation

A sum of \(\pi\)-calculus processes is implemented in PiLIB as a blocking (synchronized) list of its summands. A statically declared queue, which essentially acts as a standard message queue, holds such sums whose execution is pending. The \texttt{choice} function can be used to create a new sum \(s_1\). Then \(s_1\) is matched against the sums in the queue. If \(s_1\) is able to communicate with a sum \(s_2\) in the queue, then \(s_2\) is removed from the queue, awoken from its blocked state (or more precisely "notified"), and the communication between \(s_1\) and \(s_2\) is executed. Otherwise, \(s_1\) is added to the queue, blocking until a communication becomes available.

It is worth noting that all guarded processes in PiLIB must be introduced as sums of a single summand. This allows the \texttt{choice} function to add them to the message queue,
even though they only provide a single option for communication. This is valid in \( \pi \)-calculus since, based on the structural congruence rule \( P + 0 \equiv P \), any process \( P \) can be expressed as a sum.

It is also worth emphasizing that the mechanism to select the sum to be executed is deterministic; it selects the first sum in the queue that can communicate with \( s_1 \). This does not follow the \( \pi \)-calculus semantics where, if more than one communications are available, the order of execution is non-deterministic. For example, consider the following \( \pi \)-calculus term:

\[
\text{a}(x).P \parallel \text{a}(y).Q \parallel \text{a}(z).R
\]

In this term, either of \( \text{a}(x).P \) or \( \text{a}(y).Q \) can communicate with \( \text{a}(z).R \), and the choice between the two is arbitrary. However, the choice function will always use the first matching sum in the queue, i.e. in this case \( \text{a}(x).P \) will always be the one to interact with \( \text{a}(z).R \).

The choice function is also deterministically choosing which of the summands will be executed (assuming more than one summand can match) based on the order that they were given by the user. For example, consider the \( \pi \)-calculus term:

\[
(\text{a}(x).P + \text{a}(y).Q) \parallel \text{a}(z).R
\]

In this term, either of \( \text{a}(x).P \) or \( \text{a}(y).Q \) can communicate with \( \text{a}(z).R \), and the choice between the two is non-deterministic. However, the choice function will always use the first matching summand in the sum, i.e. in this case \( \text{a}(x).P \) will always be the one to interact with \( \text{a}(z).R \).

### 8.2.2.6 Restriction

In order to implement a restricted \( \pi \)-calculus channel in PILIB, variable scope in Scala is exploited. More specifically, a locally bound channel is introduced as a new variable of the appropriate \( \pi \)-calculus channel type of the form \( \text{Chan}[A] \) (see Section 8.2.2.1). The scope of the new variable is limited within the Scala function where it was introduced. Therefore, the corresponding \( \pi \)-calculus channel is only accessible to any \( \pi \)-calculus processes introduced within that limited scope and never outside that.
8.2.3 Extending the PiLib Library

In order to suit the particular needs of our workflow deployment library, we introduce extra features to the PiLib library that add functionalities such as logging, increased code readability through abbreviations, non-determinism, and synchronization. We discuss the implemented features in the next few sections.

8.2.3.1 Logging

A simple Java interface, namely PapPiLibLogger, has been implemented in order to support logging. PiLib allows the execution of arbitrary code when a channel is utilised (i.e. a message is transmitted through it) as well as at any point inside the body of a process. We exploit this functionality to produce log messages both at the channel (through the chanEcho function) and at the process level (through the procEcho function) at runtime. An implementation of PapPiLibLogger can control if, how, and where such messages are recorded.

8.2.3.2 Channel Types

In our implementation, a number of types and type abbreviations are introduced in order to model \( \pi \)-calculus channels and their functionality in the context of proofs-as-processes, as well as improve the readability of the produced code.

- The class \( \text{NChan}[A] \) is a type for a named \( \pi \)-calculus channel that carries a message of type \( A \) and extends PiLib’s channel type \( \text{Chan} \). Its \text{String} parameter provides a human readable name for the channel that identifies it. Channels of type \( \text{NChan}[A] \) log a generic message through \text{chanEcho} when in use (see Section 8.2.3.1).

- The type \( \text{PairChan}[A,B] \) is an abbreviation for \( \text{NChan}[\text{Pair}[A,B]] \), i.e. a channel that carries two messages, one of type \( A \) and one of type \( B \). It corresponds to a named channel in polyadic \( \pi \)-calculus that carries a message vector of size 2. These vectors appear often in proofs-as-processes \( \pi \)-calculus patterns.

- The type \( \text{OptChan}[A,B] \) corresponds to a named polyadic \( \pi \)-calculus channel that carries two channels of types \( \text{NChan}[A] \) and \( \text{NChan}[B] \) respectively, i.e. an abbreviation of type \( \text{PairChan}[\text{NChan}[A],\text{NChan}[B]] \). This type of channel
appears in the \(\pi\)-calculus translations of the \(\oplus\) and \& CLL rules (see Section 4.3). More specifically, it is the type of channel \(z\), whereas the types \(N\text{Chan}[A]\) and \(N\text{Chan}[B]\) correspond to the two channels \(u\) and \(v\) carried by \(z\). In less technical terms, \(z\) plays the role of the initial communication. It receives/sends the channels \(u\) and \(v\) that enable a conditional choice between the send/receive of two different messages. Channel \(u\) carries a message of type \(A\), whereas \(v\) carries one of type \(B\).

- Finally, the type \(N\text{Cont}[A]\) represents a continuation function for a named channel of type \(N\text{Chan}[A]\). Such functions usually correspond to \(\pi\)-calculus subterms that are parameterized by a channel. Their exact meaning is explained in Section 8.4.

The Scala code that introduces the above types is shown in Figure 8.1.

```scala
class NChan[A](name:String) extends Chan[A] {
  override def toString = name
  this.attach(s =>
    chanEcho("Transmitting message: "+ s + "] " +
    "through channel "+ this + "]")
}

type PairChan[A,B] = NChan[Pair[A,B]]
type OptChan[A,B] = PairChan[NChan[A],NChan[B]]

type NCont[A] = NChan[A] => Any

Figure 8.1: Type definitions in PAPPiLib.

8.2.3.3 Non-determinism

As mentioned in Section 8.2.2.5, PiLib makes deterministic decisions when selecting which sum in the queue to execute, and which of its summands will be executed. We replace the PiLib choice function by our own \texttt{rchoice} function, which randomly permutes the list of summands in every new sum, and then randomly inserts the sum in the queue. Sorting is based on Scala's \texttt{stableSort} algorithm and random choices
are made using Scala’s random integer generator. The corresponding Scala code is presented in Figure 8.2.

```scala
def ListShuffle[A:ClassManifest](list: List[A]): List[A] = {
  Sorting.stableSort(list,
    (_:A, _:A) => Random.nextInt(2) == 1).toList
}

def rchoice[A](s: GP[A]*): A = {
  // shuffle summands:
  val sum = Sum(ListShuffle(s.toList) map { _.untyped })
  // shuffle queue:
  synchronized { sums = ListShuffle(compare(sum, sums)) }
  (sum.continue).asInstanceOf[A]
}

Figure 8.2: Implementation of a non-deterministic choice function in PAPPIlib.

8.2.3.4 Syncronization

Generally, $\pi$-calculus processes interact and behave asynchronously. However, some level of syncronisation is required within a process itself. More specifically, in Section 6.3 we described how CLL specifications of processes assume the inputs and outputs of a process can happen in parallel. This is barely ever the case for any real process. Processes expect all of their inputs in order to be able to produce the corresponding output. Therefore, viewed concretely, a syncronisation of inputs is required before a process is able to provide an output.

When mapping CLL process specifications to normal behaving $\pi$-calculus terms, seqments of a CLL judgement, corresponding to individual inputs or outputs, are composed in parallel using the $\pi$-calculus $\parallel$ operator (see Section 6.3). In PPIlib this would translate into a call to the spawn object, which would execute all the subterms (i.e. all the inputs and outputs) in parallel. In our case, we require all of the inputs to be executed first (in parallel) before we execute the output.

For this purpose, we make use of Scala’s scala.concurrent.ops.future (or simply future) function. In contrast to spawn, future provides the means to recover the
result from running the involved thread. More specifically, it provides a function of type \( () \Rightarrow A \) that, when applied to Unit, blocks until the running thread terminates and then returns the result of type A from the thread execution.

In our context, every piece of input code (corresponding to a CLL sequent) is executed within a thread invoked by \texttt{future}. We then use the returning functions from all such input threads to block until all the inputs have been received, before moving on to the execution of the output. In addition, we are able to retrieve the actual message that was given as input and allow the implemented process to use it in order to produce the corresponding output.

Note that thread spawning using \texttt{future} allows all the input threads to execute in parallel. This means the process can still receive all the inputs in parallel, therefore does not impose any order on them (as dictated by the CLL specification), and thus no deadlocks are created. Synchronization occurs at the point of recovery of the results from each input thread. In short, all inputs are received in parallel, but the process blocks until all inputs have been received.

### 8.2.4 Type Casting

Despite our efforts to exploit Scala’s type inference as much as possible, it is not entirely possible to infer all the types of all the channels of a \( \pi \)-calculus term. There are two main reasons for this:

1. Since \( \pi \)-calculus terms are untyped, they do not contain the type information of the involved resources. For example, in the \( \pi \)-calculus expression of the axiom \texttt{buffer \( \vdash \langle x \langle a \rangle, 0 \rangle :: \langle y \langle a \rangle \rangle, x \colon A, y \colon A^\perp \)}, the information that message \( a \) is of type A is lost. Creating a framework that maintains this type information while proofs are being performed and then exploiting it during code generation is non-trivial.

2. In some cases, the Scala type system does not allow us to fully determine the type of a channel in advance. For example, in Figure 8.1 we showed that a continuation function is a channel function of type \texttt{NChan[A] => Any}. Scala’s type system does not allow a function of type \texttt{NChan[NChan[B]] => Any} to work as a continuation function, even though the corresponding \( \pi \)-calculus term may be valid. Another case of type restrictions imposed by Scala where we are forced to use the root type \texttt{Any} is discussed in Section 8.4.1.6.
Given these restrictions and the top-down recursive nature of our code generation procedures, we chose to use dynamic type casting (i.e. the `asInstanceOf[]` method in Scala) in order to cast the types of all channels to the appropriate ones so that our code compiles properly. The appropriate types are determined by a combination of exploiting the CLL type specification and our custom built type inference during code generation. Since \( \pi \)-calculus terms are extracted from CLL proofs, they are already type checked by the proof and therefore there is no risk of getting a `ClassCastException` for an invalid type cast. On the downside, the code generated often includes multiple type casts of the same variable in the same code block, but this is the only way to ensure compatibility and compositionality of the generated code templates discussed in Section 8.4.

### 8.3 Main Component Architecture

In this section, we describe the system architecture for the deployment of the main components of our constructed workflows, i.e. the atomic and composite processes. More specifically, in Section 8.3.1 we describe the deployment of atomic processes as Scala traits, and in Section 8.3.2 we describe the deployment of composite processes as concrete Scala classes. We provide UML diagrams (Object Management Group, 2010) of the resulting architecture, a description of the involved methods, and a template of the generated code in each case.

A general roadmap of our deployment procedure is shown in Figure 8.3. The \( \pi \)-calculus models of the atomic processes are composed to form \( \pi \)-calculus workflows (note that composite processes may also be used as components). Our automated deployment algorithms convert atomic process models to abstract Scala traits, whereas composite processes are translated to Scala classes with the instances of the traits of their components as members. The user is responsible to provide concrete implementations for each atomic process since our framework otherwise considers them as black-boxes.

It is important to emphasize that the code for the traits and classes described in the following sections is generated fully automatically. The user is only required to implement the concrete bodies of the abstractly specified component processes.
8.3.1 Atomic Processes as Traits

Our constructed process compositions mainly involve the information flow between the component processes. As previously described in Section 6.3, the compositions treat component processes as black boxes and take into consideration only their free input and output $\pi$-calculus channels, and the type of information expected to be carried by those channels, which is described via the corresponding CLL term. The actual functionality and implementation of the components remains abstract. Moreover, the CLL propositions offer a description for simple, abstract types which may also have an unknown concrete instantiation as explained in Section 8.2.1.

As an example, let us consider $\text{ExampleProcess}$ as a process with two inputs of types $A$ an $B$ through channels $x$ and $y$ respectively, and an output $C$ through channel $z$. Its annotated CLL definition is the following:

$$\vdash \text{ExampleProcess:: } x:A^\perp, \ y:B^\perp, \ z:C \quad (8.1)$$

Since this specification is the only concrete description of the process, and its actual
functionality (i.e. how the outputs are generated by the inputs) is abstract, we implement atomic processes as Scala traits. Scala traits can be viewed as partially abstract classes. The UML diagram of the generated trait for ExampleProcess and its dependencies is shown in Figure 8.4.

![UML diagram of the generated trait for ExampleProcess and its dependencies.](image)

Figure 8.4: UML diagram of the generated trait for the atomic process ExampleProcess specified by (8.1) and its dependencies.

When implementing an atomic process, its corresponding trait includes the human-readable name of the process (as a Scala String) and the communication code that corresponds to the $\pi$-calculus specification, but it abstracts from the actual functionality of the process.

More specifically, an atomic process trait inherits the class of functions that have the CLL specified inputs and output. Since our CLL specifications are restricted to a single (composite) output, implementing processes as Scala functions is straightforward. The functionality of the process can be described by implementing the inherited abstract function “apply”. Therefore, mapping an atomic process defined in our system is as straightforward as implementing a Scala class that inherits the generated trait and implements (overrides) the apply function. Note that, as described in Section 8.2.1, concrete instantiations of the involved CLL types must also be provided.

For example, the trait of ExampleProcess will inherit the function class $((A, B) \Rightarrow C)$. A concrete class ExampleProcessInstance that extends this trait must override an apply function of this type, i.e. implement a function that has two arguments of types $A$ and $B$ and a return type $C$. Naturally, for this to be accomplished, types $A$, $B$, and $C$ must have concrete instantiations as well. The UML diagram of all these instantiations is shown in Figure 8.5.
Figure 8.5: UML diagram of the user-provided `ExampleProcess.getInstance` class which extends the abstract `ExampleProcess` trait.

The automatically generated trait also includes the means for the process to communicate. This is dictated by the \( \pi \)-calculus specification, and is implemented as the `run` function. Its arguments are the free typed (since this is now Scala code) \( \pi \)-calculus channels in the specification. Moreover, the \( \pi \)-calculus specification is split into two parts based on the corresponding CLL terms: the `input`, which describes the means for the process to receive the appropriate inputs, and the `output`, which describes the means of sending the resulting output. The input and output \( \pi \)-calculus terms are translated into Scala functions as described in Section 8.4.1. These functions are applied on their respective channels to perform the communication.

For the `ExampleProcess`, the input part contains two functions that receive information of type \( A \) and \( B \) respectively, and the output part contains a function that sends information of type \( C \).

The information received as input is synchronised using future thread spawning (see Section 8.2.3.4 for more details) before calling the `apply` function to obtain the output of the process and then send it. A template of the trait for the `ExampleProcess` is shown in Figure 8.6. Notice the application of `this` which corresponds to a call to the
apply function (since the trait inherits a function class).

```scala
trait ExampleProcess extends ((A, B) => C) {
  var name = "ExampleProcess"
  def run(x :NChan[Any], y :NChan[Any], z :NChan[Any]) = {
    // Pi-calculus input part
    val inputf0 = //... Communication code for type A ...
    val input0 = future(inputf0 (x))

    val inputf1 = //... Communication code for type B ...
    val input1 = future(inputf1 (y))

    val outputf = {
      // Process call - apply
      val out = (this ((input0 ()).asInstanceOf[A])
        ((input1 ()).asInstanceOf[B])
        ).asInstanceOf[C]

      // Pi-calculus output part
      // ...
      // Communication code for 'out' of type C
      // ...
    }
    outputf (z)
  }
}
```

Figure 8.6: Template of generated Scala trait for a atomic process ExampleProcess specified by (8.1).

It is important to note that the code for the trait is generated completely automatically from the CLL specification and the corresponding π-calculus mapping as explained in Section 8.4.1. The only part left to the user is the instantiation of this trait into a class that implements the apply function. Our system also generates a code template to further facilitate the implementation of these classes. For example, the code template generated for ExampleProcessInstance is shown in Figure 8.7.
8.3. Main Component Architecture

class ExampleProcessInstance extends ExampleProcess {
  override def apply( arg0 :A, arg1 :B ) :C = {
    // TODO: Instantiate this method.
  }
}

Figure 8.7: Generated code template for ExampleProcessInstance which inherits the ExampleProcess template and determines the functionality of the process. The user is merely required to fill in the code for the apply function.

8.3.2 Composite Processes

A composite process functions as the workflow of communication between its components. As a result, composite processes are implemented as classes, whose constructors include objects that correspond to each of their components. Moreover, composite processes have CLL specifications that adhere to the same rules as in the case of atomic processes. Consequently, they can also inherit a function type that represents their type specification.

For example, assume a composite process CompositeExample that has three components ExampleProcess1, ExampleProcess2, and ExampleProcess3. Also assume CompositeExample has the following specification:

\[ \vdash \text{CompositeExample} :: a : X \perp, \ b : Y \perp, \ c : Z \]  \hspace{1cm} (8.2)

The corresponding Scala class inherits the function type \((X, Y) \to Z\) and includes a run function that corresponds to the \(\pi\)-calculus communication of the process. Its setup is therefore very similar to the generated traits for atomic processes. There are however two main differences:

1. First, the run function corresponds to the execution of the \(\pi\)-calculus workflow that was constructed via proof, as opposed to the translation of the CLL specification into a process with the associated input syncronisation. The translation of the \(\pi\)-calculus terms extracted from the CLL proofs into Scala code for this function is described in Section 8.4.2.

2. Second, the functionality of the composite process is known by the system, i.e.
the apply function is not abstract. In fact, it consists of the execution of the workflow (via the run function) with a provided set of inputs, and the extraction of the result of this workflow execution.

The UML diagram of the architecture of the CompositeExample process is shown in Figure 8.8. Moreover, a template of the generated Scala class for CompositeExample is shown in Figure 8.9.

Figure 8.8: UML diagram of the generated Scala class for the composite process CompositeExample with three components ExampleProcess1, ExampleProcess2, and ExampleProcess3 and specified by (8.2), and its dependencies. User-defined instances for all abstract types and each of the three components must be provided.

In order to make the composite process usable as the apply function in Scala, we need to implement the means to provide the inputs of the composite process as arguments to apply, as well as the means to retrieve the output of the composite process as the returned value of the same function. Since the composite process communicates its inputs and outputs through π-calculus channels, the functionality of the apply
class CompositeExample {
    process1 :ExampleProcess1,
    process2 :ExampleProcess2,
    process3 :ExampleProcess3
} extends ((X, Y) => Z) {

    var name = "CompositeExample"
    def run(a :NChan[Array], b :NChan[Array], c :NChan[Array]) = {
        //... Translation of pi-calculus composition to code ...
    }

    override def apply ( arg0 :X, arg1 :Y ) :Z = {
        def request( a : NChan[Any], b : NChan[Any] ) = {
            //... Communication code for inputs arg0 and arg1 ...
        }

        def response( c : NChan[Any] ) = {
            //... Communication code for output of type Z ...
        }

        // Construct local communication channels
        val a = new NChan[Array]("a")
        val b = new NChan[Array]("b")
        val c = new NChan[Array]("c")

        scala.concurrent.ops.spawn(request(a , b)) // send input
        scala.concurrent.ops.spawn(this.run(a, b, c)) // execute
        (future (response(c))) () // block to receive output
    }
}

Figure 8.9: Template of generated class for composite process CompositeExample with 3 components: ExampleProcess1, ExampleProcess2, and ExampleProcess3, and specified by (8.2).
function must involve a set of $\pi$-calculus *pseudo*-processes that communicate with the composite process in the normal proofs-as-processes style. We refer to these as *pseudo*-processes because they uniquely violate the standards of $\pi$-calculus process specifications that we have set so far.

In particular, we require pseudo-processes that have no inputs and each provide one output for a matching input in the CLL specification of the composite process, and another pseudo-process that must be able to receive the composite service’s output as its input without having any output of its own. The example of CompositeExample connected to its pseudo-processes is shown in Figure 8.10.

![Figure 8.10: Diagram of the CompositeExample process connected to the necessary pseudo-processes that allow it to be treated as a Scala function.](image)

To achieve this, we negate each of the terms in the CLL specification of the composite process. For each of these negated terms we generate the corresponding $\pi$-calculus term using the mapping described in Section 6.3. Each $\pi$-calculus term can, in turn, be translated to Scala code by using the same algorithm that generates code for atomic processes as described in Section 8.4.1. The translation of the inputs of the composite process form the request function in Scala, whereas the translation of the output forms the Scala function response.

In the example of CompositeExample, we negate the terms $a:X^\perp$, $b:Y^\perp$, and $c:Z^\perp$ from the CLL specification (8.2) to obtain the new sequents $a:X$, $b:Y$, and $c:Z^\perp$. We then introduce the pseudo-processes specified by $\vdash \text{in0} :: a:X$ and $\vdash \text{in1} :: b:Y$, which correspond to the communication of the input parameters to the composite process, and the pseudo-process specified by $\vdash \text{out} :: c:Z^\perp$, which corresponds to the retrieval of the output from the composite process. Based on these, the code for the apply function of CompositeExample consists of (a slightly optimised version of) the process corresponding to the following proof:
This proof enables the interaction of the CompositeExample process with the introduced pseudo-processes through the \textit{Cut} rule as shown in Figure 8.10. This is a valid CLL proof that is easily translated to the \(\pi\)-calculus and then, based on the procedure described in Section 8.4.2, to Scala code. The request function will incorporate the generated code for pseudo-processes \textit{in0} and \textit{in1}, whereas the response function includes the code generated for the \textit{out} pseudo-process.

