Towards a Formal Method for Distributed Object-Oriented Systems

Malcolm Tyrrell

A thesis submitted to
the University of Dublin
for the degree of
Doctor in Philosophy.

2003

Department of Computer Science, University of Dublin.

Declaration

This thesis has not been submitted as an exercise for a degree at any other University. Except where otherwise stated, the work described herein has been carried out by the author alone. This thesis may be borrowed or copied upon request with the permission of the Librarian, University of Dublin, Trinity College. The copyright belongs jointly to the University of Dublin and Malcolm Tyrrell.

Malcolm Tyrrell

Acknowledgements

I would first like to express my gratitude to Andrew Butterfield, my supervisor, for his advice and support. I am grateful for the guidance given to me by Alexis Donnelly during the early stages of my research. Arthur Hughes also devoted time and effort to reading my work and I would like to thank him here.

I am grateful to Enterprise Ireland for their generous funding.

I would like to thank my colleges in the Foundations and Methods Group for many interesting discussions about almost everything (including, occasionally, foundations and methods).

Finally, I would like to thank my family and friends for their patience.

Summary

We present two components of a special purpose formal method intended to support the modelling and development of distributed object-oriented systems. The first component is a language, called Oompa, in which designs for these systems can be expressed. The second is a behavioural specification language called sequential process specifications.

Oompa is a strongly-typed class-based object-oriented language. Its objects have private state and support multiple concurrent invocations. The primitive operations of its methods support both object-based and channel-based behaviour. We provide two operational semantics for Oompa; a reduction relation, which is intuitive, and a labelled transition system, which is superior for reasoning. We define a type system for Oompa which is statically checkable. We also give thorough consideration to issues of naming in Oompa.

Sequential process specifications allow abstract behaviour to be specified and incrementally refined. The language has two binary operators, similar to CSP's internal and external choice, which assert that a system may or must have certain behaviour. We use sequential process expressions, which form the core of CCS, to describe specific patterns of behaviour and we define what it means for one to satisfy a specification. In terms of this notion of satisfaction, we define a semantics and a refinement relation for the language. Although similar to CSP, we show that our language is semantically different.

We demonstrate how our formal method currently supports development by deriving two designs for a concurrent dictionary object.

Contents

1 Introduction **14**

1.1 Distributed Object-Oriented Systems 16

1.1.1 CORBA 17

1.1.2 The CORBA Object Model 19

1.1.3 Our Use of the CORBA Object Model 20

1.2 Our Method 22

1.2.1 The Formalism: Oompa 22

1.2.2 The Specification Language: Sequential Process Specifications 25

1.2.3 The Approach 26

1.3 Contributions 26

1.4 How to Read this Thesis 27

1.5 Synopsis 28

2 The State of the Art **30**

2.1 State-Based Approaches to Concurrency 31

2.1.1 The Rely/Guarantee Approach 32

2.1.2 The Refinement Calculus and Action Systems 33

2.2 Process Calculi 35

2.2.1 CCS 35

2.2.2 CSP . . . 36

2.2.3 π-Calculus . . . 37

2.2.4 Pict . . . 39

2.2.5 Join Calculus . . . 39

2.3 Concurrent Object-Oriented Approaches . . . 41

2.3.1 POOL . . . 42

2.3.2 $\pi o \beta \lambda$ . . . 43

2.3.3 TyCO . . . 44

2.3.4 **concς** . . . 45

2.3.5 OC . . . 47

2.3.6 BOOM and ODAL . . . 47

2.4 Evaluating the Methods . . . 48

2.4.1 State-Based Methods . . . 48

2.4.2 Process Calculi . . . 49

2.4.3 Concurrent Object-Oriented Approaches . . . 50

2.5 Our Work in Context . . . 51

2.6 Summary . . . 52

3 Static Oompa **54**

3.1 Syntax . . . 54

3.1.1 The Syntax of the π-Calculus . . . 55

3.1.2 Names . . . 56

3.1.3 Code . . . 57

3.1.4 Definitions . . . 59

3.1.5 Types . . . 60

3.1.6 Example of Oompa's syntax . . . 62

3.2 Issues in the design of Oompa . . . 63

3.3 Well-Formedness . . . 65

3.4 Renaming . . . 66
3.4.1 Permutations of $\mathcal{N}$ . . . 68
3.4.2 How to Pick New Names . . . 69
3.4.3 Shoves . . . 70
3.4.4 The Action of a Renaming on Code . . . 70
3.4.5 The Definition of Substitution on Code . . . 73
3.5 α-Equivalence for Code . . . 74
3.6 Summary . . . 78

4 Dynamic Oompa 79
4.1 Agent-Based Dynamic System . . . 80
4.1.1 Syntax . . . 80
4.1.2 Renaming . . . 84
4.1.3 Equivalence . . . 86
4.1.4 Semantics . . . 89
4.2 Configuration-Based Dynamic System . . . 95
4.2.1 Syntax . . . 96
4.2.2 Renaming . . . 99
4.2.3 Equivalence . . . 102
4.2.4 Semantics . . . 106
4.2.5 Weak Bisimulation . . . 114
4.3 Relationship . . . 115
4.3.1 The Flatten Algorithm . . . 115
4.3.2 Properties of Flatten and the Relationship Theorem . . . 118
4.4 Summary . . . 120

5 The Type System 122
5.1 Typing, Type Violations and Type Safety . . . 124

5.1.1 The Typing System 125
5.1.2 Type Violations 126
5.1.3 Type Safety 127
5.2 Oompa Type Trees 130
5.2.1 Trees and Subtrees 130
5.2.2 Interpreting Types as Trees 132
5.2.3 Expanding Definitions 135
5.3 Subtyping 136
5.3.1 Tree Simulation 138
5.3.2 The Subtyping Algorithm 139
5.4 Soundness of the Type Safety System 142
5.4.1 Dynamic Type Safety 143
5.4.2 Preservation of Dynamic Type Safety 145
5.4.3 Configurations and the Type System 145
5.5 Comparison with Other Work on Type Systems 147
5.6 Summary 149

6 Sequential Process Specifications 150

6.1 Expressing Abstract Behaviour 151
6.2 CCS 153
6.2.1 Nondeterminism in CCS 157
6.3 Some Theory of Sequential Process Expressions 161
6.4 Sequential Process Specifications 164
6.4.1 Structural Equivalence 166
6.4.2 Satisfaction 167
6.4.3 Semantics 173
6.5 Refinement 175
6.5.1 Simple Refinement Laws 176

6.5.2 Refining Simulation 178
6.6 Example: A Scheduler 182
6.6.1 Informal Requirements 182
6.6.2 Model 182
6.6.3 Specification 183
6.6.4 Further Requirements 185
6.6.5 Respecifying the Scheduler 186
6.7 Using Sequential Process Specifications with Oompa 187
6.8 Related Work 189
6.9 Summary 191

7 A Development in Oompa 193

7.1 Behavioural Requirements of a Concurrent Dictionary 194
7.1.1 Abstract Data Type for a Dictionary 194
7.1.2 Sequential Object-Oriented Dictionary 195
7.1.3 Concurrent Object-Oriented Dictionary 198
7.2 Behavioural Specification for a Concurrent Dictionary 200
7.2.1 Specification 200
7.2.2 Alternative Specification 207
7.3 Designs for a Concurrent Dictionary 210
7.3.1 A Standard Design 211
7.3.2 Justifying the Standard Design 212
7.3.3 An Alternative Design 218
7.3.4 Justifying the Alternative Design 220
7.3.5 Further Development 225
7.4 Summary 226

8 Conclusions and Future Work **228**

8.1 Overview 228

8.2 Future Work 230

A Isolated Sum **234**

A.1 Definition 234

A.2 Some Properties of Isolated Sum 236

A.3 n-Ary Isolated Sum 239

B Technical Results for Oompa **242**

B.1 Results about Code 242

B.2 Results about Agents 251

B.3 Results about Configurations 253

B.3.1 Configuration Renaming 254

B.3.2 α-Equivalence for Configurations 257

B.3.3 Structural Equivalence for Configurations 265

B.3.4 Labelled Transition System 278

B.4 Flatten and the Relationship 282

C Technical Results for the Type System **315**

C.1 Type Tress 315

C.2 The Subtyping System 322

C.2.1 Termination 323

C.2.2 Completeness 325

C.2.3 Soundness 327

C.3 Soundness of the Type Safety System 330

C.3.1 Preservation of Dynamic Type Safety 332

C.4 Configurations and the Type System 341

D Simulations **347**

D.1 Refining Simulations for the Scheduler Example 347

D.2 Satisfying Simulations for the Scheduler Example 350

D.3 Refining Simulations for Chapter 7 355

Bibliography **361**

List of Tables

3.1 Equivalence Rules for Code α-Equivalence 75
3.2 Congruence Rules for Code α-Equivalence 76
3.3 Change of Bound Name Rules for Code α-Equivalence 77

4.1 Rules for Agent Structural Equivalence 87
4.2 Rules for Agent α-Equivalence 88
4.3 Rules for Agent α-/Structural Equivalence 89
4.4 General Rules for the Agent-Based Operational Semantics . . 92
4.5 Channel Rules for the Agent-Based Operational Semantics . . 92
4.6 Object Rules for the Agent-Based Operational Semantics . . . 93
4.7 Reflexive Transitive Closure of the Operational Semantics . . . 95
4.8 Rules for Configuration α-Equivalence 103
4.9 Rules for Configuration Structural Equivalence 105
4.10 General Rules for the Labelled Transition System 109
4.11 Local Rules for the Labelled Transition System 109
4.12 Labelled Rules for the Labelled Transition System 110
4.13 Interaction Rules for the Labelled Transition System 110
4.14 Rules for Experiments 113

5.1 Rules of the Typing System 125
5.2 Rules for the Type Safety of Code 129

5.3 Rules for the Type Safety of Definitions 130
5.4 Rules for the Subtyping Algorithm 141
5.5 Rules for the Dynamic Type Safety of Code 144

6.1 Labelled Transition Rules for Sequential Process Expressions . 154
6.2 Labelled Transition Rules for CCS 155
6.3 Rules for Equivalence 166

Chapter 1

Introduction

The ultimate aim of this research is to provide a formal method to support the modelling and development of distributed object-oriented systems.

Individual formal methods can be identified as either general purpose or special purpose. A *general purpose formal method* seeks to be applicable to the modelling and development of diverse systems. It will be semantically neutral, so it will not be focused on any particular type of system, and should avoid implementation bias, so it will promote development that is strictly defined by the problem. Perhaps the most significant advantage to a general formal method is a practical one: familiarity for users. Good examples of general formal methods are VDM [Jon90] and Z [Spi92].

On the other hand, general purpose formal methods have an intrinsic disadvantage. This is the divide, or *semantic gap*, between the level at which the method's elements operate and the level at which the target system operates. This divide means that much of the effort of building a model might be expended on creating higher-level constructs in which the model proper can be built. Furthermore, real world systems are not totally distinct. The development of similar systems will involve solving the same problems and

often developing the same solutions. This repetition of work is exacerbated in the case where there is a wide semantic gap between the method and systems being developed[1].

A *special purpose formal method* [Sch94] seeks to tackle a specific type of system only and, by doing so, can be much semantically closer to a target system than a general purpose formal method can. Assumptions about the target systems can be built into the method, bridging the semantic gap, so the user can start modelling and developing at the level at which the target system is most suitably considered. The main disadvantage of using special purpose formal methods is that a new method must be learnt for each problem domain. Another disadvantage is that a special purpose formal method is likely to have a smaller community of developers and users than a general purpose formal method.

An analogy is the field of programming languages where a distinction between general purpose languages and special purpose languages can also be drawn [LR88]. There exist many programming languages touted as general purpose (such as C++ [Str91]), i.e. suitable for developing any type of software. While the obvious advantage is that users only need to become familiar with a single language, there can be little doubt that work is frequently duplicated within and between these languages[2]. An example of a special purpose programming language would be the database query language SQL [vdL89].

[1] This problem can be somewhat mitigated by techniques which promote reuse. For example, Object-Z [Smi99] uses object concepts as a structuring mechanism primarily to improve reuse between specifications.

[2] As with formal methods, techniques for reuse can help, for example the development of user libraries for certain problem domains. However, given the complexity of some libraries (e.g. Microsoft Foundation Classes [SWM96]) we might argue that learning a library can be as difficult as learning a new language.

In general, developers have a choice between using a suitable language for each problem or forcing a single language to cope with all problems.

We are considering distributed object-oriented systems, so our problem domain is fixed. We have decided to provide a special purpose formal method to provide support for developing and modelling these systems. This thesis presents the two components of such a method: a language for expressing designs, called Oompa, and a behavioural specification language, called sequential process specifications. Our first task will be to classify the problem domain.

1.1 Distributed Object-Oriented Systems

A *distributed system* is a computational system consisting of several interacting components residing at separate locations. Actual distributed systems may differ dramatically: in particular, the concept of a location and the mechanisms supporting interactions may be quite different. Nevertheless, we can delimit them with a broad definition, as above.

Unfortunately, *object-oriented systems* are not so easy to classify. There is no real agreement over what exactly constitutes object-orientation [III97]. Most would agree that it structures code into uniquely identified *objects* and, within objects, into *methods*; the act of causing the computation described in a method to be performed is almost always referred to as *invocation*. Beyond these necessary properties, insufficient to classify even the broadest definition in use, there is little common ground.

This difficulty in defining object-oriented systems leaves us with two options. The first option is to remain agnostic about the particular interpretation and provide a *meta-model* within which a variety of interpretations can

be described. An example of this approach is [Öve00]. Naturally, a semantic gap will remain when we try to discuss actual systems. The second option is to choose a single interpretation and target our formal method at it. This is in line with our decision to provide a special purpose formal method and we follow this second path. However, when choosing our single interpretation of "object-orientation", we will deliberately choose a general one.

A typical approach describes how to build an object-oriented system from the bottom up using a specific definition of object. An example is [Mey97]. An alternative and more general approach is to fit object concepts onto existing systems as unobtrusively as possible. This is the approach of the CORBA object model [OMG98][3], from which we take inspiration.

1.1.1 CORBA

Modern distributed systems can have astounding complexity, the Internet being the prime example. There are substantial difficulties involved in designing software which avails of the facilities these systems provide yet operates correctly and efficiently under the conditions to which these systems are subject. To help make the complexity manageable, software known as *middleware* was introduced. Middleware, at the very minimum, provides a high-level protocol for participating in distributed communications. The purpose of the high-level protocol is to hide the complexity of the underlying system-level protocols. Some examples of middleware are Remote Procedure Call (RPC) [BN83], Remote Method Invocation (RMI) [WRW96], Enterprise Java Beans (EJB) [MH99], Distributed Component Object Model (DCOM) [EE98]

[3]Since its introduction in 1991 [OMG91], CORBA has undergone continual refinement and expansion. To discuss CORBA consistently, we need a single slice through its development history. We fix [OMG98] as our reference document for CORBA.

and Common Object Request Broker Architecture (CORBA) [OMG98]. All of these enable a distributed communication to be performed in a fashion similar to that of a language level function call or method invocation.

A feature of RPC, RMI and EJB is that their implementations tend to support a single programming language; a feature of DCOM is that it primarily supports a single operating platform. However, to benefit most from the power of distributed systems, multi-language and multi-platform development must be supported. As well the obvious argument that certain programming languages are better suited to certain software, which obviously suggests such multi-language development, another force which acts on distributed software development is due to *legacy software* [BS95]. Large distributed software systems are not always built from scratch and often involve the reuse of older software, possibly by wrapping it with distribution-aware code. This saves on the expense of redeveloping and retesting the software, migrating data to newer formats and retraining staff. A consequence of bringing this legacy software into the distributed environment is that we cannot assume that the components of our distributed system are written in any particular language or run on any particular platform. The middleware protocols should therefore be supported across the various languages and platforms typically used.

CORBA is a middleware solution which uses object-orientation to enable these disparate software entities to interact. It consists of standards which specify protocols, and software to implement them, which are usable from many of the languages and platforms a modern distributed system might involve. The principle which makes CORBA work is that, regardless of how a piece of software is implemented, it participates in the distributed system by acting like an object (or collection of objects). Object-oriented concepts

are mapped onto each programming language and CORBA uses method invocation to expose its middleware protocol.

1.1.2 The CORBA Object Model

For software entities to act as objects, or issue method invocations, there must be a common notion of what these concepts are — a shared ontology. According to the OMG (Object Management Group, the industry body which provides the CORBA standards), an *object model* "provides an organised presentation of object concepts and terminology" [OMG98]. The *CORBA Object Model* establishes the shared view of object-orientation elements of the system must have to interact effectively. We present a brief (and mildly simplified) description of that model here.

The object model takes a very abstract view of a distributed system. The parts that make up the system are identified as belonging to one of the following categories: data, code and execution engines. Many different forms of computation may actually be taking place in the system but, for the purposes of the object model, we can ignore the details beyond the interaction of these three kinds of entity. As usual, *code* is a description of some computations to perform and *data* is some state component of the system or the real world. The work is done by the *execution engines* which interpret the code and perform the computations, possibly altering some of the data. It is not required that all code be meaningful to all execution engines but, rather, that each piece of code is meaningful to some execution engine.

A powerful abstraction comes from viewing the process of code being performed by an appropriate execution engine as providing a *service*. The behaviour of a distributed system can now be seen in terms of the performing of services. Communication between elements of the system can be seen as

requests that specific services be performed. Code which provides a service is called a *method* and, unsurprisingly, the notion of communication allows values to be exchanged, so a method may require *parameters* and produce *return values*. *Method activation* is the term used when a method is performed.

Next, we associate methods which share some relationship, such as the ability to alter a specific piece of data, into an entity called an *object*. The significance of an object is that it has a unique identifier, a value called an *object reference*, and that it provides access to its services in a standard way: in a syntactic form called an *interface*. An interface lists the object's methods, providing their name and their *signatures*. An object may have several interfaces associated with it, each exporting different sets of services. In fact, the CORBA object model does not insist that all the computational entities in the system belong to objects; a *client* is any computational entity which is capable of requesting that an object performs one of its services.

1.1.3 Our Use of the CORBA Object Model

A fundamental design goal behind CORBA was to facilitate distributed computing. A fundamental design goal behind the CORBA object model was to allow diverse systems, whether implemented in an object-oriented language or not, to participate in object-oriented systems. It seems that the CORBA object model is an ideal way to establish a general definition of "distributed object-oriented systems".

We now have a classification of our problem domain but find that, for our work, it allows too general a class of systems. There are many facilities within such a system that could make our goal extremely difficult. Consequently, we make some simplifying assumptions.