Based on the presented architecture for a composite process, no further implementation is required by the user. The composition code is essentially extracted from the CLL specification and proof and generated completely automatically. The user only needs to construct an object of the generated class by providing concrete instances for each component. The object can then be used as a function over any set of arguments in order to execute the workflow and obtain the expected result. Additionally, the \textit{run} function allows any composite process to be used as a \(\pi\)-calculus component within other compositions in the same fashion as atomic processes do.

### 8.4 From Proofs-as-processes to Functions

In this section, we introduce the Proofs-as-Processes PI\text{LIB} (PAP\text{PI\text{LIB}}) library, which includes PI\text{LIB}-based functions that correspond to \(\pi\)-calculus patterns extracted from each CLL inference rule. This library allows us to directly implement proofs-as-processes patterns in Scala in a reusable, extensible, and composeable way. In particular, we make a distinction between \(\pi\)-calculus patterns appearing in the specifications of atomic processes and \(\pi\)-calculus patterns that appear in the extracted specification of a constructed workflow. We describe both separately in detail, the former in Section 8.4.1 and the latter in Section 8.4.2.
Table 8.2: Mappings of \( \pi \)-calculus terms for atomic processes to corresponding Scala functions.

### 8.4.1 Atomic Processes

In order to be able to execute a composite process constructed by our proofs-as-processes framework, the individual components must have a concrete instantiation that corresponds to their logical specification. We introduced \( \pi \)-calculus instantiations for atomic processes with respect to the expected behaviour within the proofs-as-processes paradigm in Section 6.3. In this section, we present the translation of these generated \( \pi \)-calculus terms into Scala code. As a follow up from Table 6.1, we present the mappings from \( \pi \)-calculus terms to the corresponding Scala function that implements these terms in code in Table 8.2.

Each of the patterns in Table 8.2 receives or sends information through the free channel \( z \). Therefore, our aim when translating \( \pi \)-calculus terms for input and output into Scala is to create a function that has a \( \pi \)-calculus channel (corresponding to \( z \)) as a single parameter. We call these functions \textit{continuation functions} and their Scala type is \( \text{NCont}[A] \) as described in Section 8.2.3.2. In this way, when a continuation function is applied to a concrete channel, the corresponding input/output operation is performed over that particular channel.

The parallel and optional cases are parameterised by the code for their individual components, as discussed in Section 6.3. For example, the code to send a parallel output of type \( U \otimes V \) over channel \( z \) will be built on top of the code for an output of (arbitrarily complex) type \( U \) and the code for an output of type \( V \). For this reason, the corresponding \( \pi \)-calculus term \( (v \ a \ b) (\langle a, b \rangle \cdot (U_\pi(a) || V_\pi(b))) \) contains placeholders \( U_\pi(a) \) and \( V_\pi(b) \) corresponding to the two component outputs \( U \) and \( V \). The code for the two outputs \( U \) and \( V \) will also consist of one continuation function for each output.
These functions will be applied to channels that introduced internally in the pattern (in this example channels \(a\) and \(b\)). Therefore, each placeholder in a parameterised \(\pi\)-calculus pattern is translated as a continuation function argument of type \(NCont[A]\) in the function generated for the whole pattern. Our terminology *continuation functions* can now be justified in the sense that they *continue* the work started by the original pattern. In terms of our example, this means “send an output through \(z\) then continue by sending an output through \(a\) and another through \(b\)”. Since every \(\pi\)-calculus sub-term is translated to a continuation function (i.e. all generated functions have the same Scala type), we can compose continuation functions together and thus easily construct code that corresponds to more complicated \(\pi\)-calculus terms (see Section 8.4.1.3 for an example).

We should also note the existence of a `process:String` parameter in all of the implemented functions (e.g. see Figure 8.11). This is used to provide a human readable name that identifies the process which calls the particular function. It allows us to associate input and output actions to the corresponding process when recording log messages through the `procEcho` function (see Section 8.2.3.1).

In the next sections, we describe each of the cases presented in Table 8.2 in more detail and provide the generated code template for each one.

### 8.4.1.1 Atomic Output

An atomic output is implemented as the `PiOut` function shown in Figure 8.11 using the standard `*` operator from PPLIB (see Section 8.2.2.3). The `rchoice` function is used to introduce non-determinism (see Section 8.2.3.3). The `msg` parameter is the actual message that will be sent. The type parameter \(A\) allows the message to be of any type, including another arbitrarily typed channel. The `chan` parameter is the channel through which the message will be transmitted, and is naturally of type \(NChan[A]\) since the message has type \(A\).

We exploit Scala’s currying features so that we can use `PiOut` as a partial function, in the case where the channel is not determined by the specification (i.e. it is not a free name). This allows us to compose `PiOut` with other PAPPPLIB functions and take advantage of Scala’s type inference so that we do not have to explicitly provide the type of `chan` in these cases (see Section 8.4.1.3 for an example).
def PiOut[A](process:String, msg: A) (chan: NChan[A]) = {
  rchoice ( chan(msg) * {
    procEcho("Process ["+process"] sent: ["+msg"] " +
      "through channel ["+chan"]")
    Unit
  })
}

Figure 8.11: Implementation of the PiOut function for atomic output in PAPPI LIB.

The output function that corresponds to an atomic literal of the form \( z:A \), i.e. a single atomic output (e.g. a message \( x \) of type \( A \)) through channel \( z \) is shown in Figure 8.12.

val outputf = PiOut(name,x)_.
outputf (z)

Figure 8.12: Generated code for atomic output \( x \) of type \( A \). This corresponds to the CLL term \( z:A \) and to the \( \pi \)-calculus term \( (\nu x)(z(x).0) \).

8.4.1.2 Atomic Input

The PiIn function, as shown in Figure 8.13 implements an atomic input for a process. It is defined similarly to PiOut (see Section 8.4.1.1), only instead of having a msg parameter, the message (of type \( A \)) is the one received by the channel, and is returned by PiIn. Currying is also used in PiIn to achieve compositionality and automatic type inference.

def PiIn[A](process:String) (chan: NChan[A]) = {
  rchoice ( chan * { msg =>
    procEcho("Process ["+process"] received: ["+msg"] " +
      "through channel ["+chan"]")
    msg
  })
}

Figure 8.13: Implementation of the PiIn function for atomic input in PAPPI LIB.

The input function that corresponds to an atomic literal of the form \( z:A \), i.e. a single
8.4. From Proofs-as-processes to Functions

8.4.1 Parallel Output

Parallel output is implemented in PAPPILIB as the PiTimes function shown in Figure 8.15. This corresponds to a CLL term of the form $A \otimes B$ (see Table 6.1). It requires two channel names `name_a` and `name_b`. These correspond to the names of the two locally bound channels $a$ and $b$ as seen in the $\pi$-calculus term that corresponds to $A \otimes B$ (see Table 8.2), i.e. the two channels that will carry the two parallel outputs of type $A$ and $B$ respectively. Note that these names do not affect the functionality of the process, but are only used for logging purposes (see Section 8.2.3.2).

```
def PiTimes[A,B](process: String, name_a: String, name_b: String) (cont_a: NCont[A]) (cont_b: NCont[B]) (out: PairChan[NChan[A],NChan[B]]) = {
  val nchan_a = new NChan[A](name_a)
  val nchan_b = new NChan[B](name_b)
  rchoice ( out(Pair(nchan_a,nchan_b)) * {
    spawn < cont_a (nchan_a) | cont_b (nchan_b) >
  })
}
```

Figure 8.15: Implementation of the PiTimes function for parallel output in PAPPILIB.

The `out` parameter of PiTimes corresponds to the output channel $z$ which is a free name in the $\pi$-calculus pattern. Its type must be PairChan[NChan[A],NChan[B]]
Chapter 8. Automated Workflow Deployment

because it carries the two channels \( a \) and \( b \) that have types \( \text{NChan}[A] \) and \( \text{NChan}[B] \) respectively.

The means of delivering component outputs \( A \) and \( B \) must be provided to \( \text{PiTimes} \) in the form of the continuation functions \( \text{cont}_a \) and \( \text{cont}_b \). In the case where \( A \) and \( B \) are propositions, i.e. atomic outputs, \( \text{cont}_a \) and \( \text{cont}_b \) will be instantiated using \( \text{PiOut} \). The resulting code is shown in Figure \( 8.16 \).

The currying feature in Scala, allows us to compose \( \text{PiOut} \) with \( \text{PiTimes} \) as the code is generated. Essentially, we are constructing continuation functions that, if applied on an appropriately typed channel, can accomplish the desired output. Continuation \( \text{cont}_a \) outputs \( x \), whereas \( \text{cont}_b \) outputs \( y \). The constructed \( \text{outputf} \) function uses \( \text{PiTimes} \) to output \( \text{out} = (x,y) \), once it is applied to channel \( z \).

```scala
val outputf = out match {
  case (x,y) =>
    val cont_a = PiOut(name,x)_
    val cont_b = PiOut(name,y)_
    PiTimes(name, "a", "b") (cont_a) (cont_b) _
}
outputf (z)
```

Figure 8.16: Generated code for parallel output \( \text{out} = (x,y) \) where \( x \) and \( y \) are atomic outputs of type \( A \) and \( B \) respectively. This corresponds to the CLL term \( z : A \otimes B \) and to the \( \pi \)-calculus term \((\nu a,b)(\langle a,b \rangle.(\langle x \rangle (\overline{a}(x).0) || (\nu y)(\overline{b}(y).0))))\).

As an example to demonstrate the compositionality of the constructed continuation functions, we consider the case where \( A \) or \( B \) are not atomic. For example, consider a triple parallel output \( \text{out} = (x,(y,w)) \) which corresponds to the type \( X \otimes (Y \otimes W) \) (note the use of small letters of the channels and capital letters for the corresponding types). The continuation function for \( X \) is as a simple application of the \( \text{PiOut} \) function as before. The continuation function for \( Y \otimes W \) is a combination of \( \text{PiTimes} \) with two \( \text{PiOut} \) functions for channels \( y \) and \( w \) respectively. The generated code is shown in Figure \( 8.17 \). Notice how the code of \( \text{cont}_b \) is almost identical (apart from some of the parameters) to the code for \( A \otimes B \) in Figure \( 8.16 \). Also note the use of brackets so that the scope of \( \text{cont}_a \) and \( \text{cont}_b \) is local and does not clash with previous definitions.
val outputf = out match {
  case (x, out') =>
    val cont_a = PiOut(name, x)_
    val cont_b = out' match {
      case (x, y) =>
        val cont_a = PiOut(name, y)_
        val cont_b = PiOut(name, w)_
        PiTimes(name, "b", "c") (cont_a) (cont_b) _
      }
    PiTimes(name, "a", "z'") (cont_a) (cont_b) _
  }
  outputf {z}

Figure 8.17: Generated code for parallel output $out = (x, (y, w))$ where $x$, $y$, $w$ are atomic outputs of type $X$, $Y$, and $W$ respectively.

This corresponds to the CLL term $z : X \otimes (Y \otimes W)$ which translates to the $\pi$-calculus term $(\nu a, z') (z (a, z')).((v x) (\pi (x).0) || (v b, c) (z (b, c)).((v y) (b (y).0) || (v w) (c (w).0))))$.

8.4.1.4 Parallel Input

The $\PiPar$ function shown in Figure 8.18 implements parallel input which corresponds to a CLL term of the form $A \perp \otimes B \perp$ (see Table 6.1). It is implemented in a way similar to its counterpart $\PiTimes$. One of the main differences is that in $\PiPar$ the two input channels $a$ and $b$ are received through $z$ (see the corresponding $\pi$-calculus term in Table 8.2), and therefore do not require explicit names to be provided. The $in$ parameter corresponds to channel $z$ in the $\pi$-calculus term (similarly to how the $out$ parameter works for $\PiTimes$).

The means of receiving the two individual inputs of type $A \perp$ and $B \perp$ (which may not be atomic) are provided by the continuation functions $\text{cont}_a$ and $\text{cont}_b$. The code generated for the simple case of two parallel atomic inputs is shown in Figure 8.19. In more complicated cases where $A$ or $B$ are not atomic, the definitions of continuation functions $\text{cont}_a$ and $\text{cont}_b$ will be replaced by more complicated input functions in a way similar to that of $\PiTimes$ described in the previous section.

A notable feature of the implementation of $\PiPar$ is the use of $\text{future}$. It is used to accomplish input synchronisation as explained in Section 8.2.3.4. In short, the two
def PiPar[A,B](process: String)
    (cont_a: NCont[A]) (cont_b: NCont[B])
    (in: PairChan[NChan[A],NChan[B]]) = {
        rchoice ( in * { case Pair(chan_a,chan_b) =>
            val fa = future (cont_a (chan_a))
            val fb = future (cont_b (chan_b))
            (fa (), fb ())
        })
    }

Figure 8.18: Implementation of the PiPar function for parallel input in PAPPILib.

val inputf = {
    val cont_a = PiIn[Any](name)_.
    val cont_b = PiIn[Any](name)_.
    PiPar(name) (cont_a) (cont_b)
}

val input = future(inputf (z))

Figure 8.19: Generated code for parallel input input = (x,y) where x and y are atomic inputs of type A and B respectively. This corresponds to the CLL term z: A⊥ ⇆ B⊥ and to the π-calculus term z(a,b).((a(x).0 || b(y)).0).
inputs are received in parallel through two different threads, but PiPar blocks until both inputs have been received.

### 8.4.1.5 Optional Output

We use the PiPlus function, as shown in Figure 8.20, to implement optional output in PAPPILIB. This corresponds to a CLL term of the form \( A \oplus B \) (see Table 6.1). Similarly to PiTimes, parameters \( name_x \) and \( name_y \) correspond to the names of the bound \( \pi \)-calculus channels \( x \) and \( y \), and the \( out \) parameter corresponds to the main channel \( z \).

```scala
def PiPlus[A,B](process: String,
                 name_x: String, name_y: String)
           (cont: Either[NCont[A],NCont[B]])
           (out: OptChan[NChan[A],NChan[B]])=

   cont match {
   case Left(cont_x) => {
      val chan_x = new NChan[A](name_x)
      rchoice ( out * { case Pair(u,v) =>
                        rchoice(
                          (u(chan_x) * { cont_x (chan_x) })
                        )
                      })
   }
   case Right(cont_y) => {
      val chan_y = new NChan[B](name_y)
      rchoice ( out * { case Pair(u,v) =>
                        rchoice(
                          (v(chan_y) * { cont_y (chan_y) })
                        )
                      })
   }
   }

Figure 8.20: Implementation of the PiPlus function for optional output in PAPPILIB.

The continuation function controlling the output of component \( A \) or \( B \) is slightly more complicated than for parallel inputs/outputs. Essentially, the choice between an output
of type \(A\) or an output of type \(B\) is external to the \(\pi\)-calculus. Since we are modelling actual processes, the choice is, in fact, made by the process itself. Therefore, based on the type of the output of the process, \(\text{PiPlus}\) will either be provided with a continuation of type \(\text{NCont}[A]\) or a continuation of type \(\text{NCont}[B]\). Using Scala’s Either type, the output message is expected to be of type \(\text{Either}[A,B]\) (see Section \[8.2.1\]) and the type of the continuation parameter \(\text{cont}\) is \(\text{Either}[\text{NCont}[A],\text{NCont}[B]]\).

This can be made more clear through the simple example of two optional atomic outputs \(a\) and \(b\) of type \(A\) and \(B\) respectively, i.e. the CLL sequent \(z: A \oplus B\). The generated output function for this example is shown in Figure \[8.21\]. The \(\text{cont}_x\) function corresponds to the continuation that outputs \(a\), whereas \(\text{cont}_y\) is the continuation that outputs \(b\). Once again, the use of continuation functions allows for more complicated cases where \(A\) or \(B\) are not atomic. Pattern matching on the output message \(\text{out}\) determines whether \(\text{PiPlus}\) will be provided \(\text{Left}(\text{cont}_x)\) or \(\text{Right}(\text{cont}_y)\), and thus utilise the appropriate output function.

```scala
val outputf = {

  val cont = (out) match {
    case Left(a) =>
      val cont_x = PiOut(name,a)_
      Left(cont_x)
    case Right(x) =>
      val cont_y = PiOut(name,b)_
      Right(cont_y)
  }

  PiPlus(name, "x", "y") (cont)_
}

outputf (z)
```

Figure 8.21: Generated code for optional output \(\text{out}\) of either \(\text{Left}(a)\) or \(\text{Right}(b)\) where \(a\) and \(b\) are atomic outputs of type \(A\) and \(B\) respectively.

This corresponds to the CLL term \(z: A \oplus B\) which translates to the \(\pi\)-calculus term \((\nu x,y)(z(u,v).((\nu x).((\nu a) (\nu (a).0) + \nu (y).((\nu b) (\nu (b).0))))).\)
8.4.1.6 Optional Input

In the spirit of the functions described so far, \texttt{PiWith} implements optional input in \textsc{PAPPILib}. This corresponds to a CLL term of the form $A^\perp \& B^\perp$ (see Table 6.1). The implementation is shown in Figure 8.22. Parameters \texttt{name\_u} and \texttt{name\_v} correspond to the names of the bound \(\pi\)-calculus channels \(u\) and \(v\), and the \texttt{in} parameter corresponds to the main channel \(z\). Two continuation functions \texttt{cont\_u} and \texttt{cont\_v} for each of the two optional outputs must also be provided.

```scala
def PiWith[A,B](process: String, name_u: String, name_v: String) (cont_u: NCont[A]) (cont_v: NCont[B]) (in: OptChan[NChan[A],NChan[B]]) = {
  val nchan_u = new NChan[NChan[A]](name_u)
  val nchan_v = new NChan[NChan[B]](name_v)
  rchoice ( in(Pair(nchan_u,nchan_v)) * {
    rchoice(
      (nchan_u * { u => Left(cont_u (u)):Either[Any,Any] } ),
      (nchan_v * { v => Right(cont_v (v)):Either[Any,Any] } )
    )
  })
}
```

Figure 8.22: Implementation of the \texttt{PiWith} function for optional input in \textsc{PAPPILib}.

The implementation of the \texttt{PiWith} requires a peculiarity compared to the other functions, that of type casting the result of the application of the continuation functions to \texttt{Either[Any,Any]}. Since \texttt{cont\_u} and \texttt{cont\_v} both have return type \texttt{Any}, the type of value \texttt{Left(cont\_u (u))} is determined by Scala to be \texttt{Either[Any,Nothing]} whereas the type of value \texttt{Right(cont\_v (v))} is \texttt{Either[Nothing,Any]}. However, since either of the two values can become the final return value of \texttt{PiWith}, the two types must match, and are therefore type cast to \texttt{Either[Any,Any]}, i.e. the only common supertype of both \texttt{Either[Any,Nothing]} and \texttt{Either[Nothing,Any]}.

The generate code for the simple case of two atomic optional inputs is shown in Figure 8.23.
val inputf = {
  val cont_x = PiIn[Any](name)_
  val cont_y = PiIn[Any](name)_
  PiWith(name, "u", "v") (cont_x) (cont_y) _
}
val input = future(inputf (z))

Figure 8.23: Generated code for optional input of either type A or B.
This corresponds to the CLL term \( z: A^\perp \& B^\perp \) which translates to the \( \pi \)-calculus term
\( (v\ u\ v)\(\pi(u\ v).\(u\(x\(a\)\).0+v\(y\(b\)\).0)) \), where \( a \) and \( b \) are anonymous.

8.4.2 Composite Processes

Composite process generally consist of large \( \pi \)-calculus terms that are extracted from CLL proofs. Generating the corresponding Scala code can be accomplished by a translation of the \( \pi \)-calculus patterns of each CLL inference rule into code. For this, we follow the same approach and concepts as described in Section 8.4.1 for atomic processes, in our effort to achieve compositionality of the generated code and exploit Scala’s type inference as much as possible. Table 8.3 summarizes the mapping of \( \pi \)-calculus patterns that correspond to CLL inference rules to Scala functions. Note that PiTimes and PiWith are being reused from the set of functions used for atomic processes.

Code compositionality for \( \pi \)-calculus patterns that contain sub-processes is achieved through code templates that contain placeholders corresponding to each sub-process. For example, the Cut rule contains subprocesses \( F \) and \( G \), therefore the generated code template will contain two placeholders for the code corresponding to \( F \) and \( G \) respectively. We give more details and the corresponding generated code template for each of the patterns of Table 8.3 in the next sections.

8.4.2.1 The Axiom Buffer

The PAPPILIB implementation of the axiom buffer (i.e. the process translation of the identity axiom of CLL) is a straightforward combination of PiIn and PiOut within the PiId function shown in Figure 8.24.
### Table 8.3: Mapping of proofs-as-processes \(\pi\)-calculus patterns to Scala functions in PAPPiLib.

<table>
<thead>
<tr>
<th>Proofs-as-processes</th>
<th>(\pi)-calculus pattern</th>
<th>Scala function</th>
</tr>
</thead>
<tbody>
<tr>
<td>(I_{x,y})</td>
<td>(y(a).\overline{x}(a).0)</td>
<td>PiId</td>
</tr>
<tr>
<td>(\otimes\frac{xy}{z} (F, G))</td>
<td>((v x, y)(\overline{x}(x, y). (F</td>
<td></td>
</tr>
<tr>
<td>(\otimes\frac{xy}{z} (F))</td>
<td>(z(x, y). F)</td>
<td>PiParI</td>
</tr>
<tr>
<td>(L(P))</td>
<td>((v x)(z(u, v). \overline{u}(x). P))</td>
<td>PiPlus1</td>
</tr>
<tr>
<td>(R(Q))</td>
<td>((v y)(z(u, v). \overline{v}(y). Q))</td>
<td>PiPlus2</td>
</tr>
<tr>
<td>(\otimes\frac{xy}{z} (P, Q))</td>
<td>((v u, v)(\overline{u}(u, v). (u(x). P + v(y). Q)))</td>
<td>PiWith</td>
</tr>
<tr>
<td>(Cut^z (F, G))</td>
<td>((v z)(F[z/x]</td>
<td></td>
</tr>
<tr>
<td>(Assumption)</td>
<td>(P(x, y, z))</td>
<td>p.run(x, y, z)</td>
</tr>
</tbody>
</table>

```scala
def PiId[A](process: String)(in: NChan[A])(out: NChan[A]) = {  PiOut(process, (PiIn[A] (process) (in))) (out) }
```

**Figure 8.24:** Implementation of the \(\text{PiId}\) function for buffering in PAPPiLib.
8.4.2.2 The ‘tensor’ Pattern

The tensor (⊗) pattern matches the π-calculus translation of parallel output $A \otimes B$ in CLL for atomic processes (see Table 8.2). Therefore, the $\text{PiTimes}$ function (see Figure 8.15) can be reused here. The difference here is that we do not have specific resources to send. Continuation functions correspond to the Scala code generated for the involved processes $F$ and $G$ (see Table 8.3). In short, we parse $F$ and $G$, construct their code recursively, then feed it to $\text{PiTimes}$ in the form of continuation functions. The code template that accomplishes this is shown in Figure 8.25.

Note that underscores (_), are used in variable names in order to avoid variable capture (given that our CLL specifications forbid channel names that begin with an underscore). For example, the code generated for $F$ is expected to use $x$ as a free name (normally as an output channel). Therefore, the code for $F$ will assume $x$ be previously declared as such and is available to use, in a similar way that $z$, which is a free name for $\text{PiTimes}$, is assumed to pre-exist for this code to work (note that if $z$ is a free channel in the composition then it will end up as a parameter of the run function of the corresponding class, so there will always be an instance of $z$ for this code to use). However, since $x$ is locally bound (it is actually generated within $\text{PiTimes}$) it must not clash with any previous declarations of any other $x$ variable. This is why $x$ is declared as a local value within the definition of $\_p$, and so is $y$ within the scope of $\_q$. Moreover, the $\_x$ parameter for $\_p$ ensures it will also not clash with any other $x$ or the $x$ declared within $\_p$. The actual argument passed on to $\_p$ will be the channel $x$ constructed locally within $\text{PiTimes}$, since $\_p$ is fed as a continuation function argument to that.

Also note how channels are dynamically cast to their appropriate types, based on our justification in Section 8.2.4.

8.4.2.3 The ‘par’ Pattern

Similarly to the tensor pattern, the implementation of the par (⟨⟩) pattern closely resembles that of parallel input for atomic processes, the difference being the lack of continuation functions and, in this case, input synchronisation. The Par rule simply receives two resources through channel $z$ and passes them on to $F$. The corresponding Scala function, $\text{PiParI}$ is shown in Figure 8.26. The generated code template used for
8.4. From Proofs-as-processes to Functions

\[ \text{def } \_p \ (\_x : \text{NChan[Any]}) = \{ \]
\[ \quad \text{val } x = \_x.asInstanceOf/[\text{... type inferred from } F \text{ ...}]/] \]
\[ \quad \// \text{ ... code for } F \text{ ...} \]
\[ \} \]
\[ \text{def } \_q \ (\_y : \text{NChan[Any]}) = \{ \]
\[ \quad \text{val } y = \_y.asInstanceOf/[\text{... type inferred from } G \text{ ...}]/] \]
\[ \quad \// \text{ ... code for } G \text{ ...} \]
\[ \} \]
\[ \text{val } \_z = z.asInstanceOf[\text{PairChan[\text{NChan[Any]},NChan[Any]]}] \]
\[ \text{PiTimes(name,"x","y") (\_p) (\_q) (\_z) } \]

Figure 8.25: Code template for the \( \pi \)-calculus ‘tensor’ pattern \((\nu x,y) (\exists (x,y). (F |\mid G)) \) in PAPPILIB.

the Par rule pattern is shown in Figure 8.27

\[ \text{def PiParI[A,B](process: String) } \]
\[ \quad \text{(in: PairChan[\text{NChan[A]},NChan[B]]) = } \{
\[ \quad \text{rchoice ( in * ( case Pair(chan_a,chan_b) =>}
\[ \quad \quad (chan_a , chan_b ) )) } \]
\[ \} \]

Figure 8.26: Implementation of the PiParI function for the CLL Par rule in PAPPILIB.

\[ \text{val } \_z = z.asInstanceOf[\text{PairChan[\text{NChan[Any]},NChan[Any]]}] \]
\[ \text{PiParI(name) (\_z) match { case ( x , y ) => } } \]
\[ \quad \// \text{ ... code for } F \text{ ...} \]
\[ \} \]

Figure 8.27: Code template for the \( \pi \)-calculus ‘par’ pattern \( z(x,y).F \) in PAPPILIB.

As was the case for the Tensor rule, here too the \( z \) channel is cast to its appropriate type. Also note how \( x \) and \( y \) become available values for the code for \( F \) to use.

8.4.2.4 The ‘plus’ Patterns

The \( \oplus \) rules of CLL are easily translated to code using the PiPlus function for optional output in atomic processes. More specifically, the \( \pi \)-calculus pattern for the \( \oplus L \) (sym-
metrically \( \oplus R \) rule matches the \( \pi \)-calculus translation of \( A \oplus B \) in atomic processes (see Table 8.2) where the right (left) hand side of the summation is omitted. At the level of Scala code, this translates to the use of the \( \text{PiPlus} \) function where always a type \( \text{Left} \) continuation is given as an argument in the case of the \( \oplus L \) rule, or always a type \( \text{Right} \) continuation is given as an argument in the case of the \( \oplus R \) rule. This leads to the implementation of the \( \text{PiPlus1} \) and \( \text{PiPlus2} \) functions as shown in Figure 8.28.

```scala
def PiPlus1[A,B](process: String, chan_x: String)(out: OptChan[NChan[A],NChan[B]])(cont: NCont[A]) = {
    PiPlus (process, chan_x, "") (Left(cont)) (out)
}

def PiPlus2[A,B](process: String, chan_y: String)(out: OptChan[NChan[A],NChan[B]])(cont: NCont[B]) = {
    PiPlus (process, "", chan_y) (Right(cont)) (out)
}
```

Figure 8.28: Implementation of the \( \text{PiPlus1} \) and \( \text{PiPlus2} \) functions for the two CLL \( \oplus \) rules in \text{PAPPILIB}.

Notice that the types of both options \( A \) and \( B \) must be defined (or inferred), but only one continuation is provided, corresponding to process \( P \) for \( \oplus L \) or process \( Q \) for \( \oplus R \). The code template generated for the pattern of the \( \oplus L \) rule is shown in Figure 8.29. The code template for the \( \oplus R \) rule is similar, but uses \( \text{PiPlus2} \) instead of \( \text{PiPlus1} \).

```scala
val _z = z.asInstanceOf[OptChan[NChan[Any],NChan[Any]]]
PiPlus1(name, "x") (_z) ((__x : NChan[Any]) => {
    val x = __x.asInstanceOf[/*... type inferred from P ...*/]
    // ... code for P ...
})
```

Figure 8.29: Code template for the \( \pi \)-calculus pattern corresponding to the CLL \( \oplus L \) rule \((\nu x)(z(u,v).\pi(x).P)\) in \text{PAPPILIB}.

Note that \( x \) is dynamically cast to the appropriate type for use within the code for \( P \). The type is inferred during code generation as explained in Section 8.2.4.
8.4.2.5 The ‘with’ Pattern

Similarly to the \texttt{PiTimes} function, \texttt{PiWith} introduced in Section 8.4.1.6 is reused for the ‘with’ pattern. Instead of continuations for the two optional inputs, we have generated code for processes \( P \) and \( Q \) of the \( \pi \)-calculus pattern. The generated code template is shown in Figure 8.30.

\begin{verbatim}
def _p (__x : NChan[Any]) = {
val x = __x.asInstanceOf[/*... type inferred from P ...*/]
  // ... code for P ..
}
def _q (__y : NChan[Any]) = {
val y = __y.asInstanceOf[/*... type inferred from Q ...*/]
  // ... code for Q ..
}
val _z = z.asInstanceOf[OptChan[NChan[Any],NChan[Any]]]
PiWith(name, "u", "v") (_p) (_q) (_z)
\end{verbatim}

Figure 8.30: \texttt{PAPPILib} code template for the \( \pi \)-calculus ‘with’ pattern, i.e. the term 
\[ (v \ u, v)(\exists u, v). (u(x). P + v(y). Q)). \]

8.4.2.6 The ‘cut’ Pattern

The CLL \texttt{Cut} rule is the main communication rule in the context of proofs-as-processes (see Section 4.3). In \texttt{PAPPILib}, the corresponding pattern is implemented as the \texttt{PiCut} function shown in Figure 8.31. Note that continuations \texttt{cont\_p} and \texttt{cont\_q} correspond to processes \( F \) and \( G \) respectively.

\begin{verbatim}
def PiCut[A](process: String, name_z: String)
  (cont_p: NCont[A]) (cont_q: NCont[A]) = {
val z = new NChan[A](name_z);
spawn < cont_p(z) | cont_q(z) >
}
\end{verbatim}

Figure 8.31: Implementation of the \texttt{PiCut} function for the CLL \texttt{Cut} rule in \texttt{PAPPILib}. 
Chapter 8. Automated Workflow Deployment

The code template generated by the π-calculus pattern for the Cut rule is shown in Figure 8.32.

```scala
def _p (__x : NChan[Any]) = {
    val x = __x.asInstanceOf[/*... type inferred from F ...*/]
    // ... code for F ...
}

def _q (__y : NChan[Any]) = {
    val y = __y.asInstanceOf[/*... type inferred from G ...*/]
    // ... code for G ...
}