Firstly, we assume that all data in the system belongs to some object

and that all running code in the system belongs to a method of some object. Under this assumption, we can still discuss systems where state and code can exist separate from objects: we view them as belonging to some trivial wrapper objects which need have no structural significance.

Secondly, we assume that all state is private to some object. This assumption also seems reasonable since, if we really want to expose an object's state, we can provide the object with unrestricted update and access methods.

Thirdly, we choose a single approach to dealing with concurrent invocations. If an invocation arrives when there are processes already running at the object, a choice must be made between making the invocation wait or creating a new process to deal with it. CORBA allows this choice, called an *activation policy*, to be made in an object dependent way. The default policy is to create a new process[4] and we choose to follow this since it is the most general and other policies can be modelled within it. For example, single-threaded objects could be built using locking.

Fourthly, we consider systems to be class based. This might seem a strong assumption but it really just clarifies an aspect of system we have not yet discussed: how code is described and the way in which objects are introduced into the system.

Lastly, we prohibit two features which give rise to richer, but far more complex, systems: we exclude function passing and code mobility. The earlier assumptions do not significantly affect the class of systems under discussion although, were we to model an actual CORBA system, they might induce a mild semantic gap. This last assumption does impact the class of systems

[4] Using the terminology of [Nie93], which presents a partial categorisation of concurrent object systems, we can say that synchronisation is "orthogonal to encapsulation" and objects have "client-driven concurrency".

we can discuss but we defend it on two grounds: these features are quite rare in real systems and they could make our goal infeasible.

1.2 Our Method

What does a special purpose formal method for distributed object-oriented systems look like? All formal methods will consist of at least the following two things: a language in which concepts can be expressed and a system for reasoning about expressions in the language. A formal method focused on distributed object-oriented systems should also provide ready-made object concepts and support behaviour consistent with a distributed computing environment.

Although our formal method is in an early stage of development, our research has lead to the development of two components, Oompa and sequential process specifications. Oompa is a language in which designs for distributed object-oriented systems can be expressed. Sequential process specifications allow abstract descriptions of behaviour to be specified and refined.

1.2.1 The Formalism: Oompa

We wish to provide a language in which designs for distributed object-oriented systems can be expressed. To minimise the semantic gap between the language and these systems, the language should possess many of the features of our object model: it should have uniquely identified objects with attributes and methods. It should support method invocation and object creation. Objects in the language should allow multiple concurrent invocations. Lastly, to describe real world systems, the language should be computationally powerful. Based on these requirements, we developed Oompa, an

object-oriented process algebra.

Oompa's syntax is based upon the π-calculus [Mil99], a computationally powerful process calculus which can describe systems of parallel processes engaged in highly dynamic channel-based communication. In Oompa, we extend the language of the π-calculus with primitives for creating objects, invoking methods and accessing and updating state. This gives a syntax for code which we embed in a syntax for methods, attributes, interfaces and classes. We use this syntax for statically defining Oompa programs.

Before discussing Oompa's semantics, we need to draw an important distinction between two aspects of certain types of formalisms. For many formal languages where an operational semantics is given, the semantics applies to a term in that language and describes the reduction of that term, perhaps in some context. This is true of the λ-calculus and the π-calculus, for example.

However, in languages where structures are defined in advance there is an important difference: here the operational semantics do not apply to the terms of the syntax of definitions but, rather, to the terms of some other syntax. For example, in a class-based object-oriented language, a program is defined in classes which do not themselves have behaviour — it is the objects created during the execution of the program which have the behaviour.

For this second type of language, we group the syntax of definitions and their associated theory under the heading "static systems". Those parts which participate in the operational semantics, we group under the slightly more awkward heading "dynamic systems"[5].

We provide Oompa with two operational semantics and, hence, Oompa

[5] The distinction can be also seen in CORBA, where a *construction model* describes how objects are implemented in code whereas an *execution model* describes the parts of the system which participate in computation.

has two dynamic systems. Each consists of a syntax for the "running" systems, some associated theory of equivalences and an operational semantics. Each offers a different way of modelling the behaviour of an Oompa program.

The first of Oompa's dynamic systems we call the agent-based dynamic system. This views a distributed object-oriented system as consisting of four parts: the static definitions of classes and interfaces, a global dictionary of type information, a global dictionary of state information and a set of running processes. We might call this a *flat* model of a distributed object-oriented system, as it ignores any structure due to distribution. The semantics of this dynamic system is a reduction relation that describes how the global structure evolves. The role of this dynamic system is to provide a simple and intuitive model of running Oompa programs. Unfortunately, the semantics is non-compositional in the sense that it only applies to complete closed systems. This makes reasoning awkward.

The second of Oompa's dynamic systems we call the configuration-based dynamic system. This models a distributed object-oriented system as a hierarchy of subsystems. As before, some information can be global but, this time, type and state information can also be local to subsystems. The semantics of this dynamic system is a labelled transition relation which can apply to subsystems as well as to the complete system. The labels on the transitions allow us to describe how a subsystem interacts with its environment. We can compose such descriptions of behaviour to describe the behaviour of a set of parallel subsystems and, hence, the semantics is compositional. Because of this, it will be our primary semantics for reasoning. Its disadvantage, when contrasted with the agent-based dynamic system, is its relative complexity.

Another of the components often defined for a formal language, neither part of its static system nor its dynamic system, is a type system. The

purpose of a type system is to avoid a set of undesirable behaviours that occur when a value is used in an inappropriate context. The exclusion of these errors gives us confidence in the meaningfulness of our programs. For Oompa we give a statically checkable type system. This means we can check Oompa's class and interface definitions for type safety. We prove, for both dynamic systems, that a running system derived from checked definitions will never give rise to type errors.

1.2.2 The Specification Language: Sequential Process Specifications

Oompa is a language for expressing actual designs of distributed object-oriented systems. To talk about whether such designs are, in some sense, "correct" we will need a way to reason about their behaviour. The labelled transition relation of the configuration-based dynamic system provides a very simple language for describing the behaviour of Oompa programs. Unfortunately, its expressions are limited to statements of the form "configuration A can perform action a and become configuration B".

To allow us to express more sophisticated statements of behaviour, we provide another language called sequential process specifications. We base this on a very simple language called sequential process expression, which forms the core of CCS [Mil80], and extend it with a non-deterministic choice operator similar to internal choice in CSP [Hoa84].

We use sequential process specifications as specifications of sequential process expressions and we provide a satisfaction relation which formalises this relationship. We can view sequential process expressions as describing the behaviour pattern of a given Oompa configuration. Therefore, we can view sequential process specifications as indirectly specifying Oompa config-

urations.

1.2.3 The Approach

Our current approach to system development operates as follows, although it should be noted that our formal method is still a work in progress. After requirements capture, a stepwise refinement in sequential process specifications is used specify the abstract behaviour we require of our system. The next development stage is reached with an "invent and verify" step, whereby an Oompa class (or classes) is proposed as giving rise to the specified behaviour. This claim is then formally verified using our implementation relation.

Although it was one of our stated goals, we have not yet evaluated the usefulness of our method for modelling distributed object-oriented systems.

1.3 Contributions

In this section, we briefly describe the key contributions of the thesis.

The formalism Oompa, which combines class-based object-orientation with concurrency, is our main contribution. It differs from other comparable languages in either the level of its object-orientation or its approach to concurrency. We discuss these differences further in Chapter 2.

Oompa's type system can be considered a minimal type system for languages with channels and objects since we use recursive types to handle both features. Our work on the type system also led to the development of an approach to making types independent of the definition set, called the expansion function. However, a similar function can be found in [AC91].

Our other main contribution is sequential process specifications. These allow specifications of abstract behaviour to be developed and, for this task,

represent an alternative to both CCS and CSP.

One further contribution is found in Appendix A. Our naming system requires a way to apply two renamings (kinds of permutation) simultaneously and without interference. We have devised a technique, called isolated sum, which allows this to be done.

1.4 How to Read this Thesis

This thesis is structured as eight chapters and four appendices. The chapters constitute the *body* of the thesis and their role is to introduce the material in an accessible manner. The role of the appendices is to accommodate the more technical work we have done which, if placed within the body, might unnecessarily encumber the presentation. In particular, many of the proofs of results we present are deferred until the appendices.

Given that many results in the thesis appear twice, once in the body and once, coupled with a proof, in the appendices, we need an approach to numbering that handles this duplication sensibly. Our approach is to give each occurrence of such a result a separate number so as to preserve the numbering sequences in both body and appendices. When a result is proved in an appendix, a small box in the margin will contain a reference to the result and page number of that occurrence. When a result in an appendix has a counterpart in the body of the thesis, it will also be given a reference back to that result and its page number. Results which have neither a proof nor a reference are considered straightforward and their proofs are not given.

like this

One significant consequence of the division of material between body and appendices is that there are two different ways to read the thesis. One approach is to read from start to finish. The presentation of the material

should flow smoothly and the more technical material will be deferred until the end. Alternatively, the body and the appendices can be read concurrently. This means that the justification for a result will be available when the result is stated.

1.5 Synopsis

After this introduction, Chapter 2 considers existing formal approaches to the problem. We discuss various methods, from traditional general purpose formal methods to special purpose object calculi.

Chapters 3, 4 and 5 present our formalism, Oompa. In Chapter 3 we define Oompa's static system. This involves the definition of its syntax and some discussion of the properties of that syntax, such as renaming issues. Chapter 4 defines Oompa's two dynamic systems: the agent-based dynamic system and the configuration-based dynamic system. Each dynamic system has a syntax, some theory and an operational semantics. Chapter 5 gives the definition of Oompa's type system.

We introduce our simple specification language in Chapter 6. We provide its syntax, a semantics based on a notion of satisfaction and a refinement relation between specifications. Chapter 7 indicates how we might use our formal method by showing the development a simple system. We provide our conclusions and speculate on future work in Chapter 8.

There are four Appendices. Appendix A defines a technical construction we call *isolated sum*. This is used throughout Appendix B to simplify proofs. In Appendix B, we provide the proofs of most of the results of chapters 3 and 4, which predominantly involve renaming and equivalence properties. Appendix C similarly provides the proofs for many of the results concerning

the type system of Chapter 5. Lastly, Appendix D defines the simulations used to justify satisfaction and refinement statements in chapters 6 and 7.

Chapter 2

The State of the Art

In order to clarify the contribution offered by our system, we need to place it in context. In this chapter, we present a survey of those formal methods with which one might tackle our problem. Such a survey cannot hope to be complete but we will attempt to touch on the key approaches available. An alternative but similarly motivated survey can be found in [BD01].

We divide the methods into three categories: state-based approaches, process calculi and concurrent object-oriented approaches[1]. The methods of the first category model a system in terms of how it transforms state. We consider two methods here: the Rely/Guarantee approach and action

[1] There are two other categories of methods which we might have included: "sequential object-oriented approaches" and "approaches for mobility". There are several methods of both kinds that would provide useful context for our work but would not be suitable approaches for tackling our problem (for example, the object calculi of [AC96] or the ambient calculus [CG98]). While object-orientation is something that could, arguably, be added as a sugaring to existing formalisms, the same cannot be done for concurrency. This means that sequential approaches would not offer a suitable way of tackling our target systems. On the other hand, given that we have excluded mobility from our target systems, using an explicitly mobile formalism would add needless complexity.

systems. The second category groups a set of special purpose formal methods, called process calculi, which focus on the communication events that occur in concurrent systems rather than on state change. Here we consider CCS, CSP, the π-calculus, Pict and the join calculus. The methods of the third category are concurrent approaches which are intrinsically object-oriented. We consider the following methods in this section: POOL, $\pi o\beta\lambda$, TyCO, concς, OC and ODAL.

Once the methods have been described, we will have an opportunity to discuss their suitability for our problem. We separate their presentation from their evaluation since, irrespective of their suitability for our problem, these methods provide the context within which our method lies.

The chapter is structured as follows. We consider state-based approaches in Section 2.1, process calculi in Section 2.2 and concurrent object-oriented approaches in Section 2.3. In Section 2.4, we evaluate how these methods offer an approach to our problem. Section 2.5 identifies the role we wish our formalism to play and contrasts it with the methods of the survey. We summarise the chapter in Section 2.6.

2.1 State-Based Approaches to Concurrency

State-based formal methods model the behaviour of a program in terms of how it transforms state. In this section, we consider extensions to some primarily sequential state-based approaches which enable them to handle concurrent systems.

2.1.1 The Rely/Guarantee Approach

In VDM[2] [Jon90], an operation is described in terms of how it transforms the system state. More precisely, an operation ensures that the system state will satisfy a predicate, the *post-condition*, as long as it is performed when the system state satisfies another predicate, the *pre-condition*. There are two particularly important advantages to this approach. Firstly, it is *compositional*. So the specifications of two operations can be naturally combined to give the specification of their sequential composition. Secondly, it supports *refinement*. So abstract descriptions of operations and data can be replaced by more concrete descriptions. This method successfully supports the development of sequential programs.

Operations performed in parallel can potentially *interfere* with each other. Unfortunately, in the presence of interference, this method fails to be fully compositional and refinement becomes problematic. There is no general way of combining the specifications of operations, defined in terms of pre- and post-conditions, to give the specification of the operations' concurrent composition. Consequently, the method is no longer fully compositional. Refinement also encounters difficulties: an operation refined into several sequentially or concurrently composed operations may not satisfy its original specification in a concurrent context.

An extension to this approach [Jon81, XdRH97] allows it to take account of interference by augmenting specifications with two more predicates. A *rely-condition* describes the assumptions an operation needs to make about

[2] The description of VDM we give approximately fits two other well-known state-based methods: Z [Spi92] and the B method [Abr96]. In fact, the extension we describe, although defined specifically for the VDM, should be applicable to any system which defines operations in terms of pre- and post-conditions.

its concurrently executing environment. A *guarantee-condition* describes assumptions that the environment can make about the effect of the operation on concurrent operations. Using this technique, the method can function as before but with a significantly higher burden on the development — a set of *coexistence* proof obligations must be discharged. Some relief can be obtained by delimiting the dependencies and effect of an operation with syntactic markings such as "read-only" and "private" [CJ95].

2.1.2 The Refinement Calculus and Action Systems

Another formal method concerned with how an operation transforms system state is the refinement calculus [BvW98]. Operations, both program and specification statements, are modelled by *predicate transformers*, which are functions from sets of terminating states to sets of initial states[3]. When a predicate transformer models an operation, the initial states it returns are just those in which the operation must start in order for it to reach one of the given terminating states. The justification for using predicate transformers comes from viewing programs as contracts: "Very simple contracts with no choices (deterministic programs) can be modelled as functions from states to states. More complicated contracts may be modelled as relations on states, provided that all choices are made by only one agent. General contracts, where choices can be made by two or more agents, are modelled as predicate transformers" [BvW98]. As above, the key methodology is refinement from abstract specifications to concrete programs.

The refinement calculus manages concurrency via action systems [Bac90]. An *action system* consists of a set of atomic operations, called *actions*, which

[3]When thought of as predicates, the sets of terminating and initial states correspond to post-conditions and pre-conditions respectively.

can be run in parallel and can access global state. The behaviour of an action is described using a predicate transformer behind a boolean guard. Actions need not always be enabled but, when enabled, they can be executed arbitrarily with a single restriction: only those actions which access disjoint parts of the system state are allowed to run concurrently.

The most significant assumption is that the effect of an action is atomic. If input-output behaviour is the only concern, it means that an action system is equivalent to a sequential system where actions are non-deterministically interleaved. We can model these in the sequential refinement calculus using an iterated choice statement. For action systems where reactive behaviour is also important, the refinement calculus must be extended. *Simulation refinement* [Bac90] and *trace refinement* [Bac92] are two relations which can establish refinement between action systems in these cases.

The action system approach has been used to give a model for concurrent objects [BBS97, BKS97, BS99]. In all three approaches, objects have attributes, methods and an associated action system. In [BBS97] a object-oriented language called Action-Oberon (an extension of the Oberon 2 language) is defined. Here, object-orientation is introduced by making action systems *type-bound*, i.e. when a value of a certain type is created, the associated type-bound action system is also created. The focus is on the input-output behaviour of parallel programs so, as above, an Action-Oberon program can be translated down to the sequential refinement calculus. [BKS97] defines OO-action systems. These have class-based object-orientation where objects are special kinds of action systems. Refinement rules between classes are considered and justified by translating OO-action systems into ordinary action systems. Finally, [BS99] defines a model of object-orientation similar to OO-action systems but with the addition of an inheritance mechanism. A

notion of *atomicity refinement* is introduced by allowing early returns.

2.2 Process Calculi

Process calculi are formalisms which focus on the behaviour of a system in terms of the communications that occur rather than on state change. The decision of what to view as a communication is not clear cut and, consequently, many process calculi have emerged. Typically, a process calculus will have a syntax of *processes* which describe behaviour entirely in terms of abstract communication events called *actions*. An intuitive view of their semantics is as "an unstructured collection of autonomous agents communicating in arbitrary patterns over channels" [PT95] — a concept sometimes called *process soup*.

Process calculi are often referred to as process algebras. Milner [Mil89] suggests that the former term might be preferred (in his case for CCS but it applies to most) as it suggests the use of mathematical tools beyond algebra.

2.2.1 CCS

An action in CCS (the Calculus of Communicating Systems) [Mil80] models an indivisible synchronous communication between exactly two participants. This turns out to be sufficiently general to allow systems to be modelled at various levels of abstraction. Beyond its syntax for performing actions, CCS has primitives which describe parallel composition, choice between actions and scope restriction. According to Milner, "there is nothing canonical about the choice of the basic combinators, even though they were chosen with great attention to economy. What characterises our calculus is not the exact choice of combinators, but rather the choice of interpretation and of mathematical

framework" [Mil89]. The primary semantics is a labelled transition system over which a notion of equivalence called *bisimulation* is defined.

Bisimulation is a coinductive reasoning technique which distinguishes two processes when a third process communicating with them could find some difference in how they behave. Two particularly important bisimulations are given for CCS: *strong* and *weak bisimulation* (the latter is also called *observational equivalence*). As well as these semantic views of equivalence, a system of equational reasoning is defined over processes which can establish bisimulation and is complete on finite-state processes. Milner also defines a formal logic, $\mathcal{PL}$ (process logic), whose formulae can be used to express properties of processes. This allows an incremental method of specification but can guarantee only partial correctness.