PiCut(name,"z") (_p) (_q)
```

Figure 8.32: Code template for the π-calculus pattern corresponding to the CLL Cut rule $Cut_z(F,G)$ in PAPPiLib.

8.4.2.7 Components

In Section 7.1, we described our composition goal as the proof of the specification of the requested process using the available processes as assumptions. As a result, assumption matching in these proofs corresponds to the call to one of the available processes $P$ with specified channels (e.g. $x$, $y$, and $z$ in the example process with 3 free channels shown in Table 8.3) as parameters.

Since all the component processes are members of the class of the composite process, the above can be translated to Scala code in a straightforward way as a call to the run function of the corresponding component $P$. As mentioned in Section 8.3.2, composite processes can be used as components in further compositions (in the same way as atomic processes) since they also have a run function.

8.5 The Credit Card Example in Scala Code

In this section, we wrap up the description of our automated workflow deployment procedures by providing a concrete example for the CreditCardPayment workflow as presented in Section 7.5.
Figures 8.33 and 8.34 present the generated Scala traits for the three available processes CreditCardInit, UserPINInput, and CreditCardTransaction. The code is generated based on the π-calculus specifications shown in Figure 6.2. Note that all the involved types are declared as abstract (but these declarations are not shown in the figures).

For the construction of the CreditCardPayment workflow we followed Strategy A as shown in Figure 7.11 with the corresponding proof from Figure 7.14. The generated class is shown in Figures 8.35 and 8.36. It is worth noting that even for a small example such as the CreditCardPayment composition the generated code is fairly lengthy and complicated to follow. However, it is generated fully automatically, and the user does not need to read or edit it in any way.

The user is only required to provide concrete types for all the involved type declarations and a concrete body for each available process. The system facilitates this procedure by automatically generating the code template shown in Figure 8.37. The types are all instantiated as Scala String, and a template class for each available process is generated. The user is free to edit both the types and the classes in any desired way, as long as the declarations of the apply methods for each class remain valid. The types can be arbitrarily complicated Scala (or Java) classes or traits/interfaces, and the available processes can be implemented in any desired way, from making simple calculations to interfacing with external Web Services, or retrieving user input.

The system also generates a main function as a template to execute the deployed workflow. The function introduces objects that are instances for each class of available processes. It then constructs an instance of the CreditCardPayment class with these objects as parameters and demonstrates the means of executing the workflow with a sample input.

8.6 Conclusion and Future Work

Workflow deployment is an important step in a process modelling approach. Our goal is a systematic, automated procedure that allows design properties of the modelled workflow to persist in its deployment and minimizes the required human effort in order to maximize deployment speed and maintainability.

Scala’s PiLIB library enables a direct implementation of π-calculus processes as Scala
trait CreditCardInit extends ((PAYMENT_REQ) => Pair[TRANSACTION,PIN_REQ]) {
  var name = "CreditCardInit"
  def run(cui_out : PairChan[NChan[Any],NChan[Any]], cui_pr : NChan[Any]) = {
    val inputf0 = PiIn[Any](name)_
    val input0 = future(inputf0 (cui_pr))
    val outputf = (this ((input0 ()).asInstanceOf[PAYMENT_REQ]
      ).asInstanceOf[Any]) match {
      case (x,y) =>
        val xout = PiOut(name,x)_
        val yout = PiOut(name,y)_
        PiTimes(name, "cci_out_a", "cci_out_b") (xout) (yout) _
    }
    outputf (cui_out)
  }
}

trait UserPINInput extends ((PIN_REQ) => PIN) {
  var name = "UserPINInput"
  def run(uni_out : NChan[Any], uni_req : NChan[Any]) = {
    val inputf0 = PiIn[Any](name)_
    val input0 = future(inputf0 (uni_req))
    val outputf = PiOut(name, (this ((input0 ()).asInstanceOf[PIN_REQ]
      ).asInstanceOf[Any])_
    outputf (uni_out)
  }
}

Figure 8.33: Generated Scala code for the available processes CreditCardInit and UserPINInput of the credit card example.
trait CreditCardTransaction
    extends ((TRANSACTION, PIN) => Either[PAYMENT,EX_BAD_PIN]) {
        var name = "CreditCardTransaction"

        def run(cct_out : OptChan[NChan[Any],NChan[Any]],
            cct_tr : NChan[Any], cct_pin : NChan[Any]) = {
            val inputf0 = PiIn[Any](name)_
            val input0 = future(inputf0 (cct_tr))
            val inputf1 = PiIn[Any](name)_
            val input1 = future(inputf1 (cct_pin))
            val outputf = {
                val xout = (this ((input0 ().asInstanceOf[TRANSACTION],
                    (input1 ().asInstanceOf[PIN] :
                    ).asInstanceOf[Any]) match {
                    case Left(x) =>
                        val xout = PiOut(name,x)_
                        Left(xout)
                    case Right(x) =>
                        val xout = PiOut(name,x)_
                        Right(xout)
                }
                PiPlus(name, "cct_out_x", "cct_out_y") (xout)_
            }
            outputf (cct_out)
        }
    }

Figure 8.34: Generated Scala code for the CreditCardTransaction process of the credit card example.
class CreditCardPayment (creditCardTransaction : CreditCardTransaction, userPINInput : UserPINInput, creditCardInit : CreditCardInit) extends ((PAYMENT_REQ) => Either[PAYMENT,EX_BAD_PIN]) {
  var name = "CreditCardPayment"

  def run(cct_out_ : OptChan[NChan[Any],NChan[Any]], cci_pr_ : NChan[Any]) = {
    def _p (__cci_out : NChan[Any]) = {
      val cci_out = __cci_out.asInstanceOf[PairChan[NChan[Any],NChan[Any]]]
      creditCardInit.run(cci_out, cci_pr_)
    }
    def _q (__cci_out : NChan[Any]) = {
      val cci_out = __cci_out.asInstanceOf[PairChan[NChan[Any],NChan[Any]]]
      val _cci_out = cci_out.asInstanceOf[PairChan[NChan[Any],NChan[Any]]]
      PiParI(name) (_cci_out) match { case (cct_tr, upi_req) => {
        def _p (__upi_out : NChan[Any]) = {
          val upi_out = __upi_out.asInstanceOf[NChan[Any]]
          userPINInput.run(upi_out, upi_req)
        }
        def _q (__cct_pin : NChan[Any]) = {
          val cct_pin = __cct_pin.asInstanceOf[NChan[Any]]
          creditCardTransaction.run(cct_out_, cct_tr, cct_pin)
        }
        PiCut(name,"z2") (_p) (_q)
      }
    }
    PiCut(name,"cci_out") (_p) (_q)
  }

  //... apply method ...
}

Figure 8.35: Generated Scala code for the composite process CreditCardPayment. The apply method is shown in Figure 8.36 as it could not fit within the size of this page.
override def apply ( arg0 :PAYMENT_REQ ) :Either[PAYMENT,EX_BAD_PIN] = {
    def request(cci_pr_ : NChan[Any]) = {
        val outputf0 = PiOut(name,(arg0.asInstanceOf[Any]))._
        spawn < outputf0 (cci_pr_) >
    }
    def response(cct_out_ : OptChan[NChan[Any],NChan[Any]]) = {
        val inputf = {
            val xin = PiIn[Any](name)._  
            val yin = PiIn[Any](name)._  
            PiWith(name, "cct_out__u", "cct_out__v") (xin) (yin) _
        }
        val input = future(inputf (cct_out_))
        (input ().asInstanceOf[Either[PAYMENT,EX_BAD_PIN]])
    }
    val cci_pr_ = new NChan[Any]("cci_pr_")
    val cct_out_ = new OptChan[NChan[Any],NChan[Any]]("cct_out_")
    scala.concurrent.ops.spawn(request(cci_pr_))
    scala.concurrent.ops.spawn(this.run(cct_out_ , cci_pr_))
    (future (response(cct_out_))) ()
}

Figure 8.36: Generated Scala code for the apply method of the class corresponding to composite process CreditCardPayment.
type EX_BAD_PIN = String
type PAYMENT = String
type PAYMENT_REQ = String
type PIN = String
type PIN_REQ = String
type TRANSACTION = String

class CreditCardInitInstance extends CreditCardInit {
  override def apply( arg0 :PAYMENT_REQ ) :Pair[TRANSACTION,PIN_REQ] = {
    // TODO: Instantiate this method.
  }
}

class CreditCardTransactionInstance extends CreditCardTransaction {
  override def apply( arg0 :TRANSACTION,
    arg1 :PIN ) :Either[PAYMENT,EX_BAD_PIN] = {
    // TODO: Instantiate this method.
  }
}

class UserPINInputInstance extends UserPINInput {
  override def apply( arg0 :PIN_REQ ) :PIN = {
    // TODO: Instantiate this method.
  }
}

def main(args: Array[String]): Unit = {
  val creditCardInit = new CreditCardInitInstance
  val creditCardTransaction = new CreditCardTransactionInstance
  val userPINInput = new UserPINInputInstance
  val creditCardPayment =
    new CreditCardPayment(creditCardTransaction , userPINInput , creditCardInit)

  val payment_request = ""
  // TODO: Provide actual parameters:
  val result = creditCardPayment( payment_request )
}

Figure 8.37: Generated Scala code template for the implementation of the available processes in the credit card example and the execution of the CreditCardPayment workflow.
functions. Based on this we have systematically mapped π-calculus patterns used in atomic process specifications as well as those appearing in proofs-as-processes based compositions to executable code templates. This enables the automatic translation of both atomic and composite process specifications to executable code.

More specifically, atomic processes are deployed as abstract traits, which handle the π-calculus based communication automatically. Given that the process specifications are abstract and the processes themselves are considered black-boxes, the user is expected to provide concrete implementations for each process such that the given specification is satisfied. Moreover, composite processes are deployed as classes with their component processes as members.

Both atomic and composite processes are deployed in such a way that they can be used both as Scala functions, with a type specification corresponding to the CLL type specification of the process, and as π-calculus agents, with the required π-calculus names as arguments.

There are a number of further improvements that can be applied to the current workflow deployment library. Firstly, there are a number of code optimisations that can be used to improve upon the existing code templates being generated. For example, we have observed cases of unnecessary repetitions of the same type casting. Another interesting idea for further practical improvements, is the addition of more meta-data in each atomic process specification. This will allow, for example, the identification of certain processes as web services, so that the Java Enterprise Edition web services toolkit can be used to automatically connect a process to the corresponding, live web service. Other processes could be tagged as Human Provided Services (HPSs). As part of an ongoing side project, we are considering an automated algorithm for checklist generation from process specifications of HPSs. Finally, as part of further work, we plan to add more functionality to the executed workflows, such as live visualisation (as an extension to logging), monitoring, persistence across crashes and reboots, simulation, etc., as well as migrating our implementation to the actively maintained Akka library for actor-based concurrency in Scala.
Chapter 9

Use Case: Holiday Booking Web Agent

We proceed to describe a use case of a holiday booking web agent for our process composition framework. We give a breakdown of the problem and the processes involved and demonstrate some of the properties and benefits gained from our methodology.

9.1 Introduction

Our approach to workflow composition allows us to use as building blocks any kind of processes that can be described abstractly in terms of inputs, outputs, preconditions, and effects (IOPE’s) as building blocks. Since our framework is agnostic with regards to any other properties of the involved components, it can be used in a wide variety of cases involving many different types of workflows. The components, for example, may be either automated programs or human provided services (HPS’s), and they may have different levels of coupling, from tightly coupled program blocks to loosely coupled API calls or even completely independent web services from different providers. As a result, the resulting workflow may play a variety of roles, including, for example, that of a synthesized program, a complex procedure/part of a larger system, a business process model, or a (semantic) web agent [Antoniou and van Harmelen, 2004].

Among the various possible uses of our workflow models, we explore one that may be quite relevant to the casual computer user, namely the construction of a verified web agent. This involves the construction of an agent that acts as a proxy for the user on the web by accessing information from various web services and online APIs, filtering it, and then providing it to the user in a tailored fashion.
Note that the idea of a personal semantic web agent is a major part of the Semantic Web vision (Antoniou and van Harmelen, 2004), although in our approach we are not necessarily restricted to semantic web services. The development of composable web agents is aimed at exploiting the computer-based accessibility of web applications in today’s Web 2.0 (O’Reilly, 2005) in order to facilitate user tasks that involve many different applications from multiple providers. Instead of having to visit a variety of web pages and interact with all the different providers while tracking intermediate information manually, the user can simply run a personalized web agent and interact with it as a single, local process. Our logic-based approach allows the dynamic construction of such workflow-based web agents in the form of composite services, while offering guarantees of correctness, systematic accounting of information, explicit handling of exceptions, and concurrent execution.

In our running example of the CreditCardPayment workflow (see Section 2.1.3), the CreditCardInit and CreditCardTransaction components are viewed as secure calls to the API of the corresponding credit card company, whereas UserPINInput is viewed as an HPS that requires user input. The resulting CreditCardPayment can be used as a generic web agent of credit card payments.

The Ski example introduced by Rao et al. (see Sections 5.2.1 and 7.6) can also be viewed as a simple web agent capable of retrieving the price of an appropriate ski set based on the user’s criteria and the user’s local currency, by interacting with 5 different web services.

In a similar fashion, we also constructed a workflow for a web agent for home purchasing based on an existing paper-based breakdown of the involved web services (Zhang et al., 2005). This proved to be a helpful, non-trivial use case especially during the main development stage of our system. It involved 10 components, including web services such as a real-estate search engine and a mortgage application service, and HPS’s whenever user input is required, such as for the selection of a suitable property from the search results. We will not be examining this case-study further here, but for a more detailed analysis we refer the interested reader to our corresponding publication (Papapanagiotou and Fleuriot, 2011).

In this chapter, we describe a larger, more complicated example of web services composition to demonstrate the scalability of our approach. We investigate the case of a web agent as a composite service that allows the user to go through all the necessary
steps for booking their holidays. In particular, the user may want to book a flight, a hotel, rent a car, and arrange secure payments for these bookings using their credit card.

We begin by introducing the atomic component services in Section 9.2, then proceed to demonstrate some of the steps towards building a single composite service that suits our needs in Section 9.3. Finally, in Section 9.4, we describe the results extracted from our framework, based on a single composition strategy, including the executable model of the composite service.

9.2 Component Services

In a typical example of holiday planning, one would have to arrange the means of travelling to the destination, accommodation, and the means of local transportation. In this example, we assume that the user prefers to fly, stay in a local hotel, and rent a car for their local transportation. In order to satisfy these requirements, the user would expect to be able to search for the most suitable offers with regards to specific criteria, and pay securely using their credit cards.

In order to accomplish this task, a number of web services, possibly developed by several independent companies, will need to be called upon. As discussed in Section 6.2, these services can be described in a straightforward, abstract way as processes based on their inputs, outputs, preconditions, and effects. We split our component web services based on their functionality (flight booking, hotel booking, or car rental), describe each one of them briefly, and present their specification using our methodology in Figures 9.1, 9.2, 9.3, 9.4, and 9.5.

We assume that all pre-payments are made with the use of a credit card. For this purpose, we use the composite service CreditCardPayment as presented in Section 7.5.1. We discuss this in more detail in Section 9.3.1.

It is immediately obvious that the large number of involved services makes the task of booking for vacations particularly tedious for the average user. They are forced, for example, to spend a lot of time visiting the variety of websites hosting these services and the whole booking process results in much redundancy when inputting information and in the risk of mistakes, thereby producing an inconsistent set of bookings. Composing the services in a single workflow allows the user to only interact with a single composite service and share the necessary information such as locations, dates,
Chapter 9. Use Case: Holiday Booking Web Agent

**FlightAgent**

⊢ DATE_TO ⊥, DATE_FROM ⊥, FL_TO ⊥, FL_FROM ⊥, FL_QUERY

This web service provides an interface for searching flights from a variety of airlines. The user can provide their source location FL_FROM and their destination FL_TO, as well as the date they wish to travel DATE_FROM and return DATE_TO. Based on this information, the service provides an appropriately constructed, machine readable query FL_QUERY that can be used to search using various agents.

**FlightSearch**

⊢ FL_QUERY ⊥, FL_LIST ⊕ (EX_FL_NOT_FOUND ⊗ FL_QUERY)

This web service corresponds to a search engine for flight carriers. Given a query FL_QUERY, it retrieves a list of matching available flights FL_LIST from various airlines or an exception EX_FL_NOT_FOUND paired with the original query FL_QUERY, if no flights were found. Notably, this particular search engine does not search low budget airlines. This may be useful for users who require a higher standard of flight service.

Figure 9.1: Web services for booking flights (1 of 2).
9.2. Component Services

SelectFlight

⊢ \(FL_LIST\),

\((FL_SELECTION \otimes FL_PREFS \otimes CLIENT_INFO \otimes PAYMENT_REQ) \oplus FL_REJECT\)

This service corresponds to a flight selection by the user based on the list of available flights \(FL_LIST\). It outputs the user selection \(FL_SELECTION\) and the corresponding payment request information \(PAYMENT_REQ\) that they submitted. The user also needs to provide their flight preferences \(FL_PREFS\) (e.g. whether they want to add extra luggage, or prefer a window or aisle seat, etc.) as well as their personal information \(CLIENT_INFO\). Alternatively, the user may reject all of the listed flights \(FL_REJECT\).

FlightBooking

⊢ \(PAYMENT\), \(CLIENT_INFO\), \(FL_PREFS\), \(FL_SELECTION\), \(FL_BOOKING\)

Given the user selection \(FL_SELECTION\), preferences \(FL_PREFS\), and client information \(CLIENT_INFO\), as well as proof of payment \(PAYMENT\), this web service finalizes the flight booking \(FL_BOOKING\).

Figure 9.2: Web services for booking flights (2 of 2).
Chapter 9. Use Case: Holiday Booking Web Agent

This web service corresponds to a search engine for hotels. The user can provide the desired location $HL_{LOCATION}$, date of arrival $DATE_{FROM}$ and departure $DATE_{TO}$ as well as extra specified preferences $HL_{PREFS}$ (such as hotel star rating, availability of various amenities, etc.). This results in a list of available hotels $HL_{LIST}$ or an exception $EX_{HL_{NOT\_FOUND}}$ if none is found.

The user selection of the desired hotel is modelled through this web service. Given a list of hotels $HL_{LIST}$ the user may make a selection of one of them $HL_{SELECT\_ION}$ and provide their personal information $CLIENT_{INFO}$ for the booking. Naturally, they have the option to reject all the hotels in the list $HL_{REJECT}$. Note that $CLIENT_{INFO}$ also contains credit card information that is necessary in order to confirm a booking in most hotels.

This web service finalizes the booking $HL_{BOOKING}$ for a particular hotel selected by the user $HL_{SELECT\_ION}$, using their personal information $CLIENT_{INFO}$ and preferences $HL_{PREFS}$. We assume that, unlike flight bookings, hotel bookings are not prepaid.

Figure 9.3: Web services for booking hotels.
When renting a car, there is a limited number of available locations to pick up and return the rented car. This service provides a search engine for such car pick up \( CAR\_PICK\_UP\_LIST \) and drop off \( CAR\_DROP\_OFF\_LIST \) points given a particular location \( CAR\_LOCATION \).

Given lists of available pick up and drop off points, the user can use this service to select the most suitable ones, \( CAR\_PICK\_UP \) and \( CAR\_DROP\_OFF \).

This web service is a search engine for rental cars, given a selected pick up \( CAR\_PICK\_UP \) and drop off location \( CAR\_DROP\_OFF \), as well as the desired rental dates \( DATE\_FROM \) and \( DATE\_TO \). The result is either a list of available cars \( CAR\_LIST \) or an exception \( EX\_CAR\_NOT\_FOUND \) if no available cars were found for these particular preferences.

Figure 9.4: Web services for renting a car (1 of 2).
Chapter 9. Use Case: Holiday Booking Web Agent

SelectCar

⊢ CAR_LIST ⊥, (CAR_SELECTION ⊗ CAR_PREFS ⊗ CLIENT_INFO ⊗ PAYMENT_REQ) ⊗ CAR_REJECT

This service models the user’s car selection from the list of available cars CAR_LIST. Apart from the selection CAR_SELECTION it also outputs and the corresponding payment request information PAYMENT_REQ submitted by them. The service also requires the user’s personal information CLIENT_INFO as well as any individual preferences CAR_PREFS they may have (e.g. extra insurance or extra equipment such as navigational systems, child safety seats, etc.) as input. Alternatively, the user may reject CAR_REJECT the entire list of suggested cars.

CarBooking

⊢ PAYMENT ⊥, CLIENT_INFO ⊥, CAR_PREFS ⊥, CAR_SELECTION ⊥, CAR_BOOKING

This web service finalizes the booking for the car rental CAR_BOOKING based on the selection CAR_SELECTION, user preferences CAR_PREFS, and personal information CLIENT_INFO of the user. Proof of payment PAYMENT is also required to complete the process.

Figure 9.5: Web services for renting a car (2 of 2).
and preferences across the workflow, instead of jumping from site to site.

9.3 Composing Services

Once we have introduced all the available services using their CLL specifications, we can start composing them with the aim of constructing a single composite service that achieves the entire holiday booking. For simplicity we use the system’s discovery mode (see introduction to Section 7.5.1) throughout, so that we do not specify any restrictions on our goal.

Using our graphical interface, we form connections and compositions of the available services with relative ease. Our HOL Light backend tracks all the information (including alternative outcomes such as exceptions) automatically and guarantees the correctness of the information flow being constructed via CLL proofs.

Analysing the entire composition for the holiday booking example in detail would require considerable effort and will not be covered here. However, we focus on two particular examples of partial compositions to demonstrate some of the properties of our system.

9.3.1 Example of Compositionality: Credit Card Payments

In the credit card example from Section 7.5.1, we constructed the composite service CreditCardPayment for arranging secure credit card payments. The system now allows us to reuse this service as a component in our more complex holiday booking example. As soon as the CreditCardPayment is constructed, it becomes an available service that can be used in other compositions. The workflow of the reused composite service is hidden (with CreditCardPayment appearing as shown in Figure 7.11d), but the user may still examine its sub-components if necessary.

In the example of the payment for the flight booking, the FlightBooking service expects the proof of payment input called PAYMENT in order to confirm the booking (see Figure 9.2). This can be provided by the CreditCardPayment service that has PAYMENT as an output. We can, thus, compose these two services using the JOIN action (see Section 7.4.3). The result is shown in Figure 9.6a. Note that, in the case
where an \texttt{EX\_BAD\_PIN} exception is issued, this is forwarded to the output of this composition together with any unused resources that were inputs to \texttt{FlightBooking}, ie. \texttt{FL\_SELECTION}, \texttt{FL\_PREFS}, and \texttt{CLIENT\_INFO} since, as mentioned previously, the underlying logic ensures that no resources disappear unless they are consumed by a service. In our example, these resources could be consumed by an exception handler that deals with \texttt{EX\_BAD\_PIN}, for instance by allowing the user a second attempt to enter the correct PIN.

As explained in Section 7.4, every composition action in our system corresponds to a series of applications of CLL inference rules. In this particular situation, the applied \texttt{JOIN} action constructs the proof tree shown in Figure 9.7 automatically in the background. The procedure that generates this proof is described in detail in Section 7.4.3.3. Note, for example that subproof (9.2) constructs a parallel buffer, and is itself generated by \texttt{PARBUF\_TAC} (see Section 7.3.2). The proof tree of the \texttt{JOIN} action is given here to display the relative complexity of the logical proof being performed and the reader is not expected to go through it in detail.

The inputs of this intermediate composition now match the outputs of \texttt{SelectFlight} (see Figure 9.2). Using the \texttt{JOIN} tactic once more, we can connect these, obtaining the result shown in Figure 9.6b. In this, a third possible (optional) output is introduced, namely that of type \texttt{FL\_REJECT} corresponding to the rejection of the search result by the user in \texttt{SelectFlight}.

It is worth noting that, once the \pi-calculus term of the composition is extracted and the part corresponding to the composition of Figure 9.6b is executed, the \pi-calculus workflow of \texttt{CreditCardPayment} is naturally expanded, showing its atomic components. This can be seen in the PiVizTool screenshot of Figure 9.8.

### 9.3.2 Example of Exception Handling: Flight Search

The \texttt{FlightSearch} service, as described in Figure 9.1, is a search engine for available flights from a number of possible airline databases. It returns either a list of flights \texttt{FL\_LIST} or an \texttt{EX\_FL\_NOT\_FOUND} exception if none were found. In the latter case, the exception will normally be forwarded to the user as an output. However, it also possible for the designer of the composite service to add a handler for this particular exception. In our example, we assume that if no available flights are found,
9.3. Composing Services

(a) Result of \texttt{JOIN} between \texttt{CreditCardPayment} and \texttt{FlightBooking}.

(b) Result of further \texttt{JOIN} with \texttt{SelectFlight}.

Figure 9.6: Example of composing using \texttt{CreditCardPayment} as a component.
Figure 9.7: The CLL proof constructed by the composition of CreditCardPayment and FlightBooking with the JOIN tactic.

(9.1) \[ \text{CreditCardPayment} \] (9.2) \[ \text{FlightBooking} \] (9.3) \[ \text{JOIN} \]
9.3. Composing Services

Figure 9.8: Snapshot of the PiVizTool simulation of the execution of a composition containing CreditCardPayment. The composite service is automatically split into its components.

an alternative search engine can be used that examines low budget airlines. We assume the user prefers a higher standard airline by default, unless there are no available flights with his search criteria.

We call our exception handler FlightSearchLowBudget. As a service, it is similar to FlightSearch, but searches low budget airlines instead. It accepts an exception of type EX_FL_NOT_FOUND as an input and the failed search query FL_QUERY, it reruns the query on its own database, and returns either a list of available flights FL_LIST or a new EX_FL_NOT_FOUND exception if no flights are found. The graph of FlightSearchLowBudget is shown in Figure 9.9. The FlightSearchLowBudget service can be composed with FlightSearch so that it handles the latter’s exception using the the JOIN action. The result is shown in Figure 9.10

Figure 9.9: The FlightSearchLowBudget exception handler.

Figure 9.10: Using FlightSearchLowBudget as an exception handler for FlightSearch. If no exception is thrown, FL_LIST is merely forwarded (grey edge).

The behaviour of the composed service can be examined in the PiVizTool simulation as shown in Figures 9.11 and 9.12. The initial state in the two figures is the point of execution where FlightSearch will either return a list FL_LIST that was found...
Chapter 9. Use Case: Holiday Booking Web Agent

Figure 9.11: Simulation of the BookAnyFlight composition using the PiVizTool: FlightSearch returns a list of available flights.

In Figure 9.11 we simulate the case where a flight is found by FlightSearch (i.e. we select the reduction corresponding to \(co5_x\)). This leads to an \(fl\_list\) being forwarded directly to the composition. Since there is no exception, FlightSearchLowBudget is never called by BookAnyFlight, and the \(fl\_list\) is simply forwarded through it, via the \(a14\) buffer edge, to SelectFlight. Note that \(co7_x\) and \(co7_y\) are the (system generated) channels corresponding to the two possible outputs of SelectFlight, i.e. either a flight selection (and accompanying resources) from the list or \(FL\_REJECT\) if all the flights are rejected by the user.

In Figure 9.12 we simulate the case where FlightSearch throws an exception of type \(EX\_FL\_NOT\_FOUND\) through channel \(co5_y\). The exception, as well as the original
Figure 9.12: Simulation of the BookAnyFlight composition using the PiVizTool: FlightSearch throws an EX_FL_NOT_FOUND exception.
are now handled by FlightSearchLowBudget. This, in turn, has two possible outcomes: either we find an appropriate list of flights \( FL\_LIST \) from low budget airlines (\( co6\_x \) edge) or another \( EX\_FL\_NOT\_FOUND \) exception is thrown (\( co6\_y \) edge). In this simulation, we assume a flight list is found by FlightSearchLowBudget and directly forwarded to SelectFlight.

### 9.4 Results

The final composition for holiday bookings consists of a single service that can be used to book flights, hotels, and car rentals concurrently. Our main strategy is to compose three main components, namely BookFlight, BookHotel, and BookCar that are responsible for each of these tasks respectively. Depending on our system design choices, we may replace BookFlight with BookAnyFlight, which takes low budget airlines into consideration through the exception mechanism described in the previous section.

To complete our BookHoliday service, the three main components are then composed in parallel. Copy nodes (see Section 6.2) are also used in order to avoid redundancy in the user input. They are used, for example, to duplicate the dates of arrival \( DATE\_FROM \) and departure \( DATE\_TO \) so that the end-user only needs to input them once and they are shared among all the services that require them (FlightAgent, HotelSearch, and CarSearch). Due to space and formatting limitations it is not possible for us to provide the relatively complicated diagram corresponding to the full composition. However, screenshots of the main constructed components mentioned above are shown in Figure 9.13.

The proof constructed by our automated HOL Light tactics in the background consists of 136 CLL inference steps. The resulting \( \pi \)-calculus model uses a total of 373 distinct channels and names. The PiVizTool simulation, however, gives us a much more intuitive picture of the behaviour of our composed service. An exceptionless simulation of the model takes around 98 (\( \pi \)-calculus reduction) steps to complete.
Figure 9.13: BookFlight, BookAnyFlight, BookHotel, and BookCar as composed services.
Chapter 9. Use Case: Holiday Booking Web Agent

9.5 Conclusion

This concludes our analysis of a travel agent use-case. It is worth noting that a number of existing travel agencies already provide composite services for holiday bookings. However, our approach provides a number of advantages, including the following:

- The specifications of the component services are abstract and based solely on inputs and outputs allowing maximum flexibility. Any service that matches this abstract specification can be used at runtime. For example, this allows for any search engine for flights that follows the specification of FlightSearch to be used, including search engines for different airlines, or even a composite search engine that goes through multiple airlines’ databases.

- The information flow of the constructed composite service is fully verified, and all the information is tracked explicitly. This provides a level of trust for the information exchanges that happen during runtime, including explicit tracing of sensitive information such as the client’s personal or credit card information. For example, it can be easily verified that the credit card information is never given to the individual airlines or car rental companies, but only supplied to the CreditCardPayment service to perform the payment, and that the this information will only be requested once at the appropriate stage of the entire process.

- The constructed composite service is described as a concurrent system, where independent components may run in parallel. Moreover, based on the underlying theory of proofs-as-processes, the composition is guaranteed to avoid any deadlocks and livelocks. For example, if the processing of the payment for the flight booking is being delayed, the execution may continue so that the user can proceed with their car rental and hotel bookings.

- Exceptions are handled or forwarded explicitly. For example, if a flight with the desired user preferences was not found, it is guaranteed that the corresponding exception will be forwarded to the user. Furthermore, it is possible to handle that exception within the system enabling a different course of action to be taken automatically (see Section 9.3.2).

- Finally, our automated workflow deployment procedures in conjunction with the graphical user interface allow our constructed workflows to be highly dynamic. As long as there exist implemented web services that match the specifications of
the introduced components, new workflows can easily be generated based on the individual needs of the user. For example, it is trivial to construct a workflow that only books a hotel and a flight if the user does not require a car rental. In addition, such new workflows can be automatically converted to executable code, as described in Section 8.3.2, thus offering a pragmatic and efficiently deployed solution.

In summary, we have demonstrated the usability of our logic-based methodology for designing web agents as compositions of web services. The final result is a type-checked, concurrent, and deadlock-free composition. Moreover, CLL ensures that all resources are accounted for, thus producing a fully verified information flow as well as guaranteeing proper exception handling. Using PiVizTool we can simulate the execution of a composed service and monitor the information being exchanged between the component services step by step in a visual fashion.

We demonstrated the scalability of our implemented system by describing several aspects of this particular use-case that involves 15 component services for booking flights, hotels, renting cars, and making secure credit card payments. We focused on the compositionality of our approach, as well as a mechanism for exception handling.

The systematic construction of the web agent workflow with its associated, formally verified properties add significant advantages and value to the end result and offer a high level of trust, not only for the buyer as the primary user, but also for all involved stakeholders.
Chapter 10

Formal Verification of Collaboration Patterns in Healthcare

In this chapter, we review the use of our formal process modelling framework in the context of collaboration patterns in healthcare. This particular application offers a solution to pragmatic, real-world problem by following a BPM-inspired approach (see Section 2.1.2). It has the potential of reducing preventable medical errors that are, in fact, one of the primary causes of deaths in hospitals (American Association for Justice, 2013).

10.1 Introduction

Our aim is to exploit its capabilities in order to offer a complete solution to the problem of modelling and deploying patterns of collaborative work in healthcare, where specified guidelines and best practices are enforced. We target both the design phase, where the guidelines are modelled using our diagrammatic interface, and the implementation phase, where the end result is a running, web based software system. In this context, our approach can offer multiple benefits. More specifically, some of the added value in the design phase consists of the following:

1. An intuitive, diagrammatic user interface that clinicians with limited knowledge of workflows and process composition can easily learn to use.

2. Information is tracked automatically and explicitly. The user does not need to
worry about accounting for all the inputs and outputs in every single case, especially in the highly complex hospital cases where multiple resources need to be accounted for at each stage.

3. Preconditions and effects are verified formally without the need for human verification. Clinical guidelines can thus be modelled in a way such that their proper enforcement can be taken for granted.

4. The framework offers compositionality so that the modelling can be performed at different levels of abstraction, thus simplifying the task.

Similarly, multiple benefits are gained at the implementation phase, some of which are listed below:

1. The clinical policies modelled in the design phase are automatically and formally propagated to the implemented system.

2. Workflow deployment is fully automatic, assuming the involved components are already implemented according to their specification. Changes in the policies are easily propagated to the system offering optimal maintainability.

3. The involved processes are abstract. With appropriate interfacing (e.g. through a web-based API) the workflow can be attached to any modern software system, including systems for automated documentation.

4. The deployed workflow takes care of all communications in an asynchronous way. The users (typically medical staff) need not memorize complex and often informal guidelines about who to contact, at what stage, and what information to send. They can then in principle focus on delivering the healthcare related services rather than the involved administrative procedures.

5. The concurrent, asynchronous execution of the workflow offers increased efficiency since independent processes are executed in parallel.

The motivation behind this work is discussed in more detail in Section 10.2. In particular, we focus on two patterns of collaborative work in healthcare, namely assignment and delegation, as described in Section 10.3. We explain the mapping of keystones to processes (see Section 10.3.2) and the constructed process compositions that correspond to the two collaborative patterns (see Section 10.3.3). We proceed to describe some of the aspects of modelling collaboration patterns within our formal framework.
in more detail in Section 10.4, including the mechanism used to enforce hospital policies and guidelines on the constructed patterns in Section 10.4.1, an analysis of the simulation capabilities of our system in Section 10.4.2, the exception handling mechanisms in Section 10.4.3, and the enhancement of our specifications with ontological classes in Section 10.4.4. In Section 10.5, we give some technical details of the deployment and implementation of the constructed patterns as workflows in a web-based software system. We conclude with a brief overview of relevant work in Section 10.6 and an overview of some related ongoing and future work in Section 10.7.

We note that most of the work presented in this chapter has been published in the form of 2 peer reviewed scientific papers, in the 25th IEEE International Symposium on Computer-Based Medical Systems (CBMS 2012) (Papapanagiotou et al., 2012a) and a special issue of the Behaviour & Information Technology journal (Papapanagiotou and Fleuriot, 2013) respectively.

### 10.2 Motivation

Recent research has shown the necessity for computer-based information systems in healthcare in order to reduce the likelihood of medical errors (Woolf et al., 2004). One important area is collaboration among medical staff. In particular, patient handover is a crucial process that requires consistency and completeness of the transmitted information. In a recent trial at the Emory University School of Medicine, deficits in handovers related to variability of transferred information was observed, and it was shown that 51% of resident doctors are not confident about patient handovers using traditional hospital methods (Payne et al., 2012). Handover related errors may lead to patient deaths (Kohn et al., 2000), therefore it is important that the underlying issues are addressed systematically and generically.