2.2.2 CSP

Hoare's CSP (Communicating Sequential Processes) [Hoa84, Ros98] is a process calculus whose actions are indivisible and synchronous but, unlike CCS, may involve many participants. Furthermore, CSP's syntax is richer than that of CCS and has constructs which allow processes to be specifications of behaviour — for example, the synchronous parallel operator and internal choice. The primary semantics of CSP is not an operational semantics (although it has one) but three denotational semantics in terms of sets: the *traces model*, the *failures model* and the *failures and divergences model*. As well as these, the algebra for equational reasoning in CSP can be viewed as an axiomatic semantics.

One of the key advantages for using CSP in system development is its fundamental support for refinement. Reasoning in the models is possible, for example by specifying a system property in terms of traces, but [Ros98] em-

phasises "process-based specification", i.e. refining from a non-deterministic process expression. Several industrial strength tools exist which can model check [For00a] and animate [For00b] CSP specifications.

2.2.3 π-Calculus

CCS and CSP share the property that the systems they describe have a largely static communication network[4]. This can become a problem when modelling or developing modern systems which can dynamically change their communication network. The π-calculus [Mil99, SW01] overcomes this limitation by allowing processes to generate new names which can be sent as values and then used as communication channels[5]. With a very spartan syntax, it succeeds in being both elegant and computationally complete — a π-calculus encoding of the λ-calculus is given in [MPW89a].

The π-calculus is a descendent of CCS and inherits much of its semantic theory. The primary semantics for the π-calculus are operational: a reduction relation and a labelled transition system [MPW89b]. The notion of

[4]In fact, in both CCS and CSP, the communication network can change in a limited respect. Connections can be lost and processes can split into two newly communicating parts. However, for a truly dynamic network, it must be possible for new connections to arise between previously separate subsystems. In CCS, this cannot happen since channel names cannot be communicated. In CSP, the model of communication does not allow *new* names to be received.

[5]The facility for new communication possibilities to arise during execution has been called *mobility* but it is arguable whether the π-calculus is suitable for modelling mobile systems. An toy example of a mobile phone system is given in [Mil91], where a mobile phone switches base as it travels into a different region. However, since the π-calculus has no concept of location or domain, the regions are completely outside the model. Examples of languages with a true notion of mobility would be the ambient calculus [CG98] or the programming language Obliq [Car94].

equivalence used for the reduction relation is *barbed bisimulation* which uses a predicate (called a *barb*) to detect whether communication is possible on a given channel name. For the labelled transition system, strong and weak bisimulation are defined, as in CCS. Several other notions of equivalence between π-calculus terms are considered in [SW01] and an algebraic theory has also been developed [PS95].

The π-calculus is a name-passing calculus and names are a primary feature of object-orientation. For this reason, and others, the π-calculus has been used several times as a model of object-orientation [Mil99, SW01, Wal95, San96, KS98]. The π-calculus has also been used to give concurrent object-oriented language a semantics in [Jon93b] and [Öve00]. Moreover, encodings of concurrent objects have been given in extensions of the π-calculus, for example the form calculus [SL00] and Pict [PT95, SL96] (see the next section).

Many variants of the π-calculus have been proposed; three interesting cases are Aπ, HOπ and Dπ. In the asynchronous π-calculus [HT91, SW01], Aπ, two simple restrictions on the syntax of the π-calculus give a model of concurrent systems where communication is asynchronous. Firstly, a process cannot follow a send operation with any other operation and, secondly, a send operation may not occur in a summation. In the higher-order π-calculus [San92], HOπ, abstractions (processes which require parameters) may be the objects of a communication. In fact, this extension does not provide extra power; HOπ can be naturally encoded in the π-calculus by communicating access to processes rather than the processes themselves. Distributed π-calculus [RH98], Dπ, places π-calculus-like terms at locations and introduces a notion of location failure. This gives a more accurate model of systems where distribution is important.

2.2.4 Pict

Pict [PT98] is a formally defined programming language which rests upon the π-calculus much in the way a functional language rests upon the λ-calculus. Specifically, its core language is asynchronous π-calculus without summation but augmented with records and pattern matching. Upon this small core language more sophisticated programming constructs are built as encodings. “The goal in Pict is to identify and support idioms that arise naturally when [π-calculus] primitives are used to build working programs — idioms such as basic data structures, protocols for returning results, higher-order programming, selective communication and concurrent objects” [Pie98]. So the focus in Pict is support for programming over support for reasoning. Nevertheless, it is likely that much of the theory of the π-calculus carries over to Pict.

Pict has been used as an “experimental testbed” to explore different ways of modelling objects: various options are considered in [PT95] and [SL96]. Typically, an object will be encoded as a record of channels — each channel provides access to a process managing a single service. Although π-calculus' summation (i.e. choice) is not present in the core language, it can be introduced as a sugaring and is available to users as a library module. Using choice, Pict could also support the object encodings for the π-calculus that we mentioned in the previous section.

2.2.5 Join Calculus

The join calculus [FG02] is a name passing process calculus designed for distributed and mobile programming. The syntax can be seen either as an asynchronous π-calculus extended with pattern matching or as a small (ML-like) functional programming language extended with concurrency and message

passing. The intention is to provide a language which models distributed programming[6].

The processes of the join calculus are hierarchically structured into sites (locations) and communication is by asynchronous channel-based message passing. Contrasted with more typical process calculi, there are two interesting features of the join calculus communication model. Firstly, channels are "defined" at a single site; all messages sent on that channel get delivered to that one site. Consequently, all processes waiting to receive on the same channel must reside at the same site. Secondly, processes can consume several messages atomically. This is expressed, syntactically, with a *join pattern*. Until all the messages a process wishes to receive have arrived, and that process consumes them, those messages remain available for other processes. From the perspective of modelling and implementation, the significant consequence of this is that any contention between processes can be resolved locally.

The join pattern is sufficiently expressive to allow many standard programming idioms to be naturally encoded. "The join calculus gives us a general language for writing synchronisation devices; devices are just common idioms in that language. This makes it much easier to turn from one type of device to another, to use several kind of devices, or even combine devices" [FG02]. Encodings of asynchronous and synchronous π-calculus channels, actors, objects and Ada rendez-vous are given. Although the consideration of object encodings is brief, several different styles of concurrent

[6] It is also worth mentioning the distributed join calculus [FGL$^+$96] which extends the join calculus with an explicit notion of location and a primitive for process migration. The intention here is to provide a model and basis for *mobile* programming. An extension of the Objective CAML programming language, JoCAML [FFMS01], embodies this model of distribution and mobility.

objects are considered.

The join calculus is given a semantics in terms of a computational model called the Reflective Chemical Abstract Machine. It defines an operational semantics based on three relations between processes: reduction steps, heating steps and cooling steps. This model relies on the metaphor of computational systems as chemical solutions, as in the Chemical Abstract Machine [BB90]. The join calculus is intended to be the basis of a programming language so an important advantage of this computation model is that it can be refined into an efficient implementation. A labelled semantics is defined with the Open Reflective Chemical Abstract Machine which allows us to consider the interactions a process may have with its environment. A simple and intuitive semantics, based on an equivalence and a reduction relation, is also defined.

For reasoning about the join calculus, no one notion of program equivalence will satisfy all requirements: "finding the 'right' equivalence for concurrent programs is a tall order" [FG02]. Consequently, a hierarchy of five equivalences is defined[7], each striking a different balance between precision and the richness of the generated identities. In increasing order of precision, they are: may testing, fair testing, coupled-similarity equivalence, bisimilarity equivalence and labelled bisimilarities.

2.3 Concurrent Object-Oriented Approaches

We now consider some concurrent formalisms which are intrinsically object-oriented. The richer of these formalisms are full object-oriented languages; the others are *object calculi*.

[7] In fact, the equivalences and techniques of the hierarchy are not restricted to the join calculus. "In principle, they can be applied to any calculus with a small-step reduction-based semantics, evaluation contexts, and some notion of observation" [FG02].

2.3.1 POOL

The first we consider is the POOL (Parallel Object-Oriented Language) family of programming languages due to Pierre America [Ame87, AdBKR89, Ame89, Ame91]. This family includes POOL, POOL-I, POOL-S, POOL-T, POOL2 and a sequential version called SPOOL. For brevity, we will consider this collection of languages as if it was a single language so some of the comments need not be thought to apply to all of them. POOL "aims at the systematic construction of reliable and maintainable software" [Ame87]. America views object-orientation primarily as a protection mechanism; he "considers [the] principle of protection of objects against each other as the basic and essential characteristic of object-oriented programming" [Ame91]. Consequently, state is private to an object and can only be accessed by the object's methods.

One novel feature of POOL is the way in which concurrency is achieved. POOL is class based and each class defines for its objects a section of code called a *body*. At object creation, the code in the body is started in parallel with the code already running in the system. The object is allowed only a single process so the body must, at some time, execute an *answer* statement in order to service requests on the object. This answer statement can choose to service certain requests only; those requests which are not serviced are queued.

An important contribution of POOL is the complete separation of inheritance from subtyping[8]. Most object-oriented languages conflate the two

[8] In fact, subtyping in POOL is more than just interface conformance — methods and classes have an attached *property* which asserts that the tagged entity satisfies a certain specification. Unsurprisingly, the checking of code against its specification is not automated but, if the tags are believed to be correct, then it can be used to make the subtyping relationship more discriminating. For example, in many object-oriented languages

ideas, which can be useful since they often coincide. On the other hand, by insisting that they occur together, desirable inheritance relationships can be frustrated and undesirable subtyping relationships established [Ame87]. POOL decouples the two concepts and, along with its subtyping interface, a class can export an interface specifically to let other classes inherit from it.

Several semantics have been provided for POOL but the two significant ones are a denotational semantics in terms of complete metric spaces [AdBKR89] and an operational semantics in terms of an unlabelled transition relation [Ame89]. Walker [Wal95] gives a translation into the π-calculus and, in doing so, gives POOL a new operational semantics in terms of a labelled transition relation.

2.3.2 $\pi o\beta\lambda$

In [Jon93a], Jones came to see "language restrictions as a way of taming interference". $\pi o\beta\lambda$ [Jon92] is not intended to be a full programming language but rather as a design notation which can be used for the development of concurrent programs in an actual implementation language. It has class-based object-orientation and is "heavily influenced" by POOL. One similarity between POOL and $\pi o\beta\lambda$ is the restriction that only one method can be active in each object at any one time[9]. One difference is how parallelism arises in a $\pi o\beta\lambda$ program. Rather than POOL's bodies, $\pi o\beta\lambda$ has synchronous invo-

Bag, Queue and Stack would be interchangeable since they all satisfy the same simple container interface. In POOL, Queue and Stack fail to be subtypes of each other because Queue has the property FIFO (first in, first out) but Stack has the property LIFO (last in, first out). Since Bag doesn't specify an ordering policy, both Stack and Queue can subtype from it.

[9] $\pi o\beta\lambda$ does have a parallel statement but it is not made clear whether it can be used to override this restriction.

cations whose receiver may issue a reply, yet continue working (a technique called *early returns*).

The development method associated with $\pi o\beta\lambda$ proposes sequential development followed by the introduction of concurrency using rules which can commute code statements under certain conditions. These rules are justified by arguments in the semantics [Jon94]. Reasoning requires a mixture of pre-conditions, post-conditions and invariants together with assumptions about the interference of the environment using a version of rely-condition. In the development of certain programs, arguments about the object graph (which records the interconnection of object references) can be used to make the reasoning easier. Another aid is marking state variables in terms of whether they can be read or written by the environment [Jon93c].

Although $\pi o\beta\lambda$ is a design notation, it still needs a semantics to fix the meaning of its terms. The primary semantics is a mapping into the π-calculus [Jon93b]: "It would be possible to argue that this semantics is [...] giving too fine a level of granularity". So the behaviour of an encoded $\pi o\beta\lambda$ construct may take several steps to accomplish what might be thought to be an atomic $\pi o\beta\lambda$ operation and, moreover, this stuttering may lead to two parallel operations overlapping or enclosing each other. However, "the π-calculus has an algebra which makes it easy to reason about equivalence of processes".

2.3.3 TyCO

Another object-oriented formalism with concurrency is Vasconcelos' Typed Concurrent Objects (TyCO) [Vas94a]. The motivation for TyCO is to "capture basic features present in most notions of objects and to give precise (operational) semantics and type inference systems to object-oriented con-

current programming languages" [Vas94b]. The name TyCO is used both for a calculus and a programming language [VB98] built on top of the calculus, using sugarings. TyCO was originally presented as an object calculus[10] but later it was described as "a form of the asynchronous π-calculus featuring first class objects, asynchronous messages and template definitions" [LSV99].

Objects in TyCO are, in general, ephemeral, i.e. invocation acts like a destructor[11] of the object. We can create persistent objects by using recursion but we can also create objects which change behaviour and even change interface. So, although class-like definitions can be introduced as a sugaring, the system allows for much more flexible object behaviour than a class based language would typically allow.

Aside from its work on the foundations of object-orientation, TyCO is being put to one other interesting use: at the University of Lisbon, students are using TyCO as training in concurrent object-oriented programming [Vas01].

2.3.4 concς

In [AC96], Abadi and Cardelli present several sequential object calculi which are "at the same level of abstraction as the λ-calculus". Instead of function application, their computational power comes from method invocation and method update. An object is a labelled record of methods and a method

[10] This change in presentation is associated with an improved syntax that makes the relationship with the π-calculus clearer. First, augment the π-calculus' send primitive with a label, e.g. l in $x!l[\tilde{e}]$, and its receive primitive with a labelled record, as in $x?\{l_1(\tilde{e_1}) = P_1, \ldots l_k(\tilde{e_k}) = P_k\}$. Then, with an appropriate rule in the operational semantics, we do indeed get object-like behaviour.

[11] By *destructor* we mean in the sense of constructed data types, not to be confused with a distinguished method that manages object deletion in certain object-oriented programming languages.

can contain a value which, upon invocation, is substituted by the containing object (by value or reference depending on the object calculus). The presence of method update make their calculi somewhat unusual since few object-oriented languages support this behaviour. Nevertheless, method update is simple to formalise and it allows them to illustrate various forms of reuse.

The most primitive language they present is the ς-calculus, a tiny functional object-based language. This is already sophisticated enough to encode the λ-calculus. Other languages considered include impς, a similarly foundational imperative object calculus, and three programming languages, called O-1, O-2 and O-3. Although predominantly object-based, class structures for some of their languages are given. The main contribution of the study is the development of type systems for various object-oriented features.

Gordon and Hankin have taken the imperative object calculus impς and given it a concurrent form concς [GH98]. As in its parent object calculus, objects in concς are labelled records which support method update and invocation, where self-substitution is by reference. Here, however, object references can have local scope and in this regard concς resembles TyCO.

Concurrency is achieved by adding parallel composition and a particularly interesting feature of this system is that this operator has has a non-symmetric form, $(a \upharpoonright b)$, which can evaluate to a result. The operational semantics of concς is an unlabelled transition relation which describes how objects and processes invoking, updating methods or cloning interact.

The distributed object calculus, an extension of concς with explicit distribution and mobility, is presented in [Jef00].

2.3.5 OC

In [Nie92], Nierstrasz proposes OC, an object calculus which is “a merge of the π and λ calculi” extended with a prioritised choice operator and pattern matching[12]. Communication has two forms: local communication, which resembles function application, and remote communication, similar to that of a typical process calculus. Interestingly, the function composition that is available enables a kind of inheritance where an object can be built out of more primitive objects. Two features of OC’s object orientation are that multiple requests on an object can be serviced simultaneously and objects need not have a fixed set of requests which they are willing to service.

In fact, OC’s object-orientation has a problem: the pattern matching can violate strict object encapsulation. This has lead Threet, Hale and Shenoi to develop a modified version of the calculus called Robust Object Calculus, or ROC [THS96]. In ROC, values can be marked as unbindable. A wild card in a pattern cannot match an unbindable value and, used appropriately, this guarantees full object encapsulation.

2.3.6 BOOM and ODAL

The BOOM framework [Öve00] is a meta-modelling approach whose aim is to provide object-oriented modelling languages, in particular UML [OMG01], with precise definition. Because the intended users will be familiar with object technology, the framework itself uses a concurrent object-oriented language, called ODAL, as its specification language. Our work is not a meta-modelling exercise so we consider ODAL, rather than the BOOM framework, to be of interest here.

[12] Unfortunately, the name OC has also been used to refer to Abadi and Cardelli’s object calculi, for example in [San96].

ODAL is a relatively rich language when compared to some other object-oriented formalisms. It has classes with inheritance, assertions, invariants, dynamic type support and a strong static type system. It is given a semantics primarily by a translation into the π-calculus which, given the richness of the language, is predictably complex. Of course, the primary purpose of BOOM and ODAL is to give concepts a standard meaning, not to reason about their behaviour, so the complexity of this semantics is not a particular problem for their work. Fortunately, a simpler operational semantics in terms of a labelled transition system is also given.

2.4 Evaluating the Methods

Our intention is to provide a special-purpose formal method to support the modelling and development of distributed object-oriented systems. In this section, we discuss how the approaches we have considered could be used to this end.

2.4.1 State-Based Methods

In the first category, we considered state-based methods:

> "The state-based paradigm is characterised by the explicit specification of states and the implicit specification of system behaviour [...]. This is in contrast to process algebra approaches, where the behaviour is explicit and states are implicit." [Smi01].

For our purposes, we feel that behaviour should be given primacy over state. The distributed object-oriented systems of our problem domain are not just state-based systems where some concurrency occurs; they are *primarily* concurrent systems. More significantly, these systems are interactive systems:

we experience them by communicating with them, not by inspecting their current and possibly distributed state.

Other aspects of our target systems suggest that a state-based approach is not a good fit. We would, for example, need to provide data structures to explicitly encode the life-cycle of objects and the network of object references. Although these notions can be added, this represents a clear semantic gap.

Considering the Rely/Guarantee approach first, we can identify two ways in which it is unsuitable. Firstly, there is a lack of object concepts, necessitating an encoding. Secondly, the rely/guarantee approach can generate huge proof obligations in a system where there is a lot of concurrency. In theory, every possible combination of concurrently executing operations must have its predicates checked to ensure that those operations can execute together safely.

In the Refinement Calculus we are provided with a model of concurrent objects. Unfortunately, we find that its computational model is not appropriate for our target systems. Within this model, the methods of an object execute atomically. However, the methods of the objects of our target systems are not just atomic transitions between states but involve communication events and take time.

2.4.2 Process Calculi

The process calculi we considered are designed specifically to tackle concurrent systems. They emphasise behaviour over state and we believe that this is appropriate for the systems of our problem domain. The general drawback of these methods, given our stated objective, is the lack of object concepts.

Encodings for objects are defined in many of the process calculi we discussed. This means that many properties of objects, such as encapsulation,

would exist only algorithmically in the encoding. Another possible issue with encodings is that an object-level operation might be encoded as a sequence of several lower-level actions. This allows the possibility of behaviours, such as strange interleavings, which are not appropriate in the object model. To discount such behaviours requires extra reasoning.

Considering CCS and CSP, we find that their static communication networks are too limiting. The interconnections in our target systems are likely to change significantly over time and CCS and CSP cannot accurately model this kind of system. Although aspects of high-level application logic or low-level system protocols might be usefully modelled using these methods, we do not believe they offer a general approach to our problem.

In contrast, the π-calculus, PICT and the join calculus all support dynamic communication networks. Moreover, their rich communication actions could offer interesting ways of modelling our target system. The only factor that counts against their use is that object-orientation is available only indirectly.

2.4.3 Concurrent Object-Oriented Approaches

Those methods of the third category seem close to our intended aim. They are formally defined, support concurrency and all possess a set of object concepts. Given that we are proposing an alternate formalism, we must respond to the important question of whether it is needed: are none of the formalisms and languages of the third category suitable for our purposes? There are two ways these methods can be a bad fit for our target systems: The first way is that their model of concurrency can be inappropriate. The second way is that their object model can be a poor match to ours.

POOL and $\pi o\beta\lambda$ seem, in many respects, ideally suited to our needs.

Unfortunately, their model of concurrency is not quite right. In both these languages, only one of an object's methods can be active at one time[13]. Given this restriction, there would be no way to model our target systems' objects at the right granularity.

TyCO, concς and OC, on the other hand, all support objects with arbitrary concurrency. Unfortunately, we feel that they are a little too foundational for our purposes. They are intended to be minimal models of object-oriented computation and have very economical syntaxes. We believe, however, that such foundational languages are more suited to experimentation than a practical formal method. As is often the case, it would be possible to build suitable high-level constructs in these languages via encodings.

ODAL allows concurrency within objects and is richer than TyCO, concς and OC. The problem with using ODAL for our purposes is the fact that its semantics is based on an encoding. Reasoning with such a semantics is likely to be highly complex.

2.5 Our Work in Context

Our approach seeks to provide a special-purpose formal method to support the modelling and development of distributed object-oriented systems. We intend there to be a narrow semantic gap between our method and these target systems. In this light, our decision to use a formally-defined concurrent object-oriented language becomes clear. So our formalism, Oompa, belongs to the third category.

The significant difference between Oompa and other methods we have

[13]An object with this style of concurrency can be called a *monitor* or a *synchronised object* [FG02]. We will occasionally refer to one as a *single-threaded* object.

discussed is that Oompa has been designed to be a close fit to the target systems we are considering. Firstly, the model of concurrency matches that of the target systems: there is no limit on the number of concurrent processes running at an object. Restrictions can be built within Oompa but none are imposed by Oompa. Secondly, Oompa's objects operate at the right level to model the target systems. Invocations can send and receive several values and the language of code is relatively rich: the methods of an Oompa object can participate in channel based communication as well as object style behaviour.

There is one other aspect of Oompa's design which, we believe, make it especially suited to our work. Oompa's semantics are operational semantics which directly embody our intuitions about object behaviour. They do not depend on encodings or sugarings which means that we can develop intuitions about behaviours and create idioms that are appropriate to our target systems. In doing so, we need not worry about avoiding behaviours which are expressible in the language but meaningless as real systems.

2.6 Summary

This chapter set out to examine various methods that exist in the literature and establish a useful context for our work.

The first type of methods we considered were state-based methods which make system state explicit and leave system behaviour implicit. We believe the behavioural aspects of our target systems are the more important and we discounted these methods for this reason. Next, we discussed process calculi which model systems in terms of the communication actions they can perform. These were not directly object-oriented so there is a semantic gap between them and our target systems. Lastly, we discussed concurrent

object-oriented approaches. These have concurrency, modelled in terms of communications, and are intrinsically object-oriented. With regard to tackling our problem, these methods had either an inappropriate concurrency model, an inappropriate object model or an overly complex semantics.

We concluded the chapter by considering our own formalism. We identified it as belonging to the third category of methods. Its advantage is that it has been designed specifically to be a close fit to the systems of our problem domain. Our next task is to provide its formal definition.

Chapter 3

Static Oompa

In this chapter, we begin the presentation of our formal language, Oompa. As a consequence of being a class-based language, there is a separation between the structures which define Oompa programs and those that participate in the operational semantics. This chapter defines the former: static Oompa.

In Section 3.1, we provide the formal definition of Oompa's syntax and give an example of an Oompa class. We discuss some of the less typical features of Oompa's syntax in Section 3.2. In Section 3.3, we describe the well-formedness conditions that an Oompa program is required to satisfy. Section 3.4 describes how names operate in Oompa and how Oompa programs undergo changes of names. In Section 3.5, we provide an equivalence system which identifies code that differs only in the choice of bound names. We summarise the contents of the chapter in Section 3.6.

3.1 Syntax

In this section we give the formal definition of the syntax of Oompa's static system. We present the syntax in a style of BNF frequently used in the

process calculus literature. This permits a trick where a meta-variable representing an element of syntax is allowed to stand for its syntactic class.

3.1.1 The Syntax of the π-Calculus

The core of Oompa's syntax is based on the polyadic π-calculus. We provide here a brief overview of the syntax of that language[1].

We will presuppose an infinite set, $\mathcal{N}$, of names, $a, b, c, \ldots \in \mathcal{N}$. The two syntactic classes of the untyped polyadic π-calculus are *prefixes* and *processes*:

$$\pi \quad ::= \quad c!\langle a_1, \ldots a_n \rangle \quad | \quad c?(b_1, \ldots b_n) \quad | \quad \tau$$

$$P \quad ::= \quad \textstyle\sum_{i \in I} \pi_i.P_i \quad | \quad (P_1 \mid P_2) \quad | \quad \text{new } c\ P \quad | \quad *P$$

Prefixes represent the communication actions that a π-calculus process can perform. The first type of prefix is *send*, $c!\langle a_1, \ldots a_n \rangle$, where a name c is used as a channel to send a tuple of names $a_1, \ldots a_n$. Next is *receive*, $c?(b_1, \ldots b_n)$, where a name c is used as a channel to receive some tuple of names, for which $b_1, \ldots b_n$ stand as parameters. Lastly, τ is the *silent action* which represents an internal private action within a process.

A process can be in one of four forms. The first form is *summation*, $\sum_{i \in I} \pi_i.P_i$, which describes a process willing to do any of the actions π_i — if it actually performs action π_j then its subsequent behaviour is described by P_j. The second form is *composition*, $(P_1 \mid P_2)$, which describes a process consisting of two processes in parallel. Next, *restriction*, new c P, restricts the scope of the name c to that process. Lastly, *replication*, $*P$, represents infinitely many copies of P in parallel.

[1]There are variations in the notations used for the π-calculus. For the syntactic forms which we use in Oompa, we choose those variants which can be written in ASCII characters.

An example of a π-calculus process is:

$$(c!\langle y\rangle.k?(r).0) \mid (c?(i).k!\langle c\rangle.0 + k!\langle y\rangle.0)$$

3.1.2 Names

As in the π-calculus, names, drawn from some infinite set, $\mathcal{N}$, play a vital role in Oompa. A name will often be written as n. Sometimes, however, we will know that a given name denotes a specific semantic entity, such as a channel, and in these cases we will use more suggestive meta-names. In particular, *parameters* and *channel*, *object* and *attribute names* are, syntactically, just names but we will typically write them r, c, o and a respectively. An example of a name would be `marmalade`.

There may also be sets containing *literal expressions*, $P_0, \ldots P_n$. These must be syntactically distinct from names and from each other. They will typically be written ℓ. Examples of literal expressions would be `144` and `'h'`.

A *value* is either a name or a literal:

$$v \ ::= \ n \ \mid \ \ell$$

We can ask for the *free names* of a value.

$$\mathrm{FN}(n) = \{n\} \qquad\qquad \mathrm{FN}(\ell) = \emptyset$$

There are two *reserved names*: `return`, which is a special channel name, and `this`, which is a special object name.

3.1.3 Code

The following table gives the syntax of *code*. The first five forms are borrowed from the π-calculus, the next four support Oompa's object-orientation.

p	$::=$	`end`	stop
	$\mid$	`fork`$\{p_0\}\ p_1$	fork
	$\mid$	`new` $c\colon \mathrm{ChT}\ p$	create a new channel
	$\mid$	$c!\langle v_1, \ldots v_n\rangle\ p$	send on a channel
	$\mid$	$c?(r_1\colon T_1, \ldots r_n\colon T_n)\ p$	receive on a channel
	$\mid$	`create` $o\colon \mathrm{ClT}\ p$	create an object
	$\mid$	$o.m!\langle v_1, \ldots v_n\rangle?(r_1, \ldots r_{n'})\ p$	invoke a method
	$\mid$	$a!v\ p$	update an attribute
	$\mid$	$a?r\ p$	access an attribute

where $T_1, \ldots T_n$, ChT and ClT are types whose syntax is given in Section 3.1.5.

It is easy to see the influence of π-calculus on the syntax of Oompa's code — the first five primitives above all have analogues in the π-calculus (`end` corresponds to the empty sum). Note, however, that the behaviour of their forms in the π-calculus is not quite the same as their behaviour in Oompa. One important difference in behaviour is that, in Oompa, only the first primitive in a piece of Oompa code can be executed. This is not the case in the π-calculus, where reductions can occur within a term[2].

Two primitives of the π-calculus which are not present in Oompa are replication and summation. Replication can be easily encoded in Oompa using method invocation. Summation, on the other hand, is not directly encodable in Oompa. We choose not to support it because, for distributed

[2] In the π-calculus, reductions can occur in the components of a parallel expression or underneath a restriction.

systems, we believe it to be too strong an operation. Summation allows contention between distributed communications to be resolved instantaneously. This is unlikely to be a feature of the systems we wish to design or model, so it would be inappropriate for our language to support it. One consequence of not having summation is that Oompa does not need a distinct syntactic class of prefixes. In the π-calculus, this separation is essential to ensure summations are well-formed.

We now provide some intuition for the primitives. Naturally, their behaviour is formally defined by the semantics, which is given in Chapter 4.

The first two primitives are essentially structural. The role of `end` is simply to terminate pieces of Oompa code — all other primitives take a continuation. The `fork` primitive causes its contained code to be run in parallel with its continuation[3]. Syntactically, a piece of code can be viewed as a binary tree whose nodes are labelled with primitives. Leaves are labelled `end` and branching occurs beneath nodes labelled `fork`.

The `new` primitive creates a new channel of the given type. The parameter c can be used in the continuation to refer to this new channel. The sending and receiving forms, $c!\langle v_1, \ldots v_n \rangle$ and $c?(r_1 \colon T_1, \ldots r_n \colon T_n)$ respectively, behave like their corresponding forms in the π-calculus, sending and receiving values on the provided channel name. When the `create` primitive is executed, a new object of class ClT is created. Occurrences of the parameter o can be used in the continuation to refer to this new object.

The invocation primitive executes in two phases. The first phase invokes the method m of the object o, sending it the tuple of values $v_1, \ldots v_n$. Also in the first phase, a new channel name is created, say x, and the invocation

[3]Although `fork` is asymmetric, this is not semantically significant. The asymmetric form makes `fork` syntactically consistent with the other Oompa primitives.

primitive becomes a receive on that channel of the form $x?(r_1{:}\,T_1, \ldots r_{n'}{:}\,T_{n'})$. In the second phase, channel-based communication is used to return values from the invoked method to the continuation along x.

The attribute update primitive sets the current value associated with the attribute a of the containing object to be v. When the attribute access primitive is executed, the current value of the attribute a is substituted in place of the parameter r in the continuation.

Several of Oompa's primitives, namely channel creation, receive, object creation, invocation and attribute access, introduce local bound names. We can find the free names in a piece of code using the following function.

$$
\begin{array}{lcl}
\mathrm{FN}(\texttt{end}) & = & \emptyset \\
\mathrm{FN}(\texttt{fork}\{p_0\}\ p_1) & = & \mathrm{FN}(p) \cup \mathrm{FN}(p_1) \\
\mathrm{FN}(\texttt{new}\ c{:}\,\mathrm{ChT}\ p) & = & \mathrm{FN}(p) \setminus \{c\} \\
\mathrm{FN}(c!\langle v_1, \ldots v_n\rangle\ p) & = & \left(\begin{array}{l} \mathrm{FN}(p)\ \cup\ \{c\} \cup \\ \mathrm{FN}(v_1) \cup \ldots \cup \mathrm{FN}(v_n) \end{array}\right) \\
\mathrm{FN}(c?(r_1{:}\,T_1, \ldots r_n{:}\,T_n)\ p) & = & (\mathrm{FN}(p) \setminus \{r_1, \ldots r_n\}) \cup \{c\} \\
\mathrm{FN}(\texttt{create}\ o{:}\,\mathrm{ClT}\ p) & = & \mathrm{FN}(p) \setminus \{o\} \\
\mathrm{FN}(o.m!\langle v_1, \ldots v_n\rangle?(r_1, \ldots r_{n'})\ p) & = & \left(\begin{array}{l} (\mathrm{FN}(p) \setminus \{r_1, \ldots r_{n'}\}) \\ \cup\ \{o\} \cup \\ \mathrm{FN}(v_1) \cup \ldots \cup \mathrm{FN}(v_n) \end{array}\right) \\
\mathrm{FN}(a!v\ p) & = & \mathrm{FN}(p) \cup \mathrm{FN}(v) \\
\mathrm{FN}(a?r\ p) & = & \mathrm{FN}(p) \setminus \{r\}
\end{array}
$$

3.1.4 Definitions

Code resides in method definitions and method definitions reside in class definitions. As well as the syntax of code, we need a syntax which describes how these definitions are written.

Oompa classes and interfaces are defined in a *definition set* which, in practical terms, corresponds to a file. *Class definitions* hold *attribute declarations*, which associate attribute names with attribute types, and *method definitions*, which consist of a signature and a piece of code. *Interface definitions* just contain a list of signatures. A *signature* has a *method name* and two lists, one specifying *input parameters* and one specifying *output parameters*.

sig	::=	$m?(r_1\colon T_1, \ldots r_n\colon T_n)!\langle s_1\colon S_1, \ldots s_{n'}\colon S_{n'}\rangle$	signature
mdef	::=	sig$\{p\}$	method definition
adecl	::=	a : `Attr`$\{T\}$	attribute declaration
Idef	::=	`interface` I $\{\text{sig}^*\}$	interface definition
Cdef	::=	`class` C $\{\text{adecl}^*\ \text{mdef}^*\}$	class definition
Γ	::=	$(\text{Idef} \mid \text{Cdef})^*$	definition set

where $T_1, \ldots T_n, S_1, \ldots S_{n'}$ are types and I and C are type variables (see Section 3.1.5). Note that the return parameters, $s_1, \ldots s_{n'}$, in a signature have no semantic significance. They serve as a convenient form of documentation.

The code in a method can return values to the code which invoked it by performing a send on the special channel name `return`. It can refer to its containing object by using the special object name `this`.

3.1.5 Types

Oompa is a statically type checked language, so an Oompa program can be checked for a certain class of misbehaviours called type violations. Supporting this procedure is a formal system, separate from Oompa's syntax and semantics, called a *type system* [Car96]. Although a certain amount of independence exists between a language and a type system for that language,

types can occur in Oompa's syntax and thus we need to provide their syntactic forms here. We leave a full consideration of Oompa's type system until Chapter 5.

The syntax of Oompa's type system is as follows:

$$
\begin{array}{rcll}
\mathrm{PrT} & ::= & \texttt{Long} \mid \texttt{Char} & \text{primitive types} \\
\mathrm{ChT} & ::= & \texttt{Chan}\langle T_1, \ldots T_n \rangle & \text{channel type} \\
 & \mid & \texttt{InCh}\langle T_1, \ldots T_n \rangle & \text{readable channel type} \\
 & \mid & \texttt{OuCh}\langle T_1, \ldots T_n \rangle & \text{writable channel type} \\
\mathrm{SgT} & ::= & m?(T_1, \ldots T_n)!\langle S_1, \ldots S_{n'} \rangle & \text{signature type} \\
\mathrm{InT} & ::= & \texttt{Intf}\{\mathrm{SgT}_1, \ldots \mathrm{SgT}_n\} & \text{interface type} \\
\mathrm{GrT} & ::= & \mathrm{InT} \mid \mathrm{ChT} & \text{guarded type} \\
T & ::= & \mathrm{PrT} \mid \mathrm{GrT} \mid t \mid \texttt{rec}\ t.\mathrm{GrT} & \text{value type} \\
\mathrm{AtT} & ::= & \texttt{Attr}\{T\} & \text{attribute type}
\end{array}
$$

where t is a type variable.

The type system has several kinds of types. We think of *primitive types* to be sets of literals and, for this thesis, we restrict consideration to long integers and characters. *Channel types* have three forms corresponding to channel access that permits reading, writing or both. A *signature type* holds the type of a method signature. Oompa objects have *interface types* which are lists of signature types describing the types of the available methods. The *Guarded types* are those type forms which have an outer type constructor (e.g. "`Chan`"). They simplify issues when using *recursive types* introduced with the `rec` operator. *Value types* are those type forms meaningful for values and parameters — these include all the type forms except attribute and signature types. *Type variables* have two roles, referring either to the type of an element of the definition set or to the `rec` binder in an enclosing recursive type. *Attribute types* are used to distinguish attributes from values.

We will sometimes refer to *class types* and represent them with ClT. A class type will be an occurrence of a free type variable that is the name of some class in the definition set. In fact, we do not consider classes to be types but permit the use of class names in type positions as we know how to obtain the unique interface type that corresponds to a class (see Section 5.2.1).

An example of an expression generated by this syntax is:

$$\texttt{rec}\ t.\texttt{Intf}\{m?()!\langle t\rangle\}$$

which would be the type of an object which has a single method m. The method takes no arguments but returns a value which is the name of an object of that same type. Note that the syntax disallows unguarded recursions, such as $\texttt{rec}\ t.t$, and recursions on recursive types, such as $\texttt{rec}\ t.\texttt{rec}\ s.\texttt{Chan}\langle t, s\rangle$. These limitations simplify proofs in Chapter 5.