The number of handovers in the everyday healthcare workflow is increasingly high due to the high level of medical specialization of staff as well as the necessity to maintain continuity in teams of people working in shifts of decreasing duration (UK Department of Health, 2009). The general lack of implemented guidelines leads medical staff to employ informal, mostly oral-based collaboration procedures that are prone to errors, omissions, and miscommunications. Such procedures can be further complicated by the workload, physical and psychological state of the involved staff, and even by social
conventions. Taking into consideration the complexity of such collaborations, especially when multiple team members are involved or when under intense operational pressures due to the treatment of critically ill patients, the increased risk to patients’ safety becomes apparent.

An important underlying issue is the incomplete or ambiguous specifications of responsibility and accountability in collaborative work of healthcare teams. This is particularly amplified by the involvement of healthcare assistants (HCAs) and assistant practitioners (APs) [Mackey and Nancarrow, 2005; Wakefield et al., 2009]. APs were introduced relatively recently in the National Health Service (NHS) in the UK in an effort to address the increased patient to medical staff ratio by relieving the specialized staff from tasks that do not require high degrees of training. However, the exact role for both HCAs and APs is not clarified in the existing NHS policies, and their use varies greatly [Spilsbury and Meyer, 2004]. Such underspecified and dynamic roles in conjunction with the lack of implemented, structured guidelines as described above, cause more work to be assigned or delegated through chains of multiple handovers, e.g. from a doctor to a specialist to a nurse to an HCA. The propagation of responsibility and accountability in such situations is unclear and may lead to problematic situations, especially in exceptional cases involving unexpected events or obstacles, with direct implications for patient safety.

In this part of our work, we used our logic-based process modelling approach to create rigorous specifications of two patterns for collaborative work in healthcare teams. In this context, abstract processes are used to specify primitively specified tasks or goals, also referred to as keystones, which describe the individual steps that need to be performed or satisfied to achieve a collaboration. Moreover, using the expressiveness and particular properties of CLL, it is possible to enforce a variety of constraints, including the correct allocation of responsibility and accountability among the involved medical staff, based on particular hospital policies and guidelines.

Using the introduced diagrammatically-driven, theorem-proving based framework, we can compose said keystones to construct formally verified workflows which correspond to patterns of collaboration. Based on the properties of the proofs-as-processes background, our framework guarantees the correctness of the information flow between the keystones that make up the healthcare patterns, and also provides an executable model of the corresponding collaboration scenario in Scala code.
Based on this work, we envision a broader framework that provides a structured model of collaboration of medical teams which directly enforces the necessary policies for patient safety. The formal, logical background allows the mechanical verification of the correctness of the implementation of the policies, whereas a diagrammatic interface and an automated extraction of executable models facilitate the adjustment of the system to constant policy changes as well as enhancing the usability of the implemented framework. This is part of ongoing work that goes beyond the results of the current PhD project.

As mentioned previously, our work focuses on two generic patterns of collaborative work in healthcare, assignment and delegation. We describe these using a simple example in the next section.

### 10.3 Collaborative Healthcare Patterns

We focus our investigation on basic collaboration scenarios in healthcare that involve two agents, also known as actors. These may correspond to any member of the medical staff, including all doctors and nurses. We are particularly interested in two patterns or skeletal plans that differentiate between the types of collaboration through assignment and delegation of clinical services. In these, one of the two actors, the requester, asks for a particular clinical service (e.g. specialized diagnosis or treatment, administration of a drug, etc.) from the other, the provider. In order for a contract to be signed between the requester and the provider, it must be ensured that the provider is competent to perform the service. Moreover, depending on whether the service is assigned or delegated, responsibility and accountability is either transferred to the provider (assignment) or maintained by the requester (delegation). We proceed to explain the approach in more detail in the next few sections, with the help of a sample scenario.

#### 10.3.1 A Realistic Scenario

Consider the situation (Grando et al., 2011) where a general practitioner (GP) diagnoses a patient with acute renal failure (ARF) and, lacking the expertise to treat the ARF, he (requester) decides to assign this task to a nephrologist (provider), Dr. John, who then settles on a haemodialysis treatment and delegates this task (as a requester
this time) to an advanced practice nurse (APN), Anna (provider). The transfer of responsibility and accountability between the medical staff in this example should be noted. In the case of the assignment, the GP transfers responsibility and accountability for the service enactment and handling of exceptions (see Section 10.4.3) to Dr. John, whereas in the case of the delegation, Dr. John maintains responsibility for handling any exceptions that may occur when the APN cannot handle them, as well as for checking the outcome of the delegated service.

10.3.2 Keystones as Processes

Clinical services are broken down to a series of individual tasks and goals, i.e. the keystones, that each of the actors must perform in order to complete the service. These include tasks that must be performed in exceptional cases and unexpected situations. Each keystone has preconditions that must be met and information that must be given as input for it to be achieved, and success conditions that describe the achieved effects and output information of a successful completion. For example, in order to achieve the goal of awarding a contract to a provider (ContractAwarded), there needs to exist a provider (HealthcareServiceProvider) and an unfulfilled contract that the latter has accepted (AcceptedHealthcareContract). The result of this goal is a signed, open contract (OpenHealthcareContract) between the requester and the provider (see Figure 10.2).

From the process-based point of view of our framework, keystones are represented as abstract processes. The preconditions and success conditions of each keystone can be realized as preconditions and effects of a process, while the information required for the keystone and that produced either by its successful achievement or by some exceptional outcome can be represented as inputs, outputs, and exceptions of the same process. Therefore, the information flow through the processes is also made explicit. We note that, based on this association, the terms keystone and process will be used interchangeably in the rest of this chapter.

Using the notation introduced in Section 6.4, Figures 10.1, 10.2, and 10.3 give the diagrammatic representation of the processes used to model the keystones involved in the assignment and delegation patterns in our framework, whereas Figure 10.4 includes two simple automated procedures that are also involved in the same patterns. The CLL specification of each process as well as a brief textual description are provided.
10.3. Collaborative Healthcare Patterns

ServiceASSGRequest

⊢ HealthcareService⊥, HealthcareServiceRequester⊥, Patient⊥,
Assignment ⊗ RequestedHealthcareContract ⊗ HealthcareServiceRequester⊗
PendingHealthcareService

The ServiceASSGRequest keystone corresponds to the task of requesting the assignment of the service to another actor. Its input comprises of information on the involved patient (Patient), the clinical service being requested (HealthcareService), as well as the information of the actor making the request (HealthcareActor). Its outputs include information on the pending service (PendingHealthcareService), the request (Assignment), the requester (HealthcareServiceRequester), and the requested contract (RequestedHealthcareContract).

ServiceDELGRequest

⊢ HealthcareService⊥, HealthcareServiceRequester⊥, Patient⊥,
Delegation ⊗ RequestedHealthcareContract ⊗ HealthcareServiceRequester⊗
PendingHealthcareService

This keystone only differs from ServiceASSGRequest in the fact that it generates a request for delegation (Delegation) rather than assignment.

Figure 10.1: Keystones represented as processes (1 of 3).
Chapter 10. Formal Verification of Collaboration Patterns in Healthcare

**CollabDecision**

\[ \vdash \text{RequestedHealthcareContract} \downarrow, \]
\[ (\text{AcceptedHealthcareContract} \odot \text{HealthcareServiceProvider}) \oplus \]
\[ \text{RejectedHealthcareContract} \]

The CollabDecision keystone corresponds to the decision of whether, given a requested contract (RequestedHealthcareContract), there is a competent actor that can provide the corresponding service. If a collaboration is decided, then the accepted contract (AcceptedHealthcareContract) as well as the information of the provider (HealthcareServiceProvider) are returned. Otherwise, a rejected contract is generated (RejectedHealthcareContract).

**ContractAwarded**

\[ \vdash \text{HealthcareServiceProvider} \downarrow, \text{AcceptedHealthcareContract} \downarrow, \]
\[ \text{OpenHealthcareContract} \odot \text{HealthcareServiceProvider} \]

Once a contract has been accepted (AcceptedHealthcareContract) by a competent provider (HealthcareServiceProvider), the requester needs to finalize the agreement and award the contract to the provider. An open contract (OpenHealthcareContract) that needs to be fulfilled is generated, in conjunction with the provider’s information (HealthcareServiceProvider).

Figure 10.2: Keystones represented as processes (2 of 3).
10.3. Collaborative Healthcare Patterns

This keystone corresponds to the task of executing a clinical service that has been requested and is currently pending (PendingHealthcareService). An associated open contract (OpenHealthcareContract) must be available for this service. If the service is completed successfully, it is marked as “completed” (CompletedHealthcareService) and is returned coupled with its open contract (OpenHealthcareContract) information pending the final check. However, it is possible that an obstacle occurs (OBSTACLE) that halts the execution of the service. The open contract (OpenHealthcareContract) and the still pending service (PendingHealthcareService) are returned in this case (see Section 10.4.3.1 for a more detailed explanation of obstacles).

This keystone corresponds to the goal of checking the outcome of the provided service (CompletedHealthcareService) by the responsible actor (HealthcareActor) based on an existing contract (OpenHealthcareContract). For simplicity we assume this goal is always successful. The output includes the information on the now completed and checked service (CheckedHealthcareService), as well as the closed contract (ClosedHealthcareContract).

Figure 10.3: Keystones represented as processes (3 of 3).
Chapter 10. Formal Verification of Collaboration Patterns in Healthcare

10.3.3 Collaboration Patterns as Process Compositions

Keystones are combined to form plans or patterns. Each pattern corresponds to a workflow of tasks/goals that must be performed in a collaboration scenario, thus clearly defining the series of steps the actors involved must follow. Workflow patterns for the appropriate handling of exceptional situations are also introduced (see Section 10.4.3).

Using our process composition framework, it is possible to create healthcare collaboration patterns that implement different policies, merely using an intuitive diagrammatic interface. The formal background of proofs-as-processes in our approach provides guarantees that the patterns are correct with respect to both the specified information flow of the individual keystones and their preconditions and success conditions (effects). Completed patterns can be stored and reused, or even combined further to create more complicated workflows (see Section 10.4.3.2 for an example). It is also possible to simulate the various scenarios of the constructed patterns as described in Section 10.4.2.

As noted previously, we focus on the two particular collaboration patterns of assignment and delegation, which we describe next.

![Diagram of AssgResponsible and DelegResponsible keystones](image-url)
10.3.3.1 Assignment

As demonstrated by our example scenario in Section 10.3.1, an assignment is a type of collaboration where a clinical service is assigned from a requester to a provider. Responsibility and accountability is also transferred to the provider. This means that the provider is responsible to check the outcome of the service and handle any exceptional events that may occur.

The diagrammatic representation of the assignment pattern as a process composition is shown in Figure 10.5. In it, the workflow of keystones is clearly shown giving emphasis to the information being exchanged between the two actors in each step. First, a request for assignment is made through ServiceASSGRequest, then the decision to collaborate is made by a competent provider through CollabDecision, followed by the signing of a contract in ContractAwarded. The actual clinical service is provided in the ServiceProvide keystone, while the automated process AssgResponsible determines the responsible actor to check the outcome (see Section 10.4.1). This check is performed in the OutcomeCheck keystone.

It is important to note that, based on the resource-oriented properties of CLL, all the possible scenarios are taken into consideration. In particular, we see that there are three potential outcomes for the assignment pattern:

1. In the first scenario, everything has gone through, and the requested service has been performed successfully. Therefore, the output includes the two products of a successful result, i.e. a completed service whose outcome has been checked (CheckedHealthcareService) and the corresponding closed contract (ClosedHealthcareContract).

2. In the second scenario, an exceptional situation occurred that the service provider was unable to handle, also referred to as an obstacle (see Section 10.4.3.1 for an example). The output in this case includes the information regarding the obstacle that occurred (OBSTACLE), the information about the service provider (HealthcareServiceProvider), the service assignment request which is still unfulfilled (Assignment), the open contract (OpenHealthcareContract) that is not yet fulfilled, and the pending service (PendingHealthcareService). All this information can be used in order to determine how to handle the obstacle, as well as how to continue with the treatment in the case where the obstacle is overcome.
3. In the third scenario, no competent providers that are currently capable and available to undertake the requested service could be found, and therefore the contract is rejected (RejectedHealthcareContract). The unfulfilled service is still pending (PendingHealthcareService) and the service assignment request (Assignment) is still open.

The assignment pattern can be collapsed and viewed as a single composite keystone as shown in Figure 10.7a. In this form it can be re-used or further composed to create more complicated patterns.

### 10.3.3.2 Delegation

A delegation differs from an assignment in the way responsibility and accountability remain with the service requester. This means that the original requester of the clinical service has to check the outcome of the service as well as handle any exceptional events that may occur if the provider is unable to deal with them.

The diagrammatic representation of the delegation pattern is shown in Figure 10.6. It can also be collapsed and viewed as a single keystone as shown in Figure 10.7b. Note that the delegation pattern also has three potential outcomes, i.e. a successful result, an exceptional event, or a rejected contract.

### 10.4 The Application of our Framework

In this section, we examine in more detail some specific aspects of the constructed collaboration patterns in order to demonstrate how some interesting and important properties emerge from the application of our framework. More specifically, we focus on the mechanism used to ensure that specific policies are enforced through the constructed workflows in Section 10.4.1. In Section 10.4.2, we discuss the benefits of running simulations using the PiVizTool, whereas in Section 10.4.3 we analyse how exceptions are handled explicitly and systematically by our system thanks to the underlying logic. Finally, in Section 10.4.4 we introduce an ontology that provides richer specifications of the types involved in the keystone specifications.
10.4. The Application of our Framework

Figure 10.5: Diagram of the assignment pattern as a verified process composition. Note that the pattern has been split into two parts at the indicated point for it to fit within the page.
Figure 10.6: Diagram of the delegation pattern as a verified process composition. Note that the pattern has been split into two parts at the indicated point for it to fit within the page.
10.4. The Application of our Framework

(a) Assignment pattern.

(b) Delegation pattern.

Figure 10.7: Diagrams of the assignment and delegation patterns, each as a single composite keystone.
10.4.1 Enforcing Policies

Our composition model focuses on the types of information being exchanged as well as the types of the preconditions and effects for each keystone. As mentioned in Section 6.1, these types can be mapped to concrete classifications, e.g. relational database tables or ontological classes, both of which are often used in modern medical frameworks such as Tallis (Tallis 2011). However, this provides a relatively abstract representation of IOPEs, where individual parameters of each resource are not specified explicitly. For example, we abstract from the parties involved in an open contract OpenHealthcareContract.

This abstraction may appear limiting with regards to expressing restrictions related to hospital policies. For example, the actor of the OutcomeCheck process must be different depending on whether the service has been assigned (where the actor doing the check is the provider) or delegated (where the actor performing the check is the requester). The model can only be useful and realistic if we can express and verify such conditions in the produced patterns so that they reflect the actual hospital policies. Using specific techniques, it is possible to specify such conditions on various parameters explicitly in our framework.

In OutcomeCheck, in order to control the actor that will achieve this keystone, we have specified it as the explicit input HealthcareActor. We then introduce two automated processes AssgResponsible, and DelegResponsible, that help determine the responsible actor for OutcomeCheck based on the context. In particular, if an Assignment has been requested, then AssgResponsible determines that the provider (HealthcareServiceProvider) will be responsible to perform the OutcomeCheck keystone, whereas if a Delegation has been requested then DelegResponsible determines that the requester (HealthcareServiceRequester) will be the responsible actor.

To summarize, by controlling the level of abstraction of the specified information for each keystone, we can enforce additional constraints on particular parameters. This allows us to express and verify specific conditions that must be enforced by hospital policies and guidelines. In our particular example, the transfer of responsibility and accountability for the outcome of the clinical service is made explicit and is automatically verified by our underlying logic-based framework.
10.4.2 Simulation

Our process composition framework produces an executable, $\pi$-calculus model of the composition. This means we can obtain an executable specification of our constructed collaboration patterns and, thus, run simulations of collaboration scenarios.

Using our modified version of the PiVizTool (see Section 3.3.2), we can simulate scenarios of the constructed patterns. For these simulations, we can specify the initial state and input information to our patterns. PiVizTool then visualizes a step-by-step simulated execution of the process, showing the connections between the various keystones and the information being exchanged between them. It is also possible to investigate alternative outcomes, such as cases where exceptions are thrown. This allows us to empirically analyze and verify the modelled collaborations and the enforcement of the corresponding hospital policies.

For example, when running a simulation of the DelegationPattern, we can observe that the information of the healthcare service requester is passed to the OutcomeCheck keystone as the HealthcareActor parameter that determines who is responsible for this keystone, thus empirically verifying the correct application of our policy (as described in the previous section).

A snapshot of the simulation of the AssignmentPattern using PiVizTool is shown in Figure 10.8.

As discussed in Chapter 8, the concrete specification of the patterns constructed by our system can be translated into actual executable Scala code, leading to a deployed, fully usable information system. We describe such a deployment in Section 10.5.

10.4.3 Exception Handling

One of the major advantages of our approach, especially when compared to other process composition approaches, is the guarantee that all resources will be handled explicitly. This includes exceptions, i.e. situations where something unexpected happened instead of the normal course of events. Information about exceptions is guaranteed to be passed on to the user as the final output of our composed patterns, or to be handled and resolved internally. Exceptions are typically accompanied by unused resources, or resources necessary for their appropriate handling. In our assignment and delega-
Chapter 10. Formal Verification of Collaboration Patterns in Healthcare

Figure 10.8: An intermediate snapshot of the simulation of the AssignmentPattern in PiVizTool. Each node corresponds to a keystone, each dark edge to an available execution step, and each grey edge to a blocked execution step. The Response node represents the user who receives the final result.
10.4. The Application of our Framework

In our example of the ARF patient and the haemodialysis treatment, there is a possibility that air bubbles appear in the circuit, causing an immediate threat of air embolism. In such a situation, it is not possible to continue the treatment until the patient is stabilized. An obstacle-type exception is then thrown, and the responsible actor (in this case Dr. John) must be informed of it and take action to resolve it. In particular, they need to perform a set of tasks to resolve this particular obstacle, i.e. they need to draw out the blood and remove the air bubbles, then change the position of the patient, then prescribe oxygen, and finally assess the condition of the patient until they are stable. Once these tasks are completed and the obstacle is removed, the execution of the original clinical service, i.e. the haemodialysis treatment, may resume. However, it is also (naturally) possible that the obstacle may lead to the cancellation of the original treatment if the patient is no longer suitable for it.

In our attempt to model these situations in a general, abstract fashion, we focus on the importance of making the information surrounding the exception explicitly available. The ServiceProvide, therefore, has two optional outputs: either the completed healthcare service (CompletedHealthcareService) or the obstacle that occurred (OBSTACLE) accompanied by the information on the service that is still pending (PendingHealthcareService). The latter information may be used to provide the context in which the exception occurred, as well as the possibility to resume the original clinical service. Both outputs are accompanied by the existing contract between the two actors (OpenHealthcareContract) that is still open.

In Section 10.3.3 and in Figures 10.5 and 10.6, we described the outputs of the complete assignment and delegation patterns. For each pattern, there was a possible out-
come of an obstacle occurring as an exceptional scenario in ServiceProvide. The corresponding output included the obstacle that occurred (OBSTACLE), the pending service (PendingHealthcareService) and open contract (OpenHealthcareContract), as well as additional information that was available in the context of the particular collaboration, including, for example, the service request information (Assignment and Delegation in each case respectively).

It is, thus, made clear that any exception that may occur during the execution of clinical service, as well as all its contextual information, will be properly forwarded as the output of the entire collaboration pattern. Our framework provides guarantees that no information will be lost, and that such exceptions will always be forwarded to the runtime user. The user can then decide on the appropriate course of action to resolve the obstacle and decide, based on the available information, if they should resume the original clinical service or cancel it and close the corresponding contract.

10.4.3.2 Rejected Request

The second type of exception that we have modelled using our framework is the case where no competent medical staff is available to accept a contract for the requested clinical service. This may be caused by several reasons, including the only hospital specialist for the particular service being busy or absent, or the lack or unavailability of specific equipment required to perform the service in the hospital. If we assume the latter reason, it is often the case that the only solution is to transfer the patient to another hospital.

In our model, the CollabDecision keystone has two possible outcomes (see Figure 10.1): either a competent actor accepts the request, which leads to an accepted contract (AcceptedHealthcareContract) accompanied by the actor’s information (HealthcareServiceProvider), or the request is rejected leading to a rejected contract (RejectedHealthcareContract).

In Section 10.3.3, we saw that both the assignment and delegation patterns had a possible outcome that corresponds to the case of the rejected contract. Similarly to the obstacle-type exception of the previous section, the rejected contract is forwarded to the final output for each of the two patterns, accompanied by contextual information such as information on the requested healthcare service.
10.4. The Application of our Framework

It is possible to construct an extension to the AssignmentPattern so that all rejected service requests are transferred to another hospital. We assume a pre-existing composition of all the necessary keystones for the hospital transfer where the hospital policies and guidelines are enforced, to form the InterhospitalTransferPattern pattern. This pattern has a rejected contract (RejectedHealthcareContract) as an input (and thus is only triggered when a contract has been rejected, i.e. it is already determined that the hospital is not able to handle the patient), as well as the information on the service being requested (PendingHealthcareService), the person who made the request (HealthcareServiceRequester), and the requested assignment (Assignment). The output is a state where the requested clinical service is considered completed and checked as it is no longer the responsibility of the hospital (CheckedHealthcareService), and a closed contract exists between the requester and the receiving hospital (ClosedHealthcareContract). The diagrammatic representation of the InterhospitalTransferPattern as a single keystone is shown in Figure 10.9.

![Figure 10.9: Diagram of the interhospital transfer pattern as a single composite keystone.](image)

The AssignmentPattern and InterhospitalTransferPattern can now be composed to create the extended ExtAssignmentPattern pattern where all rejected contracts are handled internally. The resulting composition is shown in Figure 10.10. Note that, in this diagram, only the resources involved in the case of the rejected contract are directly passed to InterhospitalTransferPattern, as denoted by the dark edges, whereas the resources for the other two cases are merely buffered through, as denoted by the grey edges. The ExtAssignmentPattern pattern can also be collapsed into a single keystone as shown in Figure 10.11. Notice that the outputs of the ExtAssignmentPattern are similar to the outputs of the AssignmentPattern, with the case of the rejected contract replaced by another case of a successful result.

We have, thus, demonstrated the capabilities of our system to accommodate patterns that handle specific exceptions internally instead of forwarding them to the user.
Chapter 10. Formal Verification of Collaboration Patterns in Healthcare

Figure 10.10: Diagram of the ExtAssignmentPattern pattern as a composition of AssignmentPattern and InterhospitalTransferPattern.
10.4.4 Ontological Classes as Types

In Section 6.5, we gave a brief description of ontologies. Ontological concepts can be used to describe the IOPE types in the process specifications.

Ontologies have seen increasing usage in the context of healthcare, biomedicine, and biology. For instance, the Open Biological and Biomedical Ontologies (OBO) Foundry is a collaborative effort to establish principles of ontology development in the biomedical domain so that such ontologies can be shared across different medical and biological domains (Smith et al., 2007). In particular, they have developed the OBO language, based on the OWL2 semantics, for the development of biological and biomedical ontologies. Moreover, the National Center for Biomedical Ontology (NCBO) (Musen et al., 2012) aims to support biomedical researchers by providing ontology tools such as the BioPortal (Whetzel et al., 2011), an ontology repository containing over 300 biological and biomedical ontologies.

In the current work, we can use ontological classes to provide a concrete classification of the information being exchanged between healthcare related processes. We introduce a new OWL2 ontology that provides a number of named classes to describe a variety of concepts involved in healthcare collaborations, such as contracts, healthcare services, and the different roles of medical staff. We can use these classes to make the specifications of the constructed workflows richer.

Our ontology builds upon existing efforts to describe such concepts, and, in particular, imports the following two medical ontologies:

- The Ontology for General Medical Science (OGMS) (OGMS, 2012) is an OBO ontology that describes general concepts in (human) healthcare such as disease,
injury, disorder, patient, healthcare process, etc. It uses the Basic Formal Ontology (BFO) (Smith and Grenon, 2002) as an upper ontology and has been built based on the results of a number of related papers and workshops.

- The Ontology of Medically Related Social Entities (OMRSE) (OMRSE, 2012) is an extension of OGMS that describes healthcare agents and their social roles, including patients, their relatives, and medical staff.

as well as the following web service related one:

- The SOA Ontology (The Open Group, 2012) was created to describe and give a better understanding of the core concepts of Service Oriented Architectures (SOA) (MacKenzie et al., 2006). It is mainly addressed to business people, as well as software engineers, designers, and architects (see Section 10.6).

The combination of the medical ontologies with the SOA ontology allows us to define a number of concepts involved in healthcare collaborations from the SOA point of view (see Section 10.6 for more details), thus matching perfectly the particular needs of our approach. We named our defined ontology medicalsoa.owl and the hierarchy tree of the introduced ontological classes is shown in Figure 10.12.

![Figure 10.12: Class hierarchy tree for the medicalsoa.owl ontology.](attachment:image.png)
A number of constraints and relationships are explicitly defined in this ontology. For example, an Assignment and a Delegation are both types of service requests, and therefore subclasses of ServiceRequest, which, in turn, is a subclass of Thing. Moreover, Assignment and Delegation are disjoint classes, which means that a request can either be an assignment or a delegation request, but not both. The ontology also includes object properties to describe the relationships between various concepts. For example, the object property isContractFor describes the relation between a ServiceContract and the corresponding Service. Finally, we have introduced individuals for some of the classes. For example, APN Anna is an individual of the HealthcareActor class, whereas the hæmodialysis treatment is an individual of the HealthcareService class.

It is worth noting that our medicalsoa.owl ontology can be extended further to describe more specific concepts depending on the needs of a particular application, e.g. in other cases of healthcare collaboration that we have not considered in this chapter.

10.5 System Deployment: The DigiHealth Prototype

In Chapter 8 we analysed how the workflows constructed by our framework can be deployed as a software system automatically by translating their $\pi$-calculus specifications to executable Scala code. In order to demonstrate the functionality of such a deployment, we have developed a prototype, web-based system that emulates part of a hospital environment. We will simply refer to this system as the DigiHealth (Digital Healthcare) prototype system.

DigiHealth is designed using a standard Model-View-Controller (MVC) architecture (Burbeck 2009). This type of architecture is commonly used especially in modern web-based applications. It maximizes code reusability and separation of concerns by identifying 3 individual but connected components:

- The Model corresponds to the programmatic structures that map to the underlying data. Typically and in our case, the data is stored in a relational database and accessed programatically through the Model component.

- The View component realises the interface through which the data become available to the user. In the context of a web application, the View most commonly consists of dynamically generated HTML pages.
The Controller implements the ways that the data can be handled both by the user and internally by the system. These are often implemented as available user commands or as an API published on the web.

The full architecture of our DigiHealth implementation is shown in Figure 10.13. It consists of a Model connected to a relational database, a View comprised of dynamic HTML pages that can be displayed on any browser and device, including mobile devices such as tablets and mobile phones, and a Controller which reveals a public API. The DigiHealth MVC server was implemented in PHP (Achour et al., 2013).

Once the healthcare collaboration patterns are constructed in our proofs-as-processes framework, the Scala deployment directly implements the corresponding workflow. We then implemented each keystone/process as an interaction with DigiHealth’s API. In this way, the order in which the API calls are made and the involved information exchanges are dictated by the formally verified workflow as opposed to some hardcoded solution. As a result, the implemented workflow gains all the properties provided by our logic-based framework, including correctness, rigour, accounting of information, maintainability, deadlock freedom, etc.
Before proceeding to describe the DigiHealth system in more detail, it is worth noting that it is only intended as a prototype for the purpose of demonstrating the formally verified workflows constructed within our framework can be applied on an actual software system. We expect the complexity of an actual healthcare software package to be much higher, with more complicated models, functionalities, and a more advanced user interface. However, we have followed the basic principles and typical design architecture of such large systems so as to provide a sufficiently realistic demonstration of how a workflow would be deployed within such an environment.

In what follows, we describe the different components of DigiHealth in more detail. More specifically, the Model is described in Section [10.5.1] and the Controller in Section [10.5.2]. We describe the implemented connection between the Scala workflow and the DigiHealth API in Section [10.5.3], whereas the View component is demonstrated in Section [10.5.4].

### 10.5.1 Model: A Relational Database

DigiHealth’s Model component was built on a SQLite relational database (Owens, 2006). The database design follows a standard Entity-Relationship (ER) model and is shown in Figure 10.14.

More specifically, the following tables are introduced:

- **The staff table** corresponds to the various actors/members of staff, including doctors, nurses, porters, etc. In our simplified system we store their staff ID, title, name, surname, and date of birth (dob).

- **The services table** corresponds to medical services, including, for example, haemodialysis treatment. Columns include a name and a short description of each service.

- **The patients table** is used to store information about the patients, including their patient ID, name, surname, and date of birth (dob).

- **The providers table** stores the relationship between services and the staff members that have the required competency to provide them. The level of competency for each provider in each service is also stored in this table.

- **The obstacles table** is used to describe possible obstacles that may occur dur-
Figure 10.14: Diagram of the ER model of the database for the DigiHealth system.

Figure 10.14: Diagram of the ER model of the database for the DigiHealth system.

...ing the provision of a service (see Section 10.4.3.1). Information stored in the columns includes a name and a short description (possibly including ways of dealing with each obstacle).

- Entries in the states table correspond to the possible state of a contract, a requested service, or an obstacle alert (see below). More specifically, the states table includes the following rows:

  1. requested
  2. accepted
  3. rejected *
  4. open/pending
  5. closed/completed
  6. checked/resolved *

Note that the final column is used to describe whether or not the state is a final state, i.e. whether no further action is expected to be taken for the corresponding entity. States marked with a * in the above list are final.

- The requestedservices table stores information about service requests, in-
cluding the actor who made the request, the involved service and patient, the actor responsible to check the outcome of the service, the date, the type of the request (Assignment or Delegation), its state, and any further notes provided by the requester.

- The **contracts** table stores information about contracts between actors over service requests. Each entry contains a key to the corresponding service request, the provider, the state of the contract, and the timestamps when the contract was requested, opened, and closed for documentation purposes.

- The **obstaclealerts** table stores alerts of obstacles that have occurred during the provision of a requested service. Entries include information on the requested service, the obstacle that occurred, the actor who issued the alert, the timestamp of issue, the state of the alert, and any further notes included by the issuing actor.

Each of the CLL types used in the specifications of the keystones (see Section 10.3.2) has some corresponding realisation in the database. This could be either a table, a join of more than one tables, or a view of these, i.e. a selection of entries with particular properties. For example:

- A **HealthcareActor** is an entry in the **staff** table.

- The **ServiceRequest** type is a join of tables **requestedservices**, **staff**, **services**, **patients**, and **states**.

- The **Assignment** type is a view of the same join as for the **ServiceRequest** type where the **type** column of **requestedservices** is set to “Assignment”.

### 10.5.2 Controller: The PHP API

The Controller component of the DigiHealth system implements and exposes a simplified, web-based (HTTP GET) API that can be used to trigger any of the specified keystones. Each API call has arguments that correspond to the database IDs of the keystone’s inputs. The call then changes the information stored in the database appropriately and stores the database IDs of the keystone outputs on a predetermined file. These files are used to communicate the results back to the workflow (see Section 10.5.3).

For keystones that are implemented as human processes, the API provides different
calls for the different possible outcomes of the keystone. The inputs of the keystone are presented to the user as part of the View component (see Section 10.5.4). In a sense, such keystones are implemented as a combination of parts from both the View and the Controller components.

A list of available API calls, the corresponding keystone, and a short description are shown in Table 10.1.

For example, assuming the API is deployed in a local machine (localhost) under the digihealth domain, a new Assignment can be initiated by the actor with ID 2, for service with ID 5, on patient with ID 6 using the following HTTP request:

```
http://localhost/digihealth/api.php?action=assg
&requesterid=2&serviceid=5&patientid=6
```

Note that naturally the API is meant to be accessed programmatically and not manually by a human. Humans can access the API calls through the View which provides a better textual visualisation of the data (see Section 10.5.4).

### 10.5.3 Workflow Execution: The Scala Implementation

The Model and Controller components we described so far follow a simplified version of the Service Oriented Architecture (SOA) of standard, modern, web-based applications. Typically at this stage, in order to control the flow of API calls in a user session involving an assignment or delegation, developers would introduce hardcoded workflows. Apart from the lack of maintainability, these workflows would also suffer from possible inconsistencies and lack of verification of the information flow (or limited verification based on the skills of the developer). The difficulty in constructing a system with a consistent workflow is compounded by the fact that multiple users are interacting during a single run, whereas multiple runs of the same workflow but different parameters could be running simultaneously (multiple assignments and delegations with different actors, services, and patients). Keeping track of the information, validating the types of inputs and outputs in each API call, and making sure all possible outcomes are handled as expected are some of the desirable properties that would require considerable effort to ensure in such a hardcoded system.

Our aim is to use our composition framework to construct an automated, formally verified, executable workflow so that the developer is relieved from the complexities
### Table 10.1: List of available API calls in the DigiHealth system and the corresponding keystones they implement.

<table>
<thead>
<tr>
<th>API call</th>
<th>Keystone</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>assg</td>
<td>ServiceASSGRequest</td>
<td>Initiates a new service request of type “Assignment”.</td>
</tr>
<tr>
<td>deleg</td>
<td>ServiceDELGRequest</td>
<td>Initiates a new service request of type “Delegation”.</td>
</tr>
<tr>
<td>collab</td>
<td>CollabDecision</td>
<td>Initiates the collaboration decision process.</td>
</tr>
<tr>
<td>accept</td>
<td>CollabDecision</td>
<td>Is triggered when a provider accepts a requested contract.</td>
</tr>
<tr>
<td>reject</td>
<td>CollabDecision</td>
<td>Is triggered when a requested contract is rejected (due to no available competent providers or timeout).</td>
</tr>
<tr>
<td>awardinit</td>
<td>ContractAwarded</td>
<td>Initiates the process of awarding an accepted contract.</td>
</tr>
<tr>
<td>award</td>
<td>ContractAwarded</td>
<td>Awards the contract to a provider who accepted it.</td>
</tr>
<tr>
<td>provide</td>
<td>ServiceProvide</td>
<td>Initiates the process of service provision.</td>
</tr>
<tr>
<td>done</td>
<td>ServiceProvide</td>
<td>Is triggered when a service is completed successfully by the provider.</td>
</tr>
<tr>
<td>obstacle</td>
<td>ServiceProvide</td>
<td>Is triggered when an obstacle occurs during the provision of a service. An obstacle alert is issued.</td>
</tr>
<tr>
<td>checkinit</td>
<td>OutcomeCheck</td>
<td>Initiates the process of checking the outcome by the responsible actor.</td>
</tr>
<tr>
<td>check</td>
<td>OutcomeCheck</td>
<td>Is triggered when the outcome of a provided service is checked.</td>
</tr>
<tr>
<td>assgresp</td>
<td>AssgResponsible</td>
<td>This call arranges the responsible actor in the case of an assignment.</td>
</tr>
<tr>
<td>delegresp</td>
<td>DelegResponsible</td>
<td>This call arranges the responsible actor in the case of an assignment.</td>
</tr>
</tbody>
</table>
Chapter 10. Formal Verification of Collaboration Patterns in Healthcare

described above. The Scala deployment based on what we described in Chapter 8 of the AssignmentPattern and DelegationPattern compositions provides us with exactly that. Essentially, this corresponds to a Scala workflow coordinator that controls the order of API calls being made and the flow of information between them. In order for this coordinator to interact with the current API, a few simple development steps were required. We describe these next.

The first step was to introduce concrete Scala datatypes for the CLL types of the key-stone specifications. In order to facilitate the communication between Scala and PHP, all the types were implemented as integers, so that an instance of a CLL type is the ID (value of the primary key) in the corresponding database table (see Section 10.5.1). It is worth mentioning that there are modern serialisation languages (such as JSON, XML, etc.) and methodologies to facilitate this kind of communication, but we decided to use this simpler technique in the context of our effort towards a rapid prototype.

Secondly, in order to execute the deployed workflow, concrete instantiations of the keystones were required. This was achieved by constructing a communication bridge between the PHP API and the Scala classes as follows:

- Scala objects can trigger API calls easily using simple HTTP GET requests. This was implemented using the Java class InputStreamReader.
- PHP functions can send messages to the Scala workflow through files. Each key-stone uses a pre-determined file to store the outputs in the form of database IDs. Given the type safety of the implemented workflow and verifying that the API calls record the correct type of information based on their specification allows us to minimize then need for data validation. We use the Java library JNotify [Yadan, 2013] to allow the Scala processes to block until a file containing an output message is created by the corresponding PHP function.

Once again, there are modern messaging technologies with advanced functionality that can allow much better communication between loosely coupled applications. We chose this file-based messaging approach in order to speed up the production of our prototype implementation.

As an example, the implementation of the AssgResponsible keystone in Java, namely the DGHAsgResponsibleAction, is shown in Figure 10.15. Notice the use of the DGHAPlCaller class, which implements a simple API call to the DigiHealth API from the list shown in Table 10.1 (in this case a assgresp call), and the DGHAPlCall class,
which combines the execution of a DGHAPICaller instance with the retrieval of a response message through the given file (in this case a file named responsible followed by the ID of the assignment request). The execute method can be used to execute the keystone. In this, the constructed DGHAPICall is executed (via its call method) and the response (in this case the ID of the responsible actor) is parsed and retrieved with the help of Java’s Scanner class.