3.1.6 Example of Oompa's syntax

Here is a simple example of an Oompa definition set. There is a class for simple cell objects, which manage a single long integer value. There is also a client, which creates a cell object, sends it a value and then inspects that value.

```
class Cell
{
  contents:Attr{Long}

  read?()!<val:Long>
  {
    contents?c
    return!<c>
```

```
      end
    }

    write?(val:Long)!<>
    {
      contents!c
      return!<>
      end
    }
  }

  class Cell_Client
  {
    main?()!<>
    {
      new cell:Cell
      cell.write!<5>?()
      cell.read!<>?(r)
      end
    }
  }
```

3.2 Issues in the design of Oompa

Some discussion of Oompa's syntax is in order.

Oompa does not have an expression language. We have deliberately postponed a consideration of this issue since evaluating expressions in a dis-

tributed environment is non-trivial and we do not wish to disguise that fact. An approach based on sugarings might still be useful and we discuss the possibility of this development in Chapter 8. In terms of the syntax, there are two main consequences of this.

Firstly, there is no atomic assignment for attributes. In many other object-oriented languages, such as Java, an attribute name can appear on the left and right of an assignment. In Oompa, the value of the attribute must be explicitly obtained and the attribute then updated; both separate actions. This is appropriate as it exposes important concurrency control issues. Furthermore, it works towards a style of refinement where local state is replaced by a remote object.

Secondly, there is no distinguished return value from an invocation. Instead, invocation binds a tuple of names to a tuple of return values when they arrive. As we will see in the semantics in Chapter 4, when an invocation occurs, a new channel is created and the calling code is blocked behind a receive on that channel. A method returns values to its caller by using the special channel name `return`.

Some programming languages have *type inference algorithms* [Car96]. This is a procedure which can automatically deduce the types in a program and allows some, or all, of the types to be omitted from the program text. As we currently do not provide a full type inference algorithm, Oompa's syntax must contain some type annotations. The parameters of a channel receive must be decorated with types but, on the other hand, we do not require type annotations for the parameter of an attribute access. An attribute's type is declared in the containing class and is therefore known statically.

3.3 Well-Formedness

There are a number of properties we wish to require of a definition set beyond mere syntactic conformance yet which do not belong to the semantically stronger concept of type correctness defined in Chapter 5. We call this somewhat disparate set of properties the *well-formedness properties* of a definition set. A definition set is *well-formed* if it satisfies the following conditions:

- Class names and interface names are all distinct.
- Within each class, attribute names are distinct and none are either `return` or `this`.
- Within each class, method names are distinct.
- A parameter may not occur twice in a parameter list.
- In the methods of a class, none of that class' attribute names may be used in a value or parameter position.

Ultimately, we imagine a predicate "WellFormed", easily implemented as a recursive function acting on the elements of Oompa's syntax.

An example of a syntactically correct class which violates well-formedness is:

```
class Ill_Behaviour
{
  // Two attributes with the same name!
  a:Attr{Long}
  a:Attr{Char}

  // Badly named attribute!
```

```
  this:Attr{Chan<>}

  // Method has repeated parameter names!
  meth?(l:Long, l:Long)!<k:Long>
  {
    // Attribute name used as a channel
    a!<b>
    // Attribute name used as a value!
    return!<a>
    end
  }
}
```

3.4 Renaming

The definition of Oompa requires a careful approach to the issue of name management. The approach we will use serves two main functions. Firstly, it will allow us to give a proper definition for substitution. Substitution is needed by Oompa's semantics and involves replacing names by values. Secondly, our approach to names will be used in Chapter 4 to relate Oompa's two dynamic systems. In order to do this, we will need to be able to change the bound names in various Oompa structures.

Issues of naming arise in many formalisms and the most common approach (used, for example, in [Bar80]) abstracts away the choice of bound names. An equivalence relation, called α-equivalence[4], is defined which iden-

[4] We will define an α-equivalence relations for several of Oompa's syntactic structures. See sections 3.5, 4.1.3 and 4.2.3.

tifies terms which differ only in their choices of bound names. Once this is done, α-equivalence classes of terms can be used instead of terms themselves.

The advantage of this approach is that bound names can be changed as needed and name clashes never need not occur. An aspect of Oompa's dynamic systems means that choice of names, even bound names, cannot be ignored in this way. Thus, the α-equivalence-class approach is not suitable for Oompa.

Another common approach is the de Bruijn approach [dB72]. This replaces bound names by numbers which act as indices to their binder. The disadvantage of this approach is that the indices are unintuitive to work with.

We elaborate on why we choose neither of these approaches in Section 4.2.2. We will also have an opportunity there to discuss some new approaches to naming ([GP99], [Hon00] and [FPT99]).

As we do not abstract away the choice of bound names, we will need a way of dealing with name clashes should they arise. We handle these using permutations on the name space and define their action on the various elements of Oompa's syntactic system. Although partial maps have been used in this way (one example is a capture-avoiding substitution defined in [Hen87]), we choose permutations since their composition behaves better and they are invertible.

The notion of "action" we employ is similar to, but usually weaker than, the standard notion of group action [Dur00]. A *group action* of a group, $(G, *)$, on a set, S, is an association of each element of the group with a function from the set to itself, such that

$$\begin{aligned} e(s) &= s & \forall s \in S \\ g_1(g_2(s)) &= (g_1 * g_2)(s) & \forall g_1, g_2 \in G,\ \forall s \in S \end{aligned}$$

where e is the group identity. The application of our renamings will satisfy

the first property but may only satisfy the second property up to some form of equivalence (i.e. not necessarily equality). Nevertheless, it seems acceptable to use the terms “act” and “action” and we will do so without mention.

In this thesis, statements of disjointness occur so frequently that we found it necessary to introduce an abbreviation for them. For sets A and B, we will write $A \pitchfork B$ to mean $A \cap B = \emptyset$.

3.4.1 Permutations of $\mathcal{N}$

A *permutation* of a set is a bijection from that set to itself. Given a permutation α of the set of names, $\mathcal{N}$, we will write its application using postfix form: $n\alpha$. We will write the composition of α_2 after α_1 as $\alpha_2 \circ \alpha_1$ and use the symbol 1 for the identity permutation. The set of permutations of a set with composition forms a group.

For us, an important property of a permutation of $\mathcal{N}$ will be its *set of changes*:

$$\text{Change}(\alpha) = \{n \in \mathcal{N} \mid n\alpha \neq n\}$$

A *renaming* is a permutation α of $\mathcal{N}$ such that $\text{Change}(\alpha)$ is finite. We will usually use the symbol ρ for a renaming.

Lemma 3.4.1

1. $\text{Change}(\rho_2 \circ \rho_1) \subseteq \text{Change}(\rho_1) \cup \text{Change}(\rho_2)$
2. $\text{Change}(1) = \emptyset$
3. $\text{Change}(\rho^{-1}) = \text{Change}(\rho)$

Corollary 3.4.2 *The set of renamings with composition forms a subgroup of the group of permutations of $\mathcal{N}$.*

We will use renamings to *act* on the various syntactic classes of our system. A renaming acts on a set of names point-wise:

$$A\rho = \{n\rho \mid n \in A\}$$

The following lemma means that the action of renamings on the set of sets of names is, in fact, a group action.

Lemma 3.4.3 *If A is a set of names and ρ_1, ρ_2 are two renamings, then $A\rho_1\rho_2 = A(\rho_2 \circ \rho_1)$.*

A few more definitions are needed. Two renamings ρ_1 and ρ_2 are *disjoint* if $\text{Change}(\rho_1) \pitchfork \text{Change}(\rho_2)$. A *transposition* is a permutation which maps two elements of a set to each other and maps all other elements of the set to themselves. Clearly, a transposition of $\mathcal{N}$ is a renaming. We will write the transposition which swaps the names n and n' as $(n\ n')$.

3.4.2 How to Pick New Names

Several of the functions we will define on Oompa structures involve choosing new names. A *new name* is a name which does not occur anywhere in the current context. Although the particular choice of this name may seem unimportant provided that it is new, the choice cannot be made arbitrarily. This is because we wish our functions to be pure[5] (a *pure function* always returns the same result when given the same arguments).

Formally, when a function involves the choice of a free name, our general procedure is to find the lexicographically highest name in its arguments and pick the next one. Informally, however, we shall pick arbitrary names for the sake of readability.

[5]In fact, this property is only required by a single result — Lemma 4.1.6.

3.4.3 Shoves

A feature of our renamings, which change only a finite number of names, is that there are always countably many names left unchanged. Occasionally we will want to protect certain names from the effect of some renaming by "moving them out of harms way". This is the role of shoves.

If A is a finite set of names, then a renaming is a *shove* of A, or *shoves* A, if it is built from disjoint transpositions swapping the elements of A with new names. We will usually use the symbol δ for a shove.

Lemma 3.4.4 *If δ is a shove then $\delta^{-1} = \delta$.*

In the following section, we see how shoves are used to protect bound names from interfering with the operation of changing free names in a piece of code.

3.4.4 The Action of a Renaming on Code

Although renamings are primarily used to change bound names, when we define how a renaming acts on code, we are primarily concerned with how it affects free names. The occurrences of a bound name occur free in the code beneath its binder; when we wish to change that name, it is those free occurrences that must be *renamed*.

A subtle danger, called *variable capture*, lies in changing free names. This occurs when a free name, falling under the scope of a binder, is changed and becomes bound. We will avoid this by using shoves to ensure the bound names are safely renamed before applying a given renaming.

The action of a renaming on code is defined as follows:

$$
\begin{array}{lcl}
\texttt{end}\rho & = & \texttt{end} \\
(\texttt{fork}\{p_0\}\ p_1)\rho & = & \texttt{fork}\{p_0\rho\}\ p_1\rho \\
(\texttt{new}\ c\colon \mathrm{ChT}\ p)\rho & = & \texttt{new}\ (c\delta\rho)\colon \mathrm{ChT}\ (p\delta\rho) \\
(c!\langle v_1, \dots v_n\rangle\ p)\rho & = & c\rho!\langle v_1\rho, \dots v_n\rho\rangle\ (p\rho) \\
(c?(r_1\colon T_1, \dots r_n\colon T_n)\ p)\rho & = & c\rho?(r_1\delta\rho\colon T_1, \dots r_n\delta\rho\colon T_n)\ (p\delta\rho) \\
(\texttt{create}\ o\colon \mathrm{ClT}\ p)\rho & = & \texttt{create}\ (o\delta\rho)\colon \mathrm{ClT}\ (p\delta\rho) \\
(o.m!\langle v_1, \dots v_n\rangle?(r_1, \dots r_{n'})\ p)\rho & = & o\rho.m!\langle v_1\rho, \dots v_n\rho\rangle?(r_1\delta\rho, \dots r_{n'}\delta\rho) \\
 & & \quad (p\delta\rho) \\
(a!v\ p)\rho & = & a!v\rho\ p\rho \\
(a?r\ p)\rho & = & a?r\delta\rho\ p\delta\rho
\end{array}
$$

where, in the five cases where shoves are introduced, they satisfy, respectively

- δ shoves $\{c\} \cap \mathrm{Change}(\rho)$
- δ shoves $\{r_1, \dots r_n\} \cap \mathrm{Change}(\rho)$
- δ shoves $\{o\} \cap \mathrm{Change}(\rho)$
- δ shoves $\{r_1, \dots r_{n'}\} \cap \mathrm{Change}(\rho)$
- δ shoves $\{r\} \cap \mathrm{Change}(\rho)$

We give an example to illustrate how a renaming acts on code. Say ρ is the renaming $(v\ c) \circ (z\ r)$. Then

$$
\begin{array}{ll}
 & (\texttt{new}\ c\colon T\ a?r\ z!\langle c\rangle\ r?(v)\ \texttt{end})\rho \\
= & \quad \langle \text{choosing the shove } (c\ k) \text{ to move } c \text{ out of the way of } \rho \rangle \\
 & \texttt{new}\ (c(c\ k))\colon T\ (a?r\ z!\langle c\rangle\ r!\langle v\rangle\ \texttt{end})(c\ k)\rho \\
= & \quad \langle \text{apply the renaming } (c\ k) \rangle \\
 & \texttt{new}\ k\colon T\ (a?r\ z!\langle k\rangle\ r!\langle v\rangle\ \texttt{end})\rho
\end{array}
$$

$=$ ⟨choosing the shove $(r\ s)$ to move r out of the way of ρ⟩

$\texttt{new}\ k\colon T\ a?r(r\ s)\ (z!\langle k\rangle\ r!\langle v\rangle\ \texttt{end})(r\ s)\rho$

$=$ ⟨apply the renaming $(r\ s)$⟩

$\texttt{new}\ k\colon T\ a?s\ (z!\langle k\rangle\ s!\langle v\rangle\ \texttt{end})\rho$

$=$ ⟨renaming procedure⟩

$\texttt{new}\ k\colon T\ a?s\ r!\langle k\rangle\ (s!\langle v\rangle\ \texttt{end})\rho$

$=$ ⟨renaming procedure⟩

$\texttt{new}\ k\colon T\ a?s\ r!\langle k\rangle\ s!\langle c\rangle\ (\texttt{end})\rho$

$=$ ⟨renaming procedure⟩

$\texttt{new}\ k\colon T\ a?s\ r!\langle k\rangle\ s!\langle c\rangle\ \texttt{end}$

The following lemma reassures us that the renaming procedure succeeds in changing the free names appropriately. We include its proof since it provides useful intuition for how renaming operates.

Lemma 3.4.5 *For a piece of code p and a renaming ρ,*

$$\mathrm{FN}(p\rho) = \mathrm{FN}(p)\rho$$

Proof: We use induction on the construction of p. The only interesting cases occur when there are bound names and the case for channel creation illustrates the reasoning in these. Say $p = \texttt{new}\ c\colon T\ p_1$. Then

$\mathrm{FN}((\texttt{new}\ c\colon T\ p_1)\rho)$

$=$ ⟨the renaming procedure, choosing δ appropriately⟩

$\mathrm{FN}(\texttt{new}\ (c\delta\rho)\colon T\ (p_1\delta\rho))$

$=$ ⟨definition of $\mathrm{FN}(\cdot)$⟩

$\mathrm{FN}(p_1\delta\rho) \setminus \{c\delta\rho\}$

$=$ ⟨induction hypothesis twice and set theory⟩

$\mathrm{FN}(p_1)\delta\rho \setminus \{c\}\delta\rho$

$= \quad \langle \rho$ and δ are bijective$\rangle$

$(\mathrm{FN}(p_1) \smallsetminus \{c\})\delta\rho$

$= \quad \langle$choice of $\delta\rangle$

$(\mathrm{FN}(p_1) \smallsetminus \{c\})\rho$

$= \quad \langle$definition of $\mathrm{FN}(\cdot)\rangle$

$\mathrm{FN}(\texttt{new}\ c{:}T\ p_1)\rho$

■

3.4.5 The Definition of Substitution on Code

Substitution can occur in two ways in Oompa: when values are received by channel based communication, method invocation or attribute access and when a new channel or object is created. The *simultaneous substitution* of values $v_1, \ldots v_n$ for names $n_1, \ldots n_n$ is written $\{\!|{}^{v_1}/n_1, \ldots {}^{v_n}/n_n|\!\}$. The effect of a substitution on a value is given by:

$$v\{\!|{}^{v_1}/n_1, \ldots {}^{v_n}/n_n|\!\} = \begin{cases} v_i & \text{if } v \text{ is } n_i \\ v & \text{otherwise} \end{cases}$$

We will typically use the symbol σ for a substitution.

When we consider the effect of a substitution on code, we must be as wary of variable capture as we were when considering renaming — names can be substituted by other names. As in the case with renamings, we use shoves to change offending bound names in advance of applying the substitution.

The effect of a substitution $\sigma = \{\!|{}^{v_1}/n_1, \ldots {}^{v_n}/n_n|\!\}$ on a piece of code is

given as follows:

$$
\begin{array}{lcl}
\texttt{end}\sigma & = & \texttt{end} \\
(\texttt{fork}\{p_0\}\ p_1)\sigma & = & \texttt{fork}\{p_0\sigma\}\ p_1\sigma \\
(\texttt{new}\ c{:}\,\mathrm{ChT}\ p)\sigma & = & \texttt{new}\ (c\delta\sigma){:}\,\mathrm{ChT}\ (p\delta\sigma) \\
(c!\langle v_1, \ldots v_n\rangle\ p)\sigma & = & c\sigma!\langle v_1\sigma, \ldots v_n\sigma\rangle\ (p\sigma) \\
(c?(r_1{:}\,T_1, \ldots r_n{:}\,T_n)\ p)\sigma & = & c\sigma?(r_1\delta\sigma{:}\,T_1, \ldots r_n\delta\sigma{:}\,T_n)\ (p\delta\sigma) \\
(\texttt{create}\ o{:}\,\mathrm{ClT}\ p)\sigma & = & \texttt{create}\ (o\delta\sigma){:}\,\mathrm{ClT}\ (p\delta\sigma) \\
(o.m!\langle v_1, \ldots v_n\rangle?(r_1, \ldots r_{n'})\ p)\sigma & = & o\sigma.m!\langle v_1\sigma, \ldots v_n\sigma\rangle?(r_1\delta\sigma, \ldots r_{n'}\delta\sigma) \\
 & & \quad (p\delta\sigma) \\
(a!v\ p)\sigma & = & a!v\sigma\ p\sigma \\
(a?r\ p)\sigma & = & a?r\delta\sigma\ p\delta\sigma
\end{array}
$$

where, in the five cases where shoves are introduced, they satisfy, respectively

- δ shoves $\{c\} \cap (\{v_1, \ldots v_n\} \cup \{n_1, \ldots n_n\})$
- δ shoves $\{r_1, \ldots r_n\} \cap (\{v_1, \ldots v_n\} \cup \{n_1, \ldots n_n\})$
- δ shoves $\{o\} \cap (\{v_1, \ldots v_n\} \cup \{n_1, \ldots n_n\})$
- δ shoves $\{r_1, \ldots r_{n'}\} \cap (\{v_1, \ldots v_n\} \cup \{n_1, \ldots n_n\})$
- δ shoves $\{r\} \cap (\{v_1, \ldots v_n\} \cup \{n_1, \ldots n_n\})$

3.5 α-Equivalence for Code

As we discussed in Section 3.4, we regard the particular choice of names to be important. Consequently, we cannot consider two pieces of code, which differ only in their choices of bound names, to be equal. Yet for many purposes, equality is overly discriminating and a coarser relation will be preferable.

Table 3.1: Equivalence Rules for Code α-Equivalence

EQV-RFL:	$p \equiv_\alpha p$
EQV-SYM:	$\dfrac{p_1 \equiv_\alpha p_2}{p_2 \equiv_\alpha p_1}$
EQV-TRN:	$\dfrac{p_1 \equiv_\alpha p_2 \qquad p_2 \equiv_\alpha p_3}{p_1 \equiv_\alpha p_3}$

Two pieces of code, p_1 and p_2, are said to be α-*equivalent*, written $p_1 \equiv_\alpha p_2$, if they differ only in their choices of bound names. This is formalised with the proof system given by the sixteen rules in tables 3.1, 3.2 and 3.3.

The three rules in Table 3.1, EQV-RFL, EQV-SYM and EQV-TRN, mean that $\equiv_\alpha$ is an equivalence relation. The eight rules in Table 3.2, CONG-FRK, CONG-NEW, CONG-SND, CONG-RCV, CONG-CRT, CONG-INV, CONG-ACC and CONG-UPD, mean that it is a congruence relation over code. The five rules in Table 3.3, CHNG-NEW, CHNG-RCV, CHNG-CRT, CHNG-INV and CHNG-ACC, permit bound names to be changed.

The following three lemmas describe some desirable properties of code α-equivalence. We defer their proofs until Appendix B and annotate them with a forward reference to their proofs. We see by Lemma 3.5.2 that the action of renamings on code is not a full group action.

B.1.1 p242

Lemma 3.5.1 *For two pieces of code p_1 and p_2,*

$$p_1 \equiv_\alpha p_2 \Rightarrow \mathrm{FN}(p_1) = \mathrm{FN}(p_2)$$

B.1.2 p244

Lemma 3.5.2 *For a piece of code p and two renamings ρ_1 and ρ_2,*

$$p\rho_1\rho_2 \equiv_\alpha p(\rho_2 \circ \rho_1)$$

B.1.3 p248

Lemma 3.5.3 *For two pieces of code p_1 and p_2 and a renaming ρ,*

$$p_1 \equiv_\alpha p_2 \Rightarrow p_1\rho \equiv_\alpha p_2\rho$$

Table 3.2: Congruence Rules for Code α-Equivalence

CONG-FRK:
$$\frac{p_1 \equiv_\alpha p_2 \qquad p_1' \equiv_\alpha p_2'}{\texttt{fork}\{p_1\}\ p_1' \equiv_\alpha \texttt{fork}\{p_2\}\ p_2'}$$

CONG-NEW:
$$\frac{p_1 \equiv_\alpha p_2}{\texttt{new}\ c\colon \mathrm{ChT}\ p_1 \equiv_\alpha \texttt{new}\ c\colon \mathrm{ChT}\ p_2}$$

CONG-SND:
$$\frac{p_1 \equiv_\alpha p_2}{c!\langle v_1, \ldots v_n\rangle\ p_1 \equiv_\alpha c!\langle v_1, \ldots v_n\rangle\ p_2}$$

CONG-RCV:
$$\frac{p_1 \equiv_\alpha p_2}{c?(r_1\colon T_1, \ldots r_n\colon T_n)\ p_1 \equiv_\alpha c?(r_1\colon T_1, \ldots r_n\colon T_n)\ p_2}$$

CONG-CRT:
$$\frac{p_1 \equiv_\alpha p_2}{\texttt{create}\ o\colon \mathrm{ClT}\ p_1 \equiv_\alpha \texttt{create}\ o\colon \mathrm{ClT}\ p_2}$$

CONG-INV:
$$\frac{p_1 \equiv_\alpha p_2}{o.m!\langle v_1, \ldots v_n\rangle?(r_1, \ldots r_n)\ p_1 \equiv_\alpha o.m!\langle v_1, \ldots v_n\rangle?(r_1, \ldots r_n)\ p_2}$$

CONG-UPD:
$$\frac{p_1 \equiv_\alpha p_2}{a!v\ p_1 \equiv_\alpha a!v\ p_2}$$

CONG-ACC:
$$\frac{p_1 \equiv_\alpha p_2}{a?r\ p_1 \equiv_\alpha a?r\ p_2}$$

Table 3.3: Change of Bound Name Rules for Code α-Equivalence

CHNG-NEW: $\texttt{new}\ c\colon \mathrm{ChT}\ p \equiv_\alpha \texttt{new}\ c\rho\colon \mathrm{ChT}\ p\rho$

where ρ is any renaming such that $\mathrm{FN}(\texttt{new}\ c\colon \mathrm{ChT}\ p) \pitchfork \mathrm{Change}(\rho)$.

CHNG-RCV: $c?(r_1\colon T_1, \ldots r_n\colon T_n)\ p \equiv_\alpha c?(r_1\rho\colon T_1, \ldots r_n\rho\colon T_1)\ p\rho$

where ρ is any renaming such that:

$$\mathrm{FN}(c?(r_1\colon T_1, \ldots r_n\colon T_n)\ p) \pitchfork \mathrm{Change}(\rho)$$

CHNG-CRT: $\texttt{create}\ o\colon \mathrm{ClT}\ p \equiv_\alpha \texttt{create}\ o\rho\colon \mathrm{ClT}\ p\rho$

where ρ is any renaming such that $\mathrm{FN}(\texttt{create}\ o\colon \mathrm{ClT}\ p) \pitchfork \mathrm{Change}(\rho)$.

CHNG-INV:

$$o.m!\langle v_1, \ldots v_n\rangle?(r_1, \ldots r_n)\ p \equiv_\alpha o.m!\langle v_1, \ldots v_n\rangle?(r_1\rho, \ldots r_{n'}\rho)\ p\rho$$

where ρ is any renaming such that:

$$\mathrm{FN}(o.m!\langle v_1, \ldots v_n\rangle?(r_1, \ldots r_n)\ p) \pitchfork \mathrm{Change}(\rho)$$

CHNG-ACC: $a?r\ p \equiv_\alpha a?r\rho\ p\rho$

where ρ is any renaming such that $\mathrm{FN}(a?r\ p) \pitchfork \mathrm{Change}(\rho)$.

3.6 Summary

The primary goal of this chapter has been to present the theory of Oompa's static system. Defining this system involved a formal definition of the syntax of Oompa programs and an accompanying discussion of well-formedness. Within the scope of this chapter was the provision of formal support for manipulating and reasoning about the structures of the static system. Thus, we provided procedures for both substitution and renaming and an α-equivalence system for code.

Chapter 4

Dynamic Oompa

Oompa's static system, covered in Chapter 3, defines how Oompa programs are built out of class and interface definitions. Our intention is to consider systems of interacting objects but we have not yet defined the form or behaviour of a running Oompa system. Using the terminology of the CORBA object model (see Section 1.1.2) we have explained how *code* is written but not provided an *execution engine* to interpret it. This is the role of Oompa's dynamic system.

In this chapter, we actually provide Oompa with two dynamic systems. Both have advantages and disadvantages and we will want both. Fortunately, we can show that they are semantically compatible. For both dynamic systems, we provide an operational semantics.

The first of the dynamic systems, Oompa's agent-based dynamic system, can describe the state and behaviour of a complete closed Oompa system. It has the advantage of being fairly intuitive but suffers from being non-compositional. The second, Oompa's configuration-based dynamic system, can consider the parts of an Oompa system independently and describe how parts interact with their environment. This is good for reasoning, since it is

compositional, but it is much more complex and less intuitive.

Sections 4.1 and 4.2 define Oompa's agent-based and configuration-based dynamic systems, respectively. In both cases, this involves defining their syntax, discussing renaming, defining equivalence relations and giving their semantics. For the configuration-based dynamic system, we also define a semantic equivalence called weak bisimulation. We relate the two dynamic systems, in Section 4.3, by defining an algorithm which translates from one to the other and using it to provide a kind of simulation between the them. Finally, Section 4.4 summarises the chapter.

4.1 Agent-Based Dynamic System

Oompa's *agent-based dynamic system* considers a running Oompa program to be a triple consisting of the currently executing object code, a dictionary of type information and a dictionary of state information. The operational semantics for this dynamic system is a reduction relation which describes the evolution of this triple in the context of the definition set. We can view this dynamic system as corresponding to a natural intuition of how a system of objects behaves.

4.1.1 Syntax

First, we consider the syntax of the agent-based dynamic system.

Syntax of Agents

Syntactically, we represent the running object code as an *agent expression* or just *agent*. The basic form of an agent is a piece of code associated with an object name. Since all attributes are private in Oompa, we need to mark

agents with the objects to which they belong in order to control access to state. The code will be the (modified) body of some method belonging to the object's class. Agents can also be nil or consist of two agents in parallel.

$$\begin{array}{rcll} g & ::= & \text{nil} & \text{null agent} \\ & | & o[p] & \text{code } p \text{ running at object } o \\ & | & g_0 \mid g_1 & \text{two agents in parallel} \end{array}$$

where the syntax of the code, p, is as defined in Section 3.1.3. The free names in an agent are found using

$$\begin{array}{rcl} \text{FN}(\text{nil}) & = & \emptyset \\ \text{FN}(o[p]) & = & \text{FN}(p) \cup \{o\} \\ \text{FN}(g_0 \mid g_1) & = & \text{FN}(g_0) \cup \text{FN}(g_1) \end{array}$$

An example of an agent is:

$$(o_1[\texttt{contents?r}\ x!\langle\texttt{r}\rangle\ \texttt{end}] \mid o_2[x?(\texttt{r})\ \texttt{end}]) \mid (o_1[\texttt{end}] \mid \text{nil})$$

Its free names are o_1, o_2 and x.

The Type and State Dictionaries

A *type dictionary* holds a sequence of typings for values. Its most important role is in holding the class type of an object so that, when an invocation is made upon the object, its class can be looked up in the definition set[1].

$$\begin{array}{rcl} \text{tAss} & ::= & v\colon T \\ \Phi & ::= & \{\text{tAss}_1, \ldots \text{tAss}_n\} \end{array}$$

We will want to know the set of values typed by a type dictionary and, on occasion, those object names typed by a type dictionary.

$$\begin{array}{rcl} \text{Dom}(\Phi) & = & \{v \mid v\colon T \in \Phi\} \\ \text{OBJECTS}(\Phi) & = & \{o \mid o\colon \text{ClT} \in \Phi\} \end{array}$$

[1]A secondary role of the type dictionary is to support run-time typing. Oompa can also support a `typecase` primitive which we have chosen not to discuss in this thesis.

Lemma 4.1.1 *For a type dictionary* Φ*,* $\mathrm{OBJECTS}(\Phi) \subseteq \mathrm{Dom}(\Phi)$*.*

A *state dictionary* manages the state of objects. For each object name in its domain it has a dictionary of attribute assignments.

$$\begin{array}{rcl} \mathrm{aAss} & ::= & a \mapsto v \\ \mathrm{oAss} & ::= & o := [\mathrm{aAss}_1, \ldots \mathrm{aAss}_n] \\ \Delta & ::= & \{\mathrm{oAss}_1, \ldots \mathrm{oAss}_n\} \end{array}$$

The domain of a state dictionary is the set of objects to which it assigns state.

$$\mathrm{Dom}(\Delta) = \{o \mid (o := [\mathrm{aAss}_1, \ldots \mathrm{aAss}_n]) \in \Delta\}$$

A state dictionary can have free names, which are the objects in its domain and the free names of the values it assigns to attributes.

$$\begin{array}{lcl} \mathrm{FN}(a \mapsto v) & = & \mathrm{FN}(v) \\ \mathrm{FN}(o := [\mathrm{aAss}_1, \ldots \mathrm{aAss}_n]) & = & \{o\} \cup \mathrm{FN}(\mathrm{aAss}_1) \cup \ldots \cup \mathrm{FN}(\mathrm{aAss}_n) \\ \mathrm{FN}(\{\mathrm{oAss}_1, \ldots \mathrm{oAss}_n\}) & = & \mathrm{FN}(\mathrm{oAss}_1) \cup \ldots \cup \mathrm{FN}(\mathrm{oAss}_n) \end{array}$$

We will occasionally use type, state or attribute dictionaries as if they were partial functions. This explains our use of the term *domain*. Moreover, we will occasionally use function application notation on dictionaries, writing $\Phi(v)$ for the type assigned to value v by the type dictionary Φ.

An example of a type dictionary is:

$$\{o_1{:}\,\texttt{Cell},\ o_2{:}\,\texttt{Cell_Client},\ x{:}\,\texttt{Chan}\langle\texttt{Long}\rangle\}$$

Here `Cell` and `Cell_Client` are class types, i.e. the names of classes in the definition set (see the example in Section 3.1.6). Its domain is $\{o_1, o_2, x\}$ and its objects are $\{o_1, o_2\}$. An example of a state dictionary is:

$$\{o_1 := [\texttt{contents} \mapsto 5],\ o_2 := \emptyset\}$$

Its domain and its set of free names are $\{o_1, o_2\}$.

Agent System

The dynamic part of the agent system is an agent, a type dictionary and a state dictionary. We will call this triple an *agent configuration* and write it

$$g\{^{\Phi}_{\Delta}$$

An *agent system* consists of an agent configuration in the context of a definition set, Γ. This is written

$$\Gamma \triangleright g\{^{\Phi}_{\Delta}$$

The syntax of definition sets is given in Section 3.1.4.

An example of an agent configuration would be:

$$\begin{array}{ll} & o_1[\texttt{contents?r}\ x!\langle\texttt{r}\rangle\ \texttt{end}] \\ | & o_2[x?(\texttt{r})\ \texttt{end}] \\ | & o_1[\texttt{end}] \\ | & \text{nil} \end{array} \left\{ \begin{array}{l} \left\{ \begin{array}{l} o_1\colon \texttt{Cell}, \\ o_2\colon \texttt{Cell_Client}, \\ x\colon \texttt{Chan}\langle\texttt{Long}\rangle \end{array} \right\} \\ \left\{ \begin{array}{l} o_1 := [\texttt{contents} \mapsto 5], \\ o_2 := \emptyset \end{array} \right\} \end{array} \right.$$

In fact, this agent configuration isn't strictly valid. According to the syntax of agents, the parallel operator is a binary operator but we have used it to group four agents. We don't consider the grouping of agents to be significant and the basic equivalence relation for agents, structural equivalence (see Section 4.1.3), justifies our use of a generalised parallel operator here.

Initial System Convention

We need a way to populate our system with objects. We adopt a convention that allows a definition set to imply a specific agent system to be the starting state of the system. Usefully, we allow this initial agent system to have global channel names.

Let $\{C_1, \ldots C_n\}$ be those classes in the definition set Γ which have a method with signature main?()!⟨⟩. Say the bodies of these methods are $p_1, \ldots p_n$ and none of the p_i uses either `this`, `return` or an attribute. Say that OBJECTS$(\Phi) = \emptyset$. Then

$$\Gamma \rhd \mathit{dummy_1}[p_1]| \ldots \mathit{dummy_n}[p_n]\{^{\Phi}_{\emptyset}$$

is an *initial agent system*. We will sometimes use the notation g_Γ to denote the agent involved.

As an illustration, we will consider the initial agent system specified by the definition set in Section 3.1.6. There is only one class which has a method of the appropriate signature: `Cell_Client`. The code of its main method doesn't use `return`, `this` or an attribute so an initial agent is created for this agent. There are no names free in the definition set, so we won't need any global channel names to be typed by the type dictionary. Therefore, the agent configuration of the initial agent system is the following.

$$\mathit{dummy_1}[\texttt{new cell:Cell cell.write!}\langle 5\rangle?()\ \texttt{cell.read!}\langle 5\rangle?(\texttt{r})\ \texttt{end}] \quad \begin{cases} \emptyset \\ \emptyset \end{cases}$$

The initial agent system is just this agent configuration in the context of the definition set.

4.1.2 Renaming

We consider here how a renaming acts on the elements of the agent-based dynamic system. Although not necessary for the agent-based dynamic system itself, we will need to be able to rename these structures when we consider the relationship between the two dynamic systems. The following table describes

how a renaming acts on an agent, a type dictionary and a state dictionary.

$$\begin{array}{lcl}
\mathrm{nil}\rho & = & \mathrm{nil} \\
(o[p])\rho & = & (o\rho)[p\rho] \\
(g_0|g_1)\rho & = & ((g_0\rho)|(g_1\rho))
\end{array}$$

$$\begin{array}{lcl}
\{\mathrm{tAss}_1, \ldots \mathrm{tAss}_n\}\rho & = & \{(\mathrm{tAss}_1\rho), \ldots (\mathrm{tAss}_n\rho)\} \\
(v{:}\,T)\rho & = & (v\rho){:}\,T
\end{array}$$

$$\begin{array}{lcl}
(\{\mathrm{oAss}_0, \ldots \mathrm{oAss}_n\})\rho & = & \{(\mathrm{oAss}_0\rho), \ldots (\mathrm{oAss}_n\rho)\} \\
(o := [\mathrm{aAss}_0, \ldots \mathrm{aAss}_n])\rho & = & (o\rho) := [(\mathrm{aAss}_0\rho), \ldots (\mathrm{aAss}_n\rho)] \\
(a \mapsto v)\rho & = & a \mapsto (v\rho)
\end{array}$$

Some useful results about the action of renamings on agents are the following. By Lemma 4.1.4 we see that the action of a renaming on type or state dictionaries is, in fact, a group action. This follows from the fact that dictionaries have no local bound names.

Lemma 4.1.2 *For an agent g and a renaming ρ,*

$$\mathrm{FN}(g\rho) = \mathrm{FN}(g)\rho$$

Lemma 4.1.3 *For a renaming ρ, type dictionary Φ and state dictionary Δ:*

$$\begin{array}{rcl}
\mathrm{Dom}(\Phi\rho) & = & \mathrm{Dom}(\Phi)\rho \\
\mathrm{OBJECTS}(\Phi\rho) & = & \mathrm{OBJECTS}(\Phi)\rho \\
\mathrm{Dom}(\Delta\rho) & = & \mathrm{Dom}(\Delta)\rho \\
\mathrm{FN}(\Delta\rho) & = & \mathrm{FN}(\Delta)\rho
\end{array}$$

Lemma 4.1.4 *For two renamings ρ_1, ρ_2, a type dictionary Φ and state dictionary Δ:*

$$\begin{array}{rcl}
\Phi\rho_1\rho_2 & = & \Phi(\rho_2 \circ \rho_1) \\
\Delta\rho_1\rho_2 & = & \Delta(\rho_2 \circ \rho_1)
\end{array}$$

4.1.3 Equivalence

We consider three equivalence relations for the agent-based dynamic system. These are structural equivalence, α-equivalence and α-/structural equivalence. Having three relations allows us to be quite discriminating when describing properties — some results can be proved up to structural equivalence, some up to α-equivalence and others only up to α-/structural equivalence. The operational semantics of the agent-based dynamic system uses structural equivalence to manipulate the structure of agents. The other two equivalence relations are necessary for the theory developed later.