```java
public DGHAssgResponsibleAction(int requestid, 
            int providerid) {
    DGHAPICaller caller = new DGHAPICaller("assgresp");
    caller.addParameter("requestid", requestid);
    caller.addParameter("providerid", providerid);
    this.call = new DGHAPICall(caller,
        "responsible" + requestid);
}

public void execute() {
    try {
        Scanner scanner = call.call();
        String response = scanner.next();
        if (response.equalsIgnoreCase("R")) {
            this.actorid = scanner.nextInt();
        }
        scanner.close();
        call.cleanup();
    } catch (JNotifyException e) { e.printStackTrace(); }
}
```

Figure 10.15: Constructor and execution method for the implementation of the keystone AssgResponsible. It involves an API call and a message retrieval in order to interact with the PHP server.

A call to the DGHAssgResponsibleAction class can easily be embedded within the Scala workflow thanks to Scala’s backwards compatibility with Java. This is accomplished by completing the instance template that our system automatically generates (see Section 8.3.1) for the AssgResponsible Scala trait. We remind the reader that each abstract process in the Scala deployment of a workflow is described using an ab-
stream trait. Its instance must implement the `apply` method which corresponds to the actual functionality of the process. For the `AssgResponsible` keystone, we implement the `AssgResponsibleInstance` Scala class shown in Figure 10.16.