Structural Equivalence

The structural equivalence relation between two agents, g_1 and g_2, written $g_1 \overset{\triangleright}{\equiv} g_2$, identifies agents which differ only in the reordering and regrouping of sub-agents and the presence and absence of nils. It is defined by the seven rules[2] in Table 4.1.

The three rules, EQV-RFL, EQV-SYM and EQV-TRN, mean that $\overset{\triangleright}{\equiv}$ is an equivalence relation. The single rule, CONG-PAR, means that it is a congruence over agents. The three rules, PAR-ID, PAR-ASS and PAR-ABL, mean that, with respect to $\overset{\triangleright}{\equiv}$, $|$ forms an Abelian monoid.

We can show that $\overset{\triangleright}{\equiv}$ has the following properties.

Lemma 4.1.5 *For two agents g_1 and g_2,*

$$g_1 \overset{\triangleright}{\equiv} g_2 \Rightarrow \mathrm{FN}(g_1) = \mathrm{FN}(g_2)$$

[2]Note that we allow rule names to be reused between proof systems, e.g. α-equivalence for code and structural equivalence for agents both have a rule called EQV-RFL. In cases of possible ambiguity, we will make explicit which rule is meant.

Table 4.1: Rules for Agent Structural Equivalence

EQV-RFL:	$g \stackrel{\triangleright}{\equiv} g$
EQV-SYM:	$\dfrac{g_1 \stackrel{\triangleright}{\equiv} g_2}{g_2 \stackrel{\triangleright}{\equiv} g_1}$
EQV-TRN:	$\dfrac{g_1 \stackrel{\triangleright}{\equiv} g_2 \qquad g_2 \stackrel{\triangleright}{\equiv} g_3}{g_1 \stackrel{\triangleright}{\equiv} g_3}$
CONG-PAR:	$\dfrac{g_1 \stackrel{\triangleright}{\equiv} g_2 \qquad g_1' \stackrel{\triangleright}{\equiv} g_2'}{(g_1 \vert g_1') \stackrel{\triangleright}{\equiv} (g_2 \vert g_2')}$
PAR-ID:	$(g \vert \mathrm{nil}) \stackrel{\triangleright}{\equiv} g$
PAR-ASS:	$((g_1 \vert g_2) \vert g_3) \stackrel{\triangleright}{\equiv} (g_1 \vert (g_2 \vert g_3))$
PAR-ABL:	$(g_1 \vert g_2) \stackrel{\triangleright}{\equiv} (g_2 \vert g_1)$

Lemma 4.1.6 *For two agents g_1 and g_2 and a renaming ρ,*

$$g_1 \stackrel{\triangleright}{\equiv} g_2 \Rightarrow g_1\rho \stackrel{\triangleright}{\equiv} g_2\rho$$

α-Equivalence

The α-equivalence relation between two agents, g_1 and g_2, written $g_1 \equiv_\alpha g_2$, merely lifts α-equivalence on code to agents. It is defined by the five rules in Table 4.2.

The three rules, EQV-RFL, EQV-SYM and EQV-TRN, mean that $\equiv_\alpha$ is an equivalence relation. The single rule, CONG-PAR, means that it is a congruence over agents. The rule, ALPHA-COD, means that it inherits code α-equivalence.

The following lemmas describe some of the desirable properties of $\equiv_\alpha$.

Lemma 4.1.7 *For two agents g_1 and g_2,*

$$g_1 \equiv_\alpha g_2 \Rightarrow \mathrm{FN}(g_1) = \mathrm{FN}(g_2)$$

Table 4.2: Rules for Agent α-Equivalence

Rule	
EQV-RFL:	$g \equiv_\alpha g$
EQV-SYM:	$\dfrac{g_1 \equiv_\alpha g_2}{g_2 \equiv_\alpha g_1}$
EQV-TRN:	$\dfrac{g_1 \equiv_\alpha g_2 \qquad g_2 \equiv_\alpha g_3}{g_1 \equiv_\alpha g_3}$
CONG-PAR:	$\dfrac{g_1 \equiv_\alpha g_2 \qquad g_1' \equiv_\alpha g_2'}{(g_1 \mid g_1') \equiv_\alpha (g_2 \mid g_2')}$
ALPHA-COD:	$\dfrac{p_1 \equiv_\alpha p_2}{o[p_1] \equiv_\alpha o[p_2]}$

Lemma 4.1.8 *For an agent g and two renamings ρ_1 and ρ_2,*

$$g\rho_1\rho_2 \equiv_\alpha g(\rho_2 \circ \rho_1)$$

Lemma 4.1.9 *For two agents g_1 and g_2 and a renaming ρ,*

$$g_1 \equiv_\alpha g_2 \Rightarrow g_1\rho \equiv_\alpha g_2\rho$$

α-/Structural Equivalence

The α-/structural equivalence relation between two agents, g_1 and g_2, written $g_1 \overset{\triangleright}{\equiv}_\alpha g_2$, encompasses both α- and structural equivalence[3]. It is defined by the six rules in Table 4.3.

The three rules, EQV-RFL, EQV-SYM and EQV-TRN, mean that $\overset{\triangleright}{\equiv}_\alpha$ is an equivalence relation. The single rule, CONG-PAR, mean that it is a congruence rule over agents. The two rules, ALPHA and STRUCT, mean

[3]We show in Appendix B, Lemma B.2.4, that an instance of α-/structural equivalence can be split into an instance of α-equivalence followed by an instance of structural equivalence or vice-versa.

Table 4.3: Rules for Agent α-/Structural Equivalence

Rule	
EQV-RFL:	$g \overset{\triangleright}{\equiv}_\alpha g$
EQV-SYM:	$\dfrac{g_1 \overset{\triangleright}{\equiv}_\alpha g_2}{g_2 \overset{\triangleright}{\equiv}_\alpha g_1}$
EQV-TRN:	$\dfrac{g_1 \overset{\triangleright}{\equiv}_\alpha g_2 \qquad g_2 \overset{\triangleright}{\equiv}_\alpha g_3}{g_1 \overset{\triangleright}{\equiv}_\alpha g_3}$
CONG-PAR:	$\dfrac{g_1 \overset{\triangleright}{\equiv}_\alpha g_2 \qquad g_1' \overset{\triangleright}{\equiv}_\alpha g_2'}{(g_1 \vert g_1') \overset{\triangleright}{\equiv}_\alpha (g_2 \vert g_2')}$
ALPHA:	$\dfrac{g_1 \equiv_\alpha g_2}{g_1 \overset{\triangleright}{\equiv}_\alpha g_2}$
STRUCT:	$\dfrac{g_1 \overset{\triangleright}{\equiv} g_2}{g_1 \overset{\triangleright}{\equiv}_\alpha g_2}$

that it inherits α-equivalence and structural equivalence. We can show that it has the following properties.

Lemma 4.1.10 *For two agents g_1 and g_2,*

$$g_1 \overset{\triangleright}{\equiv}_\alpha g_2 \Rightarrow \mathrm{FN}(g_1) = \mathrm{FN}(g_2)$$

Lemma 4.1.11 *For two agents g_1 and g_2 and a renaming ρ,*

$$g_1 \overset{\triangleright}{\equiv}_\alpha g_2 \Rightarrow g_1\rho \overset{\triangleright}{\equiv}_\alpha g_2\rho$$

4.1.4 Semantics

In order to give meaning to the terms of our formalism, we need to provide a semantics. The semantics assigned to formalisms and programming languages typically fall into three categories: Denotational semantics, algebraic semantics and operational semantics.

Denotational semantics map the terms of a formalism into structures in some mathematical space. The mathematical properties of those structures capture the intended properties of the terms. Denotational semantics are particularly useful for systems where input/output behaviour is key (e.g. pure functional languages). When denotational semantics are applied to languages with interactive behaviour they tend to be quite "operational". For example, CSP's denotational semantics are defined partially in terms of sequences of actions.

Algebraic semantics consist of an axiomatic system from which facts about the meanings of terms can be derived. Typically, the facts will have the form of equations and assert that the meanings of two terms are equivalent in some way. Undoubtedly, an algebraic semantics for Oompa would be useful but, for such a complex system, it is not obvious how to define one without first establishing an operational intuition.

Operational semantics give meaning to terms by defining how they behave, typically by describing how they execute a single operation and transform into other terms. This is particularly useful for programs with interactive behaviour since the way a program interacts with its environment depends not just on the terms it can reach but also on the execution path it followed to reach them. Operational semantics are usually more intuitive than the other semantics as they supply an *operational intuition* for the behaviour of terms. For Oompa's two dynamic system, we choose to define operational semantics.

The semantics we define for the agent-based dynamic system is a reduction relation describing the internal global behaviour of an agent configuration $g\{^{\Phi}_{\Delta}$ in the context of a definition set Γ. It is a reaction system where agents attempting corresponding actions (e.g. a send and a receive) evolve

together[4]. The reduction relation is defined by the proof system whose eleven rules are given in tables 4.4, 4.5 and 4.6.

There are three general purpose inference rules given in Table 4.4, EQV-LFT, EQV-RHT and PAR, and eight axioms, which we briefly discuss here.

The four rules in Table 4.5 deal with the primitives which form the π-calculus core of Oompa. The END rule allows agents which have finished their work to be removed from the agent expression. The FRK rule splits an agent into two parallel agents. The NEW rule chooses a new channel name and adds it to the global type dictionary. The COM rule allows two agents attempting to send and receive on the same channel to communicate, provided their tuple lists are the same length.

The four rules in Table 4.6 deal with Oompa's object-orientation. The CRT rule introduces a new object by choosing a new name and adding an entry to both the type and state dictionaries. The INV rule creates an agent for the invoked method, chooses a new channel name for returning values and blocks the invoking agent behind a receive on that name. The UPD rule updates the value of the attribute or, if there is no appropriate entry in the state dictionary, adds one. The ACC rule accesses the value of the attribute if there is one, substituting the stored value in place of the given parameter.

Example of a Reduction

We consider a few steps of a reduction of the initial system given in Section 4.1.1, in the context of the definition set given in Section 3.1.6.

[4]Internal actions of this sort are sometimes called *intraactions* [SW01].

Table 4.4: General Rules for the Agent-Based Operational Semantics

EQV-LFT:
$$\frac{g_1 \overset{\rhd}{\equiv} g'_1 \qquad \Gamma \rhd g_1\{^{\Phi_1}_{\Delta_1} \longrightarrow g_2\{^{\Phi_2}_{\Delta_2}}{\Gamma \rhd g'_1\{^{\Phi_1}_{\Delta_1} \longrightarrow g_2\{^{\Phi_2}_{\Delta_2}}$$

EQV-RHT:
$$\frac{g_2 \overset{\rhd}{\equiv} g'_2 \qquad \Gamma \rhd g_1\{^{\Phi_1}_{\Delta_1} \longrightarrow g_2\{^{\Phi_2}_{\Delta_2}}{\Gamma \rhd g_1\{^{\Phi_1}_{\Delta_1} \longrightarrow g'_2\{^{\Phi_2}_{\Delta_2}}$$

PAR:
$$\frac{\Gamma \rhd g_1\{^{\Phi_1}_{\Delta_1} \longrightarrow g_2\{^{\Phi_2}_{\Delta_2}}{\Gamma \rhd (g_1|g)\{^{\Phi_1}_{\Delta_1} \longrightarrow (g_2|g)\{^{\Phi_2}_{\Delta_2}}$$

Table 4.5: Channel Rules for the Agent-Based Operational Semantics

END:
$$\Gamma \rhd o[\texttt{end}]\{^{\Phi}_{\Delta} \longrightarrow \text{nil}\{^{\Phi}_{\Delta}$$

FRK:
$$\Gamma \rhd o[\texttt{fork}\{p_0\}\ p_1]\{^{\Phi}_{\Delta} \longrightarrow (o[p_0]|o[p_1])\{^{\Phi}_{\Delta}$$

NEW:
$$\Gamma \rhd o[\texttt{new}\ c\colon \text{ChT}\ p]\{^{\Phi}_{\Delta} \longrightarrow o[p\{\!|^{c'}/c|\!\}]\{^{\Phi \cup \{c'\colon \text{ChT}\}}_{\Delta}$$

where c' is a new name.

COM:
$$\Gamma \quad \rhd \quad \begin{pmatrix} o_1[c!\langle v_1, \dots v_n\rangle\ p_1] \mid \\ o_2[c?(r_1\colon T_1, \dots r_n\colon T_n)\ p_2] \end{pmatrix} \begin{cases} \Phi \\ \Delta \end{cases}$$
$$\longrightarrow \quad \begin{pmatrix} o_1[p_1] \mid \\ o_2[p_2\{\!|^{v_1}/r_1, \dots {}^{v_n}/r_n|\!\}] \end{pmatrix} \begin{cases} \Phi \\ \Delta \end{cases}$$

Table 4.6: Object Rules for the Agent-Based Operational Semantics

CRT: $\Gamma \rhd o[\texttt{create}\ o'{:}\,\mathrm{ClT}\ p]\{^{\Phi}_{\Delta} \longrightarrow o[p\{\!|^{o''}/o'|\!\}]\{^{\Phi\cup\{o'':\mathrm{ClT}\}}_{\Delta\cup\{o'':=\emptyset\}}$

where o'' is a new name.

INV:

$$\Gamma \quad \rhd \quad o[o_1.m!\langle v_1,\ldots v_n\rangle?(r_1,\ldots r_{n'})\ p] \ \begin{cases}\Phi\\ \Delta\end{cases}$$

$$\longrightarrow \quad \begin{pmatrix} o[x?(r_1{:}\,T_1,\ldots r_{n'}{:}\,T_{n'})\ p] \mid \\ o_1[p_1\{\!|^{v_1}/s_1,\ldots {}^{v_n}/s_n, {}^{o_1}/\texttt{this}, {}^{x}/\texttt{return}|\!\}] \end{pmatrix} \ \begin{cases}\Phi'\\ \Delta\end{cases}$$

where x is a new name, o_1's class in Φ has a method whose signature has the form $m?(s_1{:}\,S_1,\ldots s_n{:}\,S_n)!\langle d_1{:}\,T_1,\ldots d_{n'}{:}\,T_{n'}\rangle$ and $\Phi' = \Phi \cup \{x{:}\,\texttt{Chan}\langle T_1,\ldots T_{n'}\rangle\}$.

UPD: $\Gamma \rhd o[a!v\ p]\{^{\Phi}_{\Delta} \longrightarrow o[p]\{^{\Phi}_{\Delta'}$

where $o \in \mathrm{Dom}(\Delta)$ and $\Delta'(o')(a') = \Delta(o')(a')$ if $o' \neq o$ or $a' \neq a$ and v otherwise.

ACC: $\Gamma \rhd o[a?r\ p]\{^{\Phi}_{\Delta} \longrightarrow o[p\{\!|^{v}/r|\!\}]\{^{\Phi}_{\Delta}$

where $o \in \mathrm{Dom}(\Delta)$, $\Delta(o)(a)$ is defined and $\Delta(o)(a) = v$.

$$\begin{array}{ll}\mathit{dummy_1}[& \texttt{new cell: Cell} \\ & \texttt{cell.write!}\langle 5\rangle?() \\ & \texttt{cell.read!}\langle\rangle?(\texttt{r}) \\ & \texttt{end} \quad]\end{array} \quad \left\{\begin{array}{l}\emptyset \\ \\ \emptyset\end{array}\right.$$

$\longrightarrow$ $\langle$create a new `Cell` object, choosing name $\mathit{cell_1}\rangle$

$$\begin{array}{ll}\mathit{dummy_1}[& \mathit{cell_1}\texttt{.write!}\langle 5\rangle?() \\ & \mathit{cell_1}\texttt{.read!}\langle\rangle?(\texttt{r}) \\ & \texttt{end} \quad]\end{array} \quad \left\{\begin{array}{l}\{\mathit{cell_1}\colon \texttt{Cell}\} \\ \\ \{\mathit{cell_1} := \emptyset\}\end{array}\right.$$

$\longrightarrow$ $\langle$invoke $\mathit{cell_1}$'s `write` method, choosing return channel $x\rangle$

$$\begin{array}{lll} & \mathit{dummy_1}[& x?() \\ & & \mathit{cell_1}\texttt{.read!}\langle\rangle?(\texttt{r}) \\ & & \texttt{end} \quad] \\ | & \mathit{cell_1}[& \texttt{contents!}5 \\ & & x!\langle\rangle \\ & & \texttt{end} \quad]\end{array} \quad \left\{\begin{array}{l}\left\{\begin{array}{l}\mathit{cell_1}\colon \texttt{Cell} \\ x\colon \texttt{Chan}\langle\rangle\end{array}\right\} \\ \\ \{\mathit{cell_1} := \emptyset\}\end{array}\right.$$

$\longrightarrow$ $\langle$update the `contents` attribute$\rangle$

$$\begin{array}{lll} & \mathit{dummy_1}[& x?() \\ & & \mathit{cell_1}\texttt{.read!}\langle\rangle?(\texttt{r}) \\ & & \texttt{end} \quad] \\ | & \mathit{cell_1}[& x!\langle\rangle \\ & & \texttt{end} \quad]\end{array} \quad \left\{\begin{array}{l}\left\{\begin{array}{l}\mathit{cell_1}\colon \texttt{Cell} \\ x\colon \texttt{Chan}\langle\rangle\end{array}\right\} \\ \\ \{\mathit{cell_1} := [\texttt{contents} \mapsto 5]\}\end{array}\right.$$

$\longrightarrow$ $\langle$return an acknowledgement$\rangle$

$$\begin{array}{lll} & \mathit{dummy_1}[& \mathit{cell_1}\texttt{.read!}\langle\rangle?(\texttt{r}) \\ & & \texttt{end} \quad] \\ | & \mathit{cell_1}[& \texttt{end} \quad]\end{array} \quad \left\{\begin{array}{l}\left\{\begin{array}{l}\mathit{cell_1}\colon \texttt{Cell} \\ x\colon \texttt{Chan}\langle\rangle\end{array}\right\} \\ \\ \{\mathit{cell_1} := [\texttt{contents} \mapsto 5]\}\end{array}\right.$$

$\longrightarrow$ $\langle$garbage collection$\rangle$

Table 4.7: Reflexive Transitive Closure of the Operational Semantics

$$\Gamma \rhd g\{^{\Phi}_{\Delta} \Longrightarrow g\{^{\Phi}_{\Delta}$$

$$\frac{\Gamma \rhd g_1\{^{\Phi_1}_{\Delta_1} \Longrightarrow g_2\{^{\Phi_2}_{\Delta_2} \qquad \Gamma \rhd g_2\{^{\Phi_2}_{\Delta_2} \longrightarrow g_3\{^{\Phi_3}_{\Delta_3}}{\Gamma \rhd g_1\{^{\Phi_1}_{\Delta_1} \Longrightarrow g_3\{^{\Phi_3}_{\Delta_3}}$$

$$\mathit{dummy_1}\left[\begin{array}{l} \mathit{cell_1}.\mathtt{read}!\langle\rangle?(\mathtt{r}) \\ \mathtt{end} \end{array}\right] \left\{\begin{array}{l} \left\{\begin{array}{l} \mathit{cell_1}{:}\,\mathtt{Cell} \\ x{:}\,\mathtt{Chan}\langle\rangle \end{array}\right\} \\ \\ \{\mathit{cell_1} := [\mathtt{contents} \mapsto 5]\} \end{array}\right.$$

The Reflexive Transitive Closure of the Reduction Relation

We generalise the one-step system to its reflexive transitive closure "$\Longrightarrow$", which allows us to describe sequences of behaviour. It is defined by the two rules in Table 4.7.