``` scala
class AssgResponsibleInstance extends AssgResponsible {
  override def apply( arg0 :Assignment, arg1 :HealthcareServiceProvider ) :HealthcareActor = {
    val (action :DGHAssgResponsibleAction) =
      new DGHAssgResponsibleAction(arg0,arg1)
    action.execute()
    action.getActorid()
  }
}
```

Figure 10.16: Embedding the implemented `AssgResponsible` keystone as a trait instantiation in the Scala workflow.

All processes are implemented in a similar way, thus accomplishing a direct interaction between the Scala deployed workflow and the PHP API. Running the deployed workflow ensures the correct flow of execution of the various API calls based on the model built by the user in our diagrammatic interface, including all the properties of correctness, efficiency through concurrency, rigorous accounting of information etc. which our logic-based framework guarantees.

### 10.5.4 View: A Web-based User-interface

The View component consists of dynamically built HTML pages. These present data to the user in a human-understandable way and provide access to the underlying API for every stage of the workflow. A simple login system allows each member of staff to login with their personal credentials so that they can view the information and available actions that are relevant to them. Note that the simplicity of the HTML interface makes it possible to use any modern mobile device to access the system.

A sample screenshot of the interface is shown in Figure 10.17. Some of the main features are the following:

- The menu on the left gives access to the user to review their history (previous contracts and requests that they were involved in) through the `History` link and
10.5. System Deployment: The DigiHealth Prototype

Figure 10.17: Sample screenshot from the prototype DigiHealth system.

to initiate a new assignment or delegation request through the Request link. The Root link allows the raw editing of the database for various purposes such as adding/removing new staff members, patients, services etc.

- The list of open contracts displays all the contracts on services that the user is the provider of. There are two available options for each contract. The completed link should be selected when a service has been completed successfully. If, however, an obstacle occurs, the user can select the obstacle type from the drop down list and click on the Alert! link to initiate the obstacle alert. Note that the two options correspond to the two possible outcomes of the ServiceProvide keystone.

- The pending checks list contains a list of services that have been provided and that the user is responsible to check the outcome of. Upon completion of the check, the check link should be selected. This is the implementation of the OutcomeCheck keystone.

- The list of accepted contracts displays the contracts that are requested by the user and have been accepted by the provider. The award link awards the contract to the provider based on the specification of the ContractAwarded keystone.
Finally, the list of available contracts consists of requested contracts about services that the user is capable of providing. The accept link can be used to accept the contract. This result corresponds to the successful outcome of the CollabDecision keystone.

Each of the available links triggers an API call with the appropriate parameters. The API call then executes the corresponding, implemented keystone and yields the appropriate results based on the chosen action. These results are then sent to the Scala workflow which dictates any subsequent steps by enabling the next link in the interface.

As an example, the open contracts list displays contracts that are have an Open state and their provider is the active user. The completed and Alert! options become available to the user only when the corresponding contract reaches the ServiceProvide keystone in the workflow. These two options correspond to the two possible outputs of the keystone, which is shown below as a reminder.

This kind of interface yields multiple benefits for the user, including the following:

1. They only view information that is relevant to them.

2. They have a selection of available actions that can be performed by simple clicks or screen taps (in the case of mobile devices) and simple web-based forms to fill in when information is required.

3. They are not required to inspect the complex structure of the underlying workflow, nor are they required to remember complicated guidelines, specific steps they need to follow, which people they need to contact, or what information they need to send. All of this is taken care for them automatically in the background by the Scala workflow. They merely fill in the requested information in the web-based forms and select the appropriate action.
10.6 Related Work

The current research is inspired by recent work by Grando et al. (Grando et al., 2011). In this, they proposed logic-based, pen-and-paper specifications of reusable patterns for i) specifying assignment and delegation of tasks and goals during collaborative work, ii) considering mechanisms for detecting abnormal events (exceptions), and iii) for transferring responsibility and accountability to the appropriate actor when recovering from exceptions. Using those patterns, Grando et al. sketched proofs that desirable properties, known as safety principles, related to accountability, responsibility, and competence could be ensured.

The discussed assignment and delegation patterns can be seen as combinations of the basic workflow resource patterns. The latter were originally developed by Russell et al. (Russell et al., 2005) in order to describe the allocation and use of resources (including machines and humans) in collaborative workflows. More specifically, they introduced specific resource allocation patterns that are workflow language-independent and can be instantiated according to particular scenarios. In our models, for example, the allocation of actor roles in assignment and delegation scenarios corresponds to the “role-based allocation pattern”.

Workflow-based approaches are often used in the general area of healthcare informatics to provide automated IT support for practitioners. Tallis (Tallis, 2011), for example, is one of the leading tools for the specification and enactment of clinical applications. It is based on the PROforma clinical workflow language (Fox et al., 1996), a formal knowledge representation language for clinical guidelines. Similar modelling principles are followed by other widely used tools (such as Asbru (Shahar et al., 1998), EON (Tu and Musen, 1999), GLIF (Peleg et al., 2000)) for the enactment of clinical applications (Peleg et al., 2003).

The main advantage of our approach compared to the above is the formal verification of the constructed patterns and the automated extraction of an executable model. The end-product is a system whose correctness is mechanically verified with associated guarantees regarding the enforcement of modelled conditions and policies, thereby allowing a high level of trust in the functionality of the implemented workflow.

Despite the commonalities, it is important to note that workflow-based systems such as Tallis are usually used in an effort to support decision making in diagnosis and treat-
ment. In our case, though, we are closer to a business process oriented approach that abstracts from these procedures and focuses on improving the operational support and automating some of the organizational aspects of patient care by, for instance, alleviating the need to communicate the same information to multiple people separately, automatically documenting repetitive information, and detaching social conventions from the actual healthcare workflow. A related approach towards the development of organisational workflows healthcare with the aim of minimizing related medical errors is presented by Malhotra et al. (Malhotra et al., 2007). Our approach has strong similarities with theirs since both involve task/process centered workflows at various levels of abstraction. However, their goal is to construct a cognitive model of the workflow in order to identify the most error-prone regions. In our case, the goal is to automate parts of the organisational procedures (such as communication and documentation), enforce hospital policies (for example by explicitly tracking responsibility and accountability), and provide a concrete instantiation of the workflow as a usable information system.

Another related effort is TESTMED (Cossu et al., 2012), an ongoing project aimed at the development of a system that will provide operational support in hospital wards through multimodal (with an emphasis on vocal) interfaces. The implemented system also follows a business process approach to model the enactment of clinical guidelines. Particular focus in this case has been given to creating a highly usable interface, even under exigent time constraints and in a noisy and busy environment (such as the emergency ward). TESTMED is a promising project that shares a lot of ideas with our adopted point of view. However, its limited formal underpinning does not provide any guarantees that the system is correct or that the clinical guidelines will be applied in the desired way. The claim that the system correctly routes the tasks to appropriate operators and services (Cossu et al., 2012) is difficult to verify, and ultimately relies mainly on trusting the skills of the involved programmers. Finally, TESTMED provides limited support for adjustments in the implemented guidelines. As hospital policies change over time, this would require modifications in the actual implementation of the system whereas, in our approach, the diagrammatic user interface facilitates the process of modelling and implementing new guidelines. Moreover, our ability to extract concrete executable models should minimize the effort required to update and re-deploy the system.

Finally, our process-based approach also closely relates to the Service Oriented Architecture (SOA) point of view (MacKenzie et al., 2006). The latter aims to organize and
utilise distributed resources under the control of different owners, mainly in (but not necessarily limited to) the context of business management. In it, the mechanism for satisfying a need using an available resource (or capability) is referred to as a service. SOA, therefore, is an organizational paradigm that aims to optimize reuse, growth, and interoperability of such services. We apply this paradigm in the context of clinical services, where our collaboration patterns aim to support the proper management of the medical staff capabilities when used to satisfy patient needs. The introduced medicalsoa.owl ontology (see Section 10.4.4) makes a clear mapping of our concepts to notions from the SOA paradigm (see Section 2.1.1.1), whereas our formal approach guarantees a verified and efficient environment for the interoperability of the involved components.

10.7 Ongoing and Future Work

Collaborations between medical teams in real situations are often dependent on many varying factors, including for example the particularities of each patient, and involve a large number of people from different departments of the hospital and with different specialties. On the one hand, this indicates the necessity for a structured model of these collaborations in order to maximize efficiency and minimize errors. On the other hand, creating a universal model that contains different patterns capturing all possible collaborations and situations is far from trivial. In light of this, we proceed to describe the main limitations of our approach.

The patterns of assignment and delegation presented in this article are relatively generalized and at a high level of abstraction. A handover may sometimes involve more than two members of staff, or, depending on the patient and the type of treatment, specialized information may be required. For example, in order to transfer a patient to surgery, a number of checks must be performed, including equipment checks and blood orders, and the related information must be available pre-transfer. Modelling more complicated collaboration scenarios in healthcare, such as intra-hospital patient transfers, by analyzing real-world cases and the corresponding medical records is part of ongoing work.

More specifically, in a recent 6-month project funded by the College of Science and Engineering, University of Edinburgh, and EPSRC, we used our framework to model
guidelines for intra-hospital transfers of patients requiring a tracheotomy. This involved a close collaboration with St Mary’s Hospital clinicians in London for requirement analysis and to obtain domain knowledge. This included meetings with clinicians, shadowing of the theatre coordinator, and interviews from various staff members. Due to the short duration of the project, the results were limited to the design and modelling phase and did not expand to the implementation phase. However, it demonstrated the suitability of our approach to deal with a complicated, real world problem, and provided sufficient grounds to compile an EPSRC grant proposal for a longer term project with the potential for wide-ranging impact beyond academia in terms of innovative healthcare processes. This work is described in a forthcoming paper.

10.8 Conclusion

In this chapter, we presented a rigorous approach for the verification of collaborative patterns in healthcare teams based on our formal process modelling and composition framework. We demonstrated how keystones in healthcare collaborations can be represented as processes that can be composed to create collaborative patterns that deal with the notions of assignment and delegation in clinical services.

Using CLL specifications, a variety of safety properties can be captured and enforced in our model, thus reducing the risk of error in such collaborative scenarios as expressed by said patterns. The proof-based framework provides guarantees of correctness for the composed patterns, thus ensuring a formally verified information flow. The constructed patterns can be extended so that exceptional events can be handled either internally or externally. The keystone specifications can be enriched with semantic information for the involved resources with the use of OWL2 and OBO ontologies.

Additionally, the constructed models of collaboration can be deployed as executable Scala workflows. The abstract nature of the composed processes allows for integration with any modern web-based or service oriented software system. The end result ensures the correctness of the information flow and the process-to-process communications based on the modelled policies and guidelines. As such, the system relieves the user from the need to memorize large policy documents, implement social or oral-based procedures, or interpret guidelines for best practices in the administration of clinical pathways. In addition, responsibility and accountability in various collabora-
tive procedures are tracked automatically based on the modelled policy.

Our work set the foundations for a general, extensible framework which, we believe, can help with the designing out of medical errors in healthcare systems by constructing and deploying rigorous models of clinical pathways using formal methods.
Chapter 11

Conclusion

In this thesis, we presented and analysed a rigorous approach to the development of correct-by-construction process workflows. In this last chapter, we summarise the various stages of development of this project and reflect on the key points, main contributions and limitations of our work.

11.1 On the Achievements and Limitations of Our Work

Our work began as an exploration of the means to exploit formal verification theories and tools in order to add value to the specification and implementation of modern, web based systems. The initial stages of our research led us to the work of Rao et al. and suggested the use of the proofs-as-processes paradigm for the formal verification of semantic web services composition.

Our use of a rigorous, logic-based approach within the proof assistant HOL Light quickly resulted in the identification of a number of inconsistencies in their original work. This led us to build a formal proofs-as-processes based service composition framework from scratch.

We were then able to apply this methodology to a variety of case-studies from different domains. From the analysis of these use cases emerged the realisation that the abstract web services models in our framework allowed for much greater flexibility and expressivity. For example, modelling human based processes, which form important keystones in many cases, is trivial. Even though our framework started as an explo-
ration of web services composition it evolved into a general approach to modelling of
process-based systems and a much larger variety of domains, including code synthesis
and business process modelling.

The application of our methodology to the modelling of collaboration patterns in
healthcare as well as subsequent healthcare-related projects demonstrate the generic,
versatile nature of our approach and its potential use in large-scale, real-world systems.

In what follows, we briefly recapitulate the key contributions and limitations for each
development stage of our research.

11.1.1 Formalisation of Proofs-as-processes

Recent developments in the theoretical research of the correspondence between lin-
ear logic and concurrent processes highlight the potential use of such theories for the
formal verification of process-based systems. Such an approach is timely given the in-
creased usage of distributed heterogeneous systems such as web services and machine
understandable data in Web 2.0 and the Semantic Web. Having a mechanization of
the proofs-as-processes theory, accompanied by a flexible proof system, is a big step
towards bridging the gap between theory and practice. It sets the basis for the develop-
ment of methodologies for formal verification of concurrent systems using linear logic
theorem proving and it provides a framework not only for practical application and ex-
perimentation but also for further theoretical investigation of the proofs-as-processes
theory.

Using the \( \pi \)-calculus correspondence of Abramsky, and Bellin and Scott as opposed to
the more recent session type correspondences was a deliberate choice given the mature
nature of this earlier work. However, this can also be viewed as a limitation since the
newer theories handle a number of issues in a simpler, more elegant way. For exam-
ple, the complicated \( \pi \)-calculus translation of replicable resources prevented us from
accommodating those in our theory, whereas their session type translation in the newer
theories is much simpler. We believe that our modular, rigorous, and conservative em-
bedding of the proofs-as-processes paradigm can be adapted to newer theories with
minimal effort, and all the functionality for process modelling and composition can be
reused, as long as these theories mature through time with more practical use-cases
and tools.
Another limitation of the current work lies in the use of CLL, which limits the system properties that we can reason about to qualitative ones about the information flow and the involved resources. In many real-world situations, it is often desirable to reason about quantitative properties, such as time and cost. Such capabilities are limited in our framework, and can only be obtained by the combination of our approach with other formal verification methods such as program logics for resource reasoning and model checking.

11.1.2 Clarification of the Original Foundations

The inconsistencies and ambiguities we discovered in the work of Rao et al. demonstrate the importance and benefits of formalising a given theory in a rigorous logical framework. Following the original theory faithfully and building upon it conservatively may require significantly more effort and may sometimes impose modelling restrictions. However, we now have mathematical guarantees that our methodology and claims are correct given the pen-and-paper soundness proofs provided by Bellin and Scott. In addition, despite the inherent complexity of using original π-calculus terms in our specifications (including the size and low readability, the explicit buffering, etc.), we were still able to model real-world situations using an intuitive graphical interface and without losing our connection to the original theory and the accompanying guarantees due to oversimplifications.

11.1.3 Logic-based Process Modelling Tools

Using the proofs-as-processes paradigm as the basis, we developed layers of high level tools that facilitate the specification and composition of processes. We implemented sufficient proof automation to enable intuitive actions that correspond to parallel, conditional, and sequential compositions. Analysing the properties of the resulting π-calculus compositions, we observed an interesting balance between asynchronous, concurrent execution and control over the execution order of the workflow where necessary. In addition, we introduced a diagrammatic user interface that makes our process modelling framework accessible by non-experts since the underlying logic and reasoning mechanisms are hidden. Overall, we demonstrated the applicability of linear logic theorem proving for the formal verification of process composition by providing a log-
ical framework driven by a diagrammatic interface.

One of the biggest challenges in the development of process modelling methodologies and tools has been the tradeoff between expressiveness and mechanization. On the one hand, domain experts and modellers require high expressivity, the ability to store a variety of information, and the tools to model many different, specialised cases. On the other hand, process developers require a robust translation of modelled workflows to the semantics of an executable language in order to produce a software implementation. Our approach arguably leans towards the needs of the latter, since the constructed \( \pi \)-calculus models have well-defined operational semantics. The abstract nature of our process specifications allows us to model a variety of processes for a large number of possible scenarios. However, this level of abstraction combined with the sometimes complicated interactions between processes, may not be easy to familiarise with, especially for a beginner. For example, enforcing certain constraints in a particular workflow may be more intuitive in an expressive modelling language with a large number of constructs such as BPMN than our CLL based models. This may become a prohibiting factor for some users of our system despite the guarantees of correctness that we provide. Researching the means to provide more expressivity and intuitive constructs to the user, either in the form of abbreviations or abstractions of CLL specifications, or by complementing the existing models with more meta-information and possibly different verification mechanisms, would certainly make our framework more accessible and usable for the average user.

11.1.4 Automated Workflow Deployment

We have developed the Scala PAPPILib library for the automated translation of both the atomic and composite \( \pi \)-calculus processes involved in our proofs-as-processes approach to executable code. This allows for an automated transition from an abstract process model to a deployable system. Assuming a concrete implementation of each abstract component process, code generation is completely automatic. This allows maximum flexibility, adaptability, and maintainability and makes our approach a practical solution to process modelling, including both the design and deployment stages.

In comparison to modern, industrial scale workflow execution tools, there are a number of standard utilities that are particularly useful, such as workflow monitoring, analytics, persistence through crashes and reboots, and others. Given the nature of the underlying
\(\pi\)-calculus executions, adding these to our deployed workflows may not be straightforward, but is certainly desirable for our tool to be used in industrial settings. An evaluation of the efficiency and scalability of the generated code would also be useful.

11.1.5 A Variety of Applications

Our logic-based process modelling approach has been applied to a variety of use-cases, including a non-trivial scenario for a holiday booking web agent and the modelling of real-world collaboration patterns in healthcare teams. In addition, a number of small projects have since spawned, aimed at investigating larger scale applications in healthcare, such as intra-hospital transfers of patients. The results so far have been promising and well received. We believe our approach tackles some core information flow issues in communication during standard healthcare practices, such as redundancies and inconsistencies, by producing well-formed guidelines as formally verified workflows.

11.2 Future Work

This thesis spans a variety of theoretical and practical domains, and so naturally offers numerous possibilities for future work. In what follows, we summarize some of our plans for future work as they were covered at the end of each individual chapter of this thesis.

- **The \(\pi\)-calculus formalisation:** Our current formalisation of the \(\pi\)-calculus is kept relatively simple, tailored to the needs of our pragmatic goal of process composition. It could benefit from improvements such as the embedding of reduction and congruence rules and bisimilarity and the use of advanced name handling libraries that would provide support for more complicated meta-theoretic analysis.

- **Proofs-as-processes formalisation:** Mechanizing the soundness and completeness proofs of Bellin and Scott, which we currently trust to be correct, would further contribute to the robustness of our implementation. In addition, generalising the embedding in such a way that the process translations can be replaced by more modern approaches, such as those described in Section 4.5.2, without the need to reimplement the process composition rules and tactics, would help keep our framework relevant as the modern theories evolve further.
• **Process specifications:** An interesting future undertaking would be to expand upon the reasoning capabilities of our framework in order to allow reasoning about the resources themselves or performing inference about quantitative properties. This can be accomplished, for instance, by expanding the CLL process specifications with richer type systems, combining them with other logic-based specifications appropriate for the use of other verifications techniques such as Hoare logic, logics for program resources, and model checking, or by lifting the imposed CLL polarity constraints and exploring the semantics of the currently disallowed process specifications. In addition, we are considering expanding the process specifications with additional meta-data that can contribute to more concrete code generation for each specified process.

• **Process composition:** Our practical experience so far has shown that, although our composition actions for sequential, parallel, and conditional composition, cover the majority of our workflow modelling needs, some specialised cases may occur that may require an improvement of the functionality of one of the actions or an implementation of a new composition action altogether. Investigating such cases and expanding the available composition actions would help improve the expressivity and user-friendliness of our framework.

• **Diagrammatic interface:** The current diagrammatic interface helps provide an intuitive picture of the underlying logical specification of a process composition. This abstract and intuitive picture comes at the cost of some of the workflow properties being concealed from the user. For example, we have reviewed cases where buffers are either hidden or shown in a convoluted way, and some of the ordering constraints in the workflow execution not being obvious to the user. Finding solutions to such visualisation problems is non-trivial and an interesting part of our future work.

• **Workflow deployment and code generation:** We have identified some cases where the generated code for a particular workflow can be further optimised. In addition, we plan to consider practical improvements to the deployed workflows, including live visualisation, monitoring, persistence through reboots, and improved simulation. Finally, we are investigating the possibility for automated checklist generation from a process specification corresponding to a Human Provided Service.

• **Applications:** In the future, we aim to explore more real-world domains beyond healthcare and web services. The generic nature of our approach opens up a wealth of possibilities and collaborations that we are keen to explore. In the healthcare do-
main in particular, there are ongoing projects where we are actively using the current framework for the modelling of integrated care pathways.

Overall, this work has potential for further projects both on a theoretical and practical level, both in an academic and an industrial scale, and we are keen to explore those in the future.

## 11.3 Concluding Remarks

We presented a formal verification approach to process modelling and composition. It is based on the proofs-as-processes paradigm, a theory which inherently guarantees a number of interesting properties for the information flow of a constructed workflow, including type correctness, information accounting, and deadlock-freedom. Our methodology incorporates a number of techniques and tools, built within the proof assistant HOL Light, and a user-friendly, diagrammatic interface. An automatic code generation procedure allows the deployment of our logic-based process models as concrete software systems. Non-trivial applications to the composition of web services and healthcare processes demonstrate the properties and benefits obtained by our generic approach.

Our work has multiple layers of development, from the components of the proofs-as-processes theory to the interface that hides the underlying engine, and brings together ideas from many different areas, including system design, business process modelling, linear logic, process algebras, verification by formalisation and embedding, theorem proving, automated and interactive reasoning, and code generation. Moreover, our work is extensible as demonstrated by the multiple ways in which its various components can be extended and optimised, thus creating a large number of potential paths for future research.

In conclusion, we would like to emphasize two main outcomes drawn from our work. Firstly, we clearly demonstrated the applicability of formal verification and theorem proving techniques in the highly pragmatic area of process modelling. It is clear that the mathematically guaranteed benefits of such an approach can add considerable value to industrial scale models and applications, enabling formal verification to have a high practical significance in this context. Secondly, our work is indicative of the complexity of bridging the gap between theory and practice. Given a well-researched theoreti-
ical result, such as the proofs-as-processes paradigm, with potentially multiple important benefits in practice, significant research and implementation effort is required for the pragmatic application of this result in order to obtain the claimed benefits. Such effort includes the development of practical tools, a deep understanding of the practical implications of every theoretical aspect, and non-trivial research in order to match practical needs to features of the theory and vice versa. Overall, we believe that this thesis is the embodiment of a fruitful connection between theory and practice.
Appendix A

Linear Logic Theorem Provers

There are several dedicated modern theorem provers that focus on proving formulas in linear logic automatically. As there are particular problems involved in the automation of such proofs, these systems are optimised to deal with them. In what follows we briefly describe four such theorem provers:

- **Lolli** (Hodas and Miller, 1994) is a logic programming language that uses an intuitionistic fragment of LL. It can be viewed as a refinement of λProlog (Nadathur and Miller, 1988), an extension of Prolog that allows, among other things, the use of implication in the goal and higher-order unification. The original implementation was done in ML but the currently most widely used is written and compiled in Linear Logic Prolog (Hodas et al., 1998).

- **Forum** (Miller, 1996) is a logic programming language that evolved from Lolli and is defined by a sequent calculus proof system. Its basic goal in comparison to Lolli was to extend into a classical (rather than intuitionistic) environment. It is often used as a framework to embed other programming languages (for example in the work by Bugliesi, Delzanno et al. (Bugliesi et al., 2000)) for formal verification. It has multiple available implementations, the most common one (and the one used by Rao et al. as described in Section 5.2) being the University of Malaga implementation (UMA Forum) (Lopez, 1998).

- **LINK** (Habert et al., 2002) is a framework that accommodates theorem provers for a number of versions of Multiplicative Linear Logics. In particular, it extends over mixed linear logic (MNL), commutative linear logic (MLL) and non-commutative (or cyclic) linear logic (MCyLL). It is based on automatic Proof
Nets (Girard, 1995a) construction, which is essentially a graph-theoretic representation that facilitates proofs in these logics. It is written in Ocaml.

- The Resource-Aware Planning System (RAPS) (Küngas, 2002) is a linear logic theorem prover used for planning. It uses propositional intuitionistic linear logic and takes advantage of the connection between this logic and Petri-Nets (Reisig, 1985), thus reducing the planning problem to a Petri-Net reachability problem (Mayr, 1981). RAPS is written in Java.

In our research, we are considered a composition tool where the corresponding CLL proof is performed interactively. Moreover, developing a framework that allows the attachment of process calculus terms to linear logic proofs was a considerable challenge. For these reasons, the linear logic theorem provers described here were not seen as relevant to the current work.

We note, however, that the use of these tools as external reasoners in an attempt to automate part of or the entire composition via proof and the possibility of then porting the results into our HOL Light based, process calculus supporting framework is an interesting consideration for possible future work.
Appendix B

Extra Theorems and Definitions

For the purposes of our $\pi$-calculus and CLL formalisations, we extended the HOL Light theories for lists, sets, and multisets with a number of extra theorems. Moreover, we introduced definitions for the $\text{DEL}$ function that deletes a particular member from a list (similar to the $\text{DELETE}$ function for sets) and the $\text{LDIFF}$ function that calculates the difference of two lists (similar to the $\text{DIFF}$ function for sets), and proved a number of theorems about these. In what follows, we present some of our proven theorems that may be useful in a more general context. Note that all variables are implicitly universally quantified.

- **Theorems about lists:**

$(\forall x. P(x)) \Rightarrow \text{FILTER} \ P \ l = l$
$(\forall x. \neg(P \ x)) \Rightarrow \text{FILTER} \ P \ l = []$
$\text{FILTER} \ (\lambda x. T) \ l = l$
$\text{FILTER} \ (\lambda x. F) \ l = []$
$\text{FILTER} \ (\lambda x. P(x)) (\text{FILTER} \ (\lambda x. Q(x)) \ l) = \text{FILTER} \ (\lambda x. P(x) \land Q(x)) \ l$
$\text{MAP} \ (\lambda x. x) \ l = l$
$\text{MAP} \ f \ (\text{APPEND} \ l \ k) = \text{APPEND} \ l \ k \iff \text{MAP} \ f \ l = l \land \text{MAP} \ f \ k = k$
$\neg(\text{MEM} \ n \ l) \iff \text{ALL} \ (\lambda x. \neg(x = n)) \ l$
$\text{ITLIST} \ \text{MAX} \ (h :: t) \ x = \text{ITLIST} \ \text{MAX} \ t \ (\text{MAX} \ h \ x)$

- **Definition of $\text{DEL}$ and related theorems:**

Definition: $l \ \text{DEL} \ x = \text{FILTER} \ (\lambda z. \neg(z = x)) \ l$
\[ \text{Definition of } \text{LDIFF and related theorems:} \]

\[ \text{Definition: } k \text{ LDIFF } l = \text{FILTER } (\lambda z. \neg (\text{MEM } z l)) \, k \]

\[ \text{[ ] LDIFF } l = [ ] \]
\[ (h :: t) \text{ LDIFF } l = i f (\text{MEM } h l) \text{ then } (t \text{ LDIFF } l) \text{ else } (h :: (t \text{ LDIFF } l)) \]
\[ \text{MEM } x (k \text{ LDIFF } l) \iff \text{MEM } x k \land \neg (\text{MEM } x l) \]
\[ \neg (\text{MEM } x l) \Rightarrow \neg (\text{MEM } x (l \text{ LDIFF } k)) \]
\[ \text{MEM } x k \Rightarrow \neg (\text{MEM } x (l \text{ LDIFF } k)) \]
\[ \neg (\text{MEM } x (l \text{ LDIFF } (x :: t))) \]
\[ l \text{ LDIFF } [ ] = l \]
\[ (l \text{ LDIFF } t) \text{ LDIFF } t = l \text{ LDIFF } t \]
\[ (l \text{ LDIFF } m) \text{ LDIFF } n = l \text{ LDIFF } (\text{APPEND } m n) \]
\[ l \text{ LDIFF } l = [ ] \]
\[ l \text{ LDIFF } (h :: t) = (l \text{ DEL } h) \text{ LDIFF } t \]
\[ \text{FILTER } (\lambda x. P(x)) \, (l \text{ LDIFF } k) = (\text{FILTER } (\lambda x. P(x)) \, l) \text{ LDIFF } k \]
\[ \text{set_of_list}(l \text{ LDIFF } l) = \text{set_of_list}(l) \text{ DIFF } \text{set_of_list}(l) \]

\[ \text{Theorems about sets:} \]
\[ s \text{ DELETE } a \text{ DIFF } t = (s \text{ DIFF } t) \text{ DELETE } a \]
\[ \text{DISJOINT } (s \text{ DIFF } p) \, p \]
\[ (s \text{ DIFF } p) \text{ UNION } p = s \text{ UNION } p \]
\[ \text{IMAGE } (\lambda x. x) \, s = s \]

\[ \text{Theorems about multisets:} \]
\[ m \uplus n = n \uplus m \]
\[ \emptyset \uplus m = m \]
Appendix C

Application of the Commutativity Filter on a Process

In Section 5.5.3, we analysed the example of filter $TimesComm_x^y(P_a, P_b)$ that corresponds to the commutativity law of $\otimes$. More specifically in Section 5.5.3.2, we discussed that the filter is only useful when it communicates with another process $P$ with an output of type $A \otimes B$. The composition of $P$ with $TimesComm_x^y(P_a, P_b)$ results in a composite process $Q$ with output $B \otimes A$, as captured in the following lemma:

$$
\vdash P : \vec{w} : \Gamma, \ a : (A \otimes B) \vdash P_a : o_a : A, \ i_a : A^\perp \vdash P_b : o_b : B, \ i_b : B^\perp \vdash Q : \vec{w} : \Gamma, \ b : (B \otimes A)
$$

Using the proofs-as-processes paradigm, we extract the $\pi$-calculus description of $Q$ which is as follows:

$$
Q = Cut^z (P, TimesComm_x^y(P_a, P_b)) = (\nu z) \ (P[a/z] \parallel TimesComm_x^y(P_a, P_b))
$$

According to (C.1), process $P$ has a composite port $a$. In order to swap its arguments, it needs to connect to the main input port $x$ of $TimesComm_x^y(P_a, P_b)$. For this reason, $a$ and $x$ are both substituted by $z$ (as dictated by the $Cut$ rule).

To observe this interaction more concretely, we can use our $\pi$-calculus representation of composite ports. In Section 5.3.3, we argued that a composite port $(c_a, c_b)$ for process $P$ corresponds to the following $\pi$-calculus process (with the names renamed to avoid
Appendix C. Application of the Commutativity Filter on a Process

Let us assume that \( c_a \) and \( c_b \) carry messages \( m_a \) and \( m_b \) of types \( A \) and \( B \) respectively. For simplicity, we also assume that \( A \) and \( B \) are atomic so that we can make use of the TimesComm\( _{\nu}^o (P_a, P_b) \) from (5.20):

\[
P_a = I_{i_a, o_a} = i_a(a) \cdot \overline{a}(a) \cdot 0 \\
P_b = I_{i_b, o_b} = i_b(a) \cdot \overline{b}(a) \cdot 0 \\
TimesComm\( _{\nu}^o (P_a, P_b) = x(i_a, i_b) \cdot (\nu o_b, o_a) (\overline{y}(o_b, o_a) \cdot (i_b(b) \cdot \overline{b}(b) \cdot 0 \parallel i_a(a) \cdot \overline{a}(a) \cdot 0) \parallel (c_b(b)) \cdot \overline{b}(b) \cdot 0 \parallel i_a(a) \cdot \overline{a}(a) \cdot 0)) \quad (C.3)
\]

If process \( P \) above interacts with TimesComm\( _{\nu}^o (P_a, P_b) \) (i.e. TimesComm\( _{\nu}^o (P_a, P_b) \) where \( x \) is substituted by \( z \)) using the parallel composition we gave for \( Q \) in (C.2), the \( \pi \)-calculus reduction will give us the following result:

\[
P || (\nu m_a) (\nu \overline{a}(m_a) \cdot 0) || (\nu m_b) (\nu \overline{b}(m_b) \cdot 0) || (\nu o_b, o_a) (\overline{y}(o_b, o_a) \cdot (i_b(b) \cdot \overline{b}(b) \cdot 0 \parallel i_a(a) \cdot \overline{a}(a) \cdot 0)) \parallel (c_b(b)) \cdot \overline{b}(b) \cdot 0 \parallel i_a(a) \cdot \overline{a}(a) \cdot 0)
\]

There are no further available reductions at this point, because TimesComm is now waiting to output through \( y \). Since \( y \) corresponds to the new composite port with the swapped arguments, it is expected to interact with a third process that will receive the new output of type \( B \otimes A \). For the sake of the example, let us assume this third process is actually of the form \((d_b, d_a).R\) and accepts input. Based on our \( \pi \)-calculus representation of composite input ports shown in (5.6), \( R \) will have the following form:

\[
y(d_b, d_a) . (d_b(r_b) . 0 || d_a(r_a) . 0) \parallel R
\]

Interacting with this process (through another application of the Cut rule) will give the following result:
\[ P || (\nu m_a) (c_a \langle m_a \rangle).0 || (\nu m_b) (c_b \langle m_b \rangle).0 \]
\[ c_b(b) o_b(b).0 || c_a(a) o_a(a).0 \]
\[ o_b(r_b).0 || o_a(r_a).0 || R \]

Note that \( d_b \) and \( d_a \) have been substituted by \( o_b \) and \( o_a \) respectively according to the \( \pi \)-calculus reductions rules. It is now clear that message \( m_a \) will be buffered from channel \( c_a \) to channel \( o_a \), whereas \( m_b \) will be buffered from channel \( c_b \) to channel \( o_b \). Indeed, consecutive \( \pi \)-calculus reduction will give us the following result:

\[ P || o_b\langle m_b \rangle.0 || o_a\langle m_a \rangle.0 || o_b(r_b).0 || o_a(r_a).0 || R \]

From this point on, notice how \( R \) will receive \( m_b \) through its first argument \( d_b \) (substituted by \( o_b \) and \( m_a \) through its second argument \( d_a \) (substituted by \( o_a \)). Swapping the order of \( m_a \) and \( m_b \) has, therefore, been accomplished.

In the analysis above we have explained the complex translation of the commutativity property for tensor in \( \pi \)-calculus.

Rao et al. introduce their structural congruence rule without any supporting evidence of correctness or explanation of its derivation. What they represent as a simple composite port swapping from \((a,b)\) to \((b,a)\) is glossing over the entire derivation that involves relatively complex \( \pi \)-calculus reductions. Having formalised suitable rules for capturing the fundamental notions of the proofs-as-processes paradigm we are able to formally verify this property and its correspondance in the process calculus.
Appendix D

Code Excerpts

In what follows, we present some code excerpts to demonstrate some of the implementation aspects of our framework and the complexity of the involved code.

The implementation code of the BUFFER_TAC tactic presented in Section 7.3.1 is shown in Figure D.1. The tactic relies on the breakdown of the single output of the goal, which is retrieved using the find_output function. Since this is an output, there are only three possible cases of it being a variable (atomic), an application of LinTimes (⊗), or an application of LinPlus (⊕). Notice that even this conceptually simple tactic assumes the polarization of CLL connectives and that our restrictions for CLL specifications (see Section 6.2) hold. The tactics llrule and llrule_tac are part of our meta-level natural deduction implementation (see Section 4.4.5.1) and apply the CLL inference rules ll_times, ll_par, ll_with, ll_plus1, and ll_plus2 (corresponding to the ⊗, ⊃, &, ⊕R, and ⊕L rules respectively), whereas the llid tactic applies the identity axiom appropriately. Finally, the locally defined timestac tactic is responsible for dividing the context of a sequent for the appropriate use of the ⊗ rule (i.e. so that each input is paired with the corresponding output).

Next, Figure D.2 demonstrates the HOL Light proof script for the Cut′ rule described in Section 7.2.4. Note that MIMP_TAC is a tactic from the Isabelle Light library that eliminates meta-level implication, ll_cut is the CLL Cut rule, and piSUBN1_I is the following theorem about π-calculus substitution, which we have already proven in HOL Light:

\[ \forall P. \forall x. P \left[ x/x \right] = P \]
let rec (BUFFER_TAC:tactic) =  
    let timestac lh rh ((_,tm) as gl) =  
        let nlh,nrh = hashf mk_llneg (lh,rh) in  
        let nltm = mk_msing (find_ll_term ((=) nlh) tm)  
        and nrtm = mk_msing (find_ll_term ((=) nrh) tm) in  
            (llrule_tac [('A:LinProp',lh);  
                          ('B:LinProp',rh);  
                          ('G:(LinTerm)multiset',nltm);  
                          ('D:(LinTerm)multiset',nrtm)] ll_times) gl  
    in fun ((_,tm) as gl) -> try (  
        let out = find_output tm in  
        if (is_var out) then (  
            (llid out) gl  
        ) else  
            let comb, args = strip_comb out in  
            if (comb = 'LinTimes') then  
                let lh, rh = hd args, (hd o tl) args in  
                let tac =  
                    ONCE_REWRITE_TAC[NEG_CLAUSES] THEN  
                    (llrule ll_par) THEN  
                    (timestac lh rh)  
                THEN BUFFER_TAC in  
                tac gl  
            else if (comb = 'LinPlus') then  
                let tac =  
                    ONCE_REWRITE_TAC[NEG_CLAUSES] THEN  
                    (llrule ll_with) THENL  
                    [ llrule ll_plus1 ; llrule ll_plus2 ]  
                THEN BUFFER_TAC in  
                tac gl  
            else failwith "BUFFER_TAC"  
        ) with Failure _ -> failwith "BUFFER_TAC";;

Figure D.1: Implementation code for BUFFER_TAC.
let ll_cut' = prove (  
  '⊢ {x: C⊥} ⊎ Δ (Q) ⇒  
  ⊢ Γ ⊎ {x: C} (P) ⇒  
  ⊢ Γ ⊎ Δ ((v x) (P || Q))',  
MIMP_TAC THEN REPEAT DISCH_TAC  
THEN subgoal_tac '⊢ Γ ⊎ Δ ((v x) (P[x/x] || Q[x/x]))',  
THENL [ l1rule_tac ['C', 'C'] ll_cut; simp[piSUBN1_I] ]  
THEN llassumption);;  

Figure D.2: HOL Light proof script for the Cut' rule.

In Figure D.3 we present the HOL Light tactic LL_FILTER_CUT_TAC. This tactic reconstructs the proof of a filter as discussed in Section 5.5.3.2. It allows the construction of a filter based on any proven property thm of CLL, such as the commutativity property of ⊗:

\[ \vdash P_a :: o_a : A, \ i_a : A^⊥ \vdash P_b :: o_b : B, \ i_b : B^⊥ \]
\[ \vdash \text{TimesComm}_{xy}^Z (P_a, P_b) :: x: (A \otimes B)^⊥, \ y: (B \otimes A) \]

Using this proven property and given the corresponding filter lemma as a goal, it performs the proof automatically. We remind the reader that a filter lemma allows the direct application of the corresponding CLL property on a process P. In the case of the commutativity property of ⊗ given above, the filter lemma is the following:

\[ \vdash P :: \vec{w} : \Gamma, \ a: (A \otimes B) \vdash P_a :: o_a : A, \ i_a : A^⊥ \vdash P_b :: o_b : B, \ i_b : B^⊥ \]
\[ \vdash \text{Cut}^Z (P, \text{TimesComm}_{xy}^Z (P_a, P_b)) :: \vec{w} : \Gamma, \ b: (B \otimes A) \]

Note that the tactic assumes a correct description of both the CLL property and the goal, with the appropriate schematic assumptions specified. The llcut tactic uses the CLL Cut rule with the given property thm, whereas the assumption matching tactic llassumption matches the assumptions of the property (that need to be discharged for the successful application of the Cut rule) with the schematic assumptions of the lemma in order to complete the proof.

Finally, the HOL Light proof script for the example of the credit card workflow using Strategy A (see Section 7.5.1) is shown in Figure D.4. The mk_atomic_service command introduces a new atomic process given a name and a valid CLL specification. The gs command sets the CLL specification of the requested process as a goal, whereas the e command applies a tactic to the current goal. In this case, we use the
let LL_FILTER_CUT_TAC thm = 
  MIMP_TAC THEN REPEAT DISCH_TAC 
  THEN llcut thm 
  THENL [ llrule thm; llassumption ] 
  THEN llassumption;;

Figure D.3: A useful tactic for constructing filters.

LABEL
JOIN
TAC

tactic which corresponds to the JOIN action. Note that, by convention, the resulting composition after the use of a JOIN action receives the name of the process that was given as the action’s second argument. However, this does not restrict the user from reusing the original atomic process with that name (by attaching a _0 suffix on the label). More generally, the system allows increased flexibility through the use of indexed labels (although this is not demonstrated in this example). Finally, the qed command completes a proof and stores the resulting composite process for further use.

mk_atomic_service "CreditCardInit" '{cci_pr:PAYMENT_INFO⊥} ⊔
{cci_out:TRANSACTION ⊗ PIN_REQ}'';
mk_atomic_service "UserPINInput"
{'upi_req:PIN_REQ⊥} ⊔ {upi_out:PIN}'';
mk_atomic_service "CreditCardTransaction" '{cct_pin:PIN⊥} ⊔
{cct_tr:TRANSACTION⊥} ⊔ {cct_out:PAYMENT ⊕ EX_BAD_PIN}'';
gs '∃ CreditCardPayment c0 co. ⊢ {c0:PAYMENT_INFO⊥} ⊔
{co:PAYMENT ⊕ EX_BAD_PIN}{CreditCardPayment}'';
e (LABEL
JOIN
TAC "CreditCardInit" "UserPINInput");;
e (LABEL
JOIN
TAC "UserPINInput" "CreditCardTransaction");;
qed "CreditCardPayment" "CreditCardTransaction";;

Figure D.4: HOL Light proof script for Strategy A of the composition for the credit card example.
Bibliography


Bibliography


Typesafe Inc (2013). Akka Scala Documentation, Release 2.2.3. [http://doc akka.io/docs/akka/2.2.3/AkkaScala.pdf](http://doc akka.io/docs/akka/2.2.3/AkkaScala.pdf)

UK Department of Health (2009). A guide to the implications of the european working time directive for doctors in training. [http:](http://)