4.2 Configuration-Based Dynamic System

The *configuration-based dynamic system* offers an alternative to the global internal view of systems given by the agent-based dynamic system. Here, a running program is represented by an expression which has a hierarchical structure carrying local state and local type information. The operational semantics of this dynamic system is a labelled transition relation which is capable of describing the external interactive behaviour of a system as well as the internal reactive behaviours of subsystems.

4.2.1 Syntax

Syntax of Configurations

Syntactically, *configuration expressions*, or just *configurations*, are like nested agent configurations. Their basic form is that of an agent but they can also carry local type and state dictionaries.

$$
\begin{array}{rcll}
K & ::= & \mathrm{Nil} & \text{null configuration} \\
 & | & o[p] & \text{agent} \\
 & | & K_0 \mid K_1 & \text{parallel configurations} \\
 & | & K\{^{\Phi}_{\Delta} & \text{scoped configuration}
\end{array}
$$

The scoped form, $K\{^{\Phi}_{\Delta}$, causes occurrences of the names in $\mathrm{Dom}(\Phi)$ to become bound in both K and in Δ. The free names of a configuration can be found using the following function.

$$
\begin{array}{lcl}
\mathrm{FN}(\mathrm{Nil}) & = & \emptyset \\
\mathrm{FN}(o[p]) & = & \mathrm{FN}(p) \cup \{o\} \\
\mathrm{FN}(K_0 \mid K_1) & = & \mathrm{FN}(K_0) \cup \mathrm{FN}(K_1) \\
\mathrm{FN}(K\{^{\Phi}_{\Delta}) & = & (\mathrm{FN}(K) \cup \mathrm{FN}(\Delta)) \smallsetminus \mathrm{Dom}(\Phi)
\end{array}
$$

Rather than having two global dictionaries which manage type and state information respectively, we can now have many at various positions in a configuration expression. A configuration is the *manager* of an object's state when it contains a state dictionary whose domain contains that object's name and the name is free in the configuration. Note that we don't consider a configuration to be the manager of its locally bound objects (some proper sub-configuration will be their manager). The following function determines

what object state is being managed by a specific configuration.

$$
\begin{aligned}
\mathrm{STATE}(\mathrm{Nil}) &= \emptyset \\
\mathrm{STATE}(o[p]) &= \emptyset \\
\mathrm{STATE}(K_0|K_1) &= \mathrm{STATE}(K_0) \cup \mathrm{STATE}(K_1) \\
\mathrm{STATE}(K\{^{\Phi}_{\Delta}) &= (\mathrm{STATE}(K) \cup \mathrm{Dom}(\Delta)) \setminus \mathrm{Dom}(\Phi)
\end{aligned}
$$

Lemma 4.2.1 *For a configuration K, $\mathrm{STATE}(K) \subseteq \mathrm{FN}(K)$.*

Given that there is no one manager for state and type information, a concern over the well-formedness of a configuration may arise. The two main problems that may occur are, firstly, that two parallel or nested configurations may be managing state for the same object and, secondly, that a local object may have no state manager at all. The following predicate ensures that a configuration is well-formed.

$$
\begin{aligned}
\mathrm{WF}(\mathrm{Nil}) &= \mathrm{true} \\
\mathrm{WF}(o[p]) &= \mathrm{true} \\
\mathrm{WF}(K_0|K_1) &= \left(\begin{array}{ll} & \mathrm{WF}(K_0) \\ \wedge & \mathrm{WF}(K_1) \\ \wedge & (\mathrm{STATE}(K_0) \pitchfork \mathrm{STATE}(K_1)) \end{array} \right) \\
\mathrm{WF}(K\{^{\Phi}_{\Delta}) &= \left(\begin{array}{ll} & \mathrm{WF}(K) \\ \wedge & (\mathrm{STATE}(K) \pitchfork \mathrm{Dom}(\Delta)) \\ \wedge & (\mathrm{OBJECTS}(\Phi) \subseteq (\mathrm{STATE}(K) \cup \mathrm{Dom}(\Delta))) \end{array} \right)
\end{aligned}
$$

A *configuration system* consists of a configuration in the context of a definition set and a global type dictionary. We write this as $\Gamma, \Phi \blacktriangleright K$.

An example of a well-formed configuration is the following:

$$\left(o_2[x?(\mathtt{r})\ \mathtt{end}] \begin{cases} \{o_2 \colon \mathtt{Cell_Client}\} \\ \{o_2 := \emptyset\} \end{cases} \;\Big\|\; \left(\begin{array}{l} \quad o_1[\mathtt{contents?r}\ x!\langle\mathtt{r}\rangle\ \mathtt{end}] \\ \mid\ o_1[\mathtt{end}] \\ \mid\ \mathrm{Nil} \end{array} \begin{cases} \emptyset \\ \{o_1 := [\mathtt{contents} \mapsto 5]\} \end{cases} \right) \right) \begin{cases} \left\{ x \colon \mathtt{Chan}\langle\mathtt{Long}\rangle \right\} \\ \emptyset \end{cases}$$

Its set of free names and its state are both $\{o_1\}$. If we wished to make this configuration ill-formed we could remove the state manager for the local object o_2 or, alternatively, add another state assignment for the object o_1 into the leftmost or rightmost state dictionary. Say Γ is the definition set given in Section 3.1.6 and K is the configuration above. An example of a configuration system would be

$$\Gamma, \{o_1 \colon \mathtt{Cell}\} \blacktriangleright K$$

The object name o_1 would be considered global.

Initial System Convention

We take the same approach to starting the system off as we did with the agent-based dynamic system. Given that g_Γ is the initial agent of a definition set Γ and that $\mathrm{OBJECTS}(\Phi) = \emptyset$, then

$$\Gamma, \Phi \blacktriangleright g_\Gamma$$

is an *initial configuration system* of Γ.

4.2.2 Renaming

In the agent-based dynamic system, all entities (except local names in agent code) are uniquely named. Freshness side conditions on the semantics ensure that all values created in the system are given new names, so variable capture can never occur. A naive approach to substitution and naming would therefore be sufficient to manage names for the agent-based dynamic system.

The configuration-based dynamic system gives up global notions, so we lose this nice property and we are forced to deal with the typical problems associated with bound names. For the purpose of managing the variable capture issue, a standard technique would suffice. However, there is one further requirement we wish to put on our naming system: we will want to view an agent configuration as an instance of a configuration.

As we indicated in Section 3.4, the two most common approaches to naming either use α-equivalence classes of terms instead of terms themselves or abandon names altogether. We cannot use these techniques for the following reason: if we view an agent configuration as an instance of a configuration, all its names would be considered bound. Since the operational semantics we apply to agent configurations is an unlabelled relation, the names in the system are vitally important and should not be changed or abstracted away. For example, the reduction relation would lose much of its descriptive power if an object could change name across the reduction arrow. The approach we take to naming means the choice of bound names in an agent configuration can be preserved but still supports correct name management for configurations.

In one sense, our management of names could be called traditional; we explicitly change bound names to avoid name clashes. Recently, the consequences of having local bound names in syntax has come under scrutiny and we give a brief consideration of some of these new approaches here.

Related Work on Names

The ν-calculus [PS93b] and $\lambda\nu$ [Ode94] are extensions of the λ-calculus designed to study the effect of local state on reasoning. The key property of state in both these systems is that it has a name and these names can be compared. As usual, both approaches identify terms which differ only in the choices of bound variables and bound names.

Gabbay and Pitts [GP99] consider bound names in systems more general than just extensions of the λ-calculus. Using Fraenkel-Mostoski set theory they study set-like structures which have a notion of free names (called *finite support*) attached and an associated concept of renaming based on permutations. One advantage of their system is that notions of free names, bound names, renaming and fresh names are directly accessible. Also, there is no need to take the quotient space of terms over α-equivalence classes, so induction and recursion over terms is unhindered[5]. Moreover, they propose a reasoning system, Nominal Logic [Pit01], and a language, FreshML [PG00], for implementing systems with local names.

Kohei Honda's *rooted p-sets* [Hon00] offer an approach similar to Gabbay and Pitts'. A rooted p-set is a set of structures with associated free name and renaming operations. Honda's aim is "to articulate the common abstract structures of processes with names in diverse process formalisms". Within his framework, operations in process algebra can be seen as homomorphisms from product rooted p-sets to rooted p-sets.

Fiore, Plotkin and Turi [FPT99] provide a (categorical) algebraic view of syntax with variable binding. From an algebraic point of view, abstract

[5] If we take a quotient over a set of inductively defined terms, the resulting equivalence classes are no longer inductively defined structures. This fact is often ignored when using α-equivalence to avoid issues of naming.

syntax can be regarded as the initial algebra of its constructors. Here, they replace algebras over sets with *binding algebras over variable sets*. Their system seems to offer some of the same properties of the two previous approaches.

The approaches of [GP99], [Hon00] and [FPT99] may offer an alternative to our technique. Unlike other approaches, such as de Bruijn notation or α-equivalence classes, names in their system are not abstracted away — they are present but handled in a logical way. The three systems define general frameworks in which results about structures with bound names can be proved in general. It is possible that many of the results we need to make individually explicit might, in their systems, simply be instances of the general properties of structures.

The Action of a Renaming on Configurations

A renaming acts on a configuration as follows:

$$\begin{array}{lcl} \mathrm{Nil}\rho & = & \mathrm{Nil} \\ o[p]\rho & = & (o\rho)[p\rho] \\ (K_1|K_2)\rho & = & (K_1\rho|K_2\rho) \\ (K\{^{\Phi}_{\Delta})\rho & = & K\delta\rho\{^{\Phi\delta\rho}_{\Delta\delta\rho} \end{array}$$

where, in the last case, δ shoves $\mathrm{Dom}(\Phi) \cap \mathrm{Change}(\rho)$.

Lemma 4.2.2 *For a configuration K and renaming ρ,* B.3.1 p254

$$\mathrm{FN}(K\rho) = \mathrm{FN}(K)\rho$$

Lemma 4.2.3 *For a configuration K and a renaming ρ,* B.3.2 p254

$$\mathrm{STATE}(K\rho) = \mathrm{STATE}(K)\rho$$

Lemma 4.2.4 *For a configuration K and a renaming ρ,* B.3.3 p255

$$\mathrm{WF}(K\rho) \Leftrightarrow \mathrm{WF}(K)$$

4.2.3 Equivalence

We give two equivalence systems for configurations: α-equivalence and structural equivalence.

α-Equivalence

The α-equivalence relation between two configurations, K_1 and K_2, written $K_1 \equiv_\alpha K_2$, inherits α-equivalence for code and also allows the bound names from the type dictionary of a scoped configuration $K\{^{\Phi}_{\Delta}$ to be changed. It is defined by the seven rules in Table 4.8.

The three rules, EQV-RFL, EQV-SYM and EQV-TRN, mean that it is an equivalence relation. The two rules, CONG-PAR and CONG-SCP, mean that it is a congruence relation over configurations. By the rule, ALPHA-COD, it inherits code α-equivalence. The last rule, CHNG-SCP, mean that the names bound by a local type dictionary may be changed.

Configuration α-equivalence has the following properties.

Lemma 4.2.5 *For two configurations K_1 and K_2,* B.3.4 p257

$$K_1 \equiv_\alpha K_2 \Rightarrow \mathrm{FN}(K_1) = \mathrm{FN}(K_2)$$

Lemma 4.2.6 *For two configurations K_1 and K_2,* B.3.5 p257

$$K_1 \equiv_\alpha K_2 \Rightarrow \mathrm{STATE}(K_1) = \mathrm{STATE}(K_2)$$

Table 4.8: Rules for Configuration α-Equivalence

EQV-RFL:	$K \equiv_\alpha K$
EQV-SYM:	$\dfrac{K_1 \equiv_\alpha K_2}{K_2 \equiv_\alpha K_1}$
EQV-TRN:	$\dfrac{K_1 \equiv_\alpha K_2 \qquad K_2 \equiv_\alpha K_3}{K_1 \equiv_\alpha K_3}$
CONG-PAR:	$\dfrac{K_1 \equiv_\alpha K_2 \qquad K_1' \equiv_\alpha K_2'}{(K_1 \mid K_1') \equiv_\alpha (K_2 \mid K_2')}$
CONG-SCP:	$\dfrac{K_1 \equiv_\alpha K_2}{K_1\{^{\Phi}_{\Delta} \equiv_\alpha K_2\{^{\Phi}_{\Delta}}$
ALPHA-COD:	$\dfrac{p_1 \equiv_\alpha p_2}{o[p_1] \equiv_\alpha o[p_2]}$
CHNG-SCP:	$K\{^{\Phi}_{\Delta} \equiv_\alpha K\rho\{^{\Phi\rho}_{\Delta\rho}$

where ρ is any renaming such that $\text{FN}(K\{^{\Phi}_{\Delta}) \pitchfork \text{Change}(\rho)$.

B.3.6 p258

Lemma 4.2.7 *For two configurations K_1 and K_2,*

$$K_1 \equiv_\alpha K_2 \Rightarrow \mathrm{WF}(K_1) \Leftrightarrow \mathrm{WF}(K_2)$$

B.3.7 p259

Lemma 4.2.8 *For a configurations K and two renamings ρ_1 and ρ_2,*

$$K\rho_1\rho_2 \equiv_\alpha K(\rho_2 \circ \rho_1)$$

B.3.8 p261

Lemma 4.2.9 *For two configurations K_1 and K_2, and a renaming ρ*

$$K_1 \equiv_\alpha K_2 \Rightarrow K_1\rho \equiv_\alpha K_1\rho$$

Structural Equivalence

The structural equivalence relation between two configurations, K_1 and K_2, written $K_1 \stackrel{\blacktriangleright}{\equiv} K_2$, allows the restructuring of configurations and also incorporates α-equivalence. It is defined by the twelve rules in Table 4.9.

The three rules, EQV-RFL, EQV-SYM and EQV-TRN, mean that $\stackrel{\blacktriangleright}{\equiv}$ is an equivalence relation. The two rules, CONG-PAR and CONG-SCP, mean that it is a congruence relation. The three rules, PAR-ID, PAR-ASS and PAR-ABL, mean that, with respect to $\stackrel{\blacktriangleright}{\equiv}$, $|$ forms an Abelian monoid. By the ALPHA rule, it inherits α-equivalence. The EMPTY rule allows empty dictionaries to be removed.

Its two most important rules are the flatten rule and the scope extrusion rule, both of which allow the dictionaries in a scoped configuration to be repositioned. The FLATTEN rule flattens the structure of nested dictionaries and the EXTRUDE rule allows local scope to be extended over other configurations. EXTRUDE is named after the scope extrusion rule of the π-calculus.

Table 4.9: Rules for Configuration Structural Equivalence

EQV-RFL: $K \stackrel{\blacktriangleright}{\equiv} K$

EQV-SYM: $\dfrac{K_1 \stackrel{\blacktriangleright}{\equiv} K_2}{K_2 \stackrel{\blacktriangleright}{\equiv} K_1}$

EQV-TRN: $\dfrac{K_1 \stackrel{\blacktriangleright}{\equiv} K_2 \qquad K_2 \stackrel{\blacktriangleright}{\equiv} K_3}{K_1 \stackrel{\blacktriangleright}{\equiv} K_3}$

CONG-PAR: $\dfrac{K_1 \stackrel{\blacktriangleright}{\equiv} K_2 \qquad K_1' \stackrel{\blacktriangleright}{\equiv} K_2'}{(K_1|K_1') \stackrel{\blacktriangleright}{\equiv} (K_2|K_2')}$

CONG-SCP: $\dfrac{K_1 \stackrel{\blacktriangleright}{\equiv} K_2}{K_1\{^{\Phi}_{\Delta} \stackrel{\blacktriangleright}{\equiv} K_2\{^{\Phi}_{\Delta}}$

PAR-ID: $(K|\mathrm{Nil}) \stackrel{\blacktriangleright}{\equiv} K$

PAR-ASS: $((K_1|K_2)|K_3) \stackrel{\blacktriangleright}{\equiv} (K_1|(K_2|K_3))$

PAR-ABL: $(K_1|K_2) \stackrel{\blacktriangleright}{\equiv} (K_2|K_1)$

ALPHA: $\dfrac{K_1 \equiv_\alpha K_2}{K_1 \stackrel{\blacktriangleright}{\equiv} K_2}$

EMPTY: $K\{^{\emptyset}_{\emptyset} \stackrel{\blacktriangleright}{\equiv} K$

FLATTEN: $K\{^{\Phi_1}_{\Delta_1}\{^{\Phi_2}_{\Delta_2} \stackrel{\blacktriangleright}{\equiv} K\{^{\Phi_1 \cup \Phi_2}_{\Delta_1 \cup \Delta_2}$

where $\mathrm{Dom}(\Phi_1) \pitchfork \mathrm{Dom}(\Phi_2)$, $\mathrm{Dom}(\Delta_1) \pitchfork \mathrm{Dom}(\Delta_2)$

and $\mathrm{Dom}(\Phi_1) \pitchfork \mathrm{FN}(\Delta_2)$.

EXTRUDE: $K_1|(K_2\{^{\Phi}_{\Delta}) \stackrel{\blacktriangleright}{\equiv} (K_1|K_2)\{^{\Phi}_{\Delta}$

where $\mathrm{FN}(K_1) \pitchfork \mathrm{Dom}(\Phi)$.

The first two side conditions of the flatten rule prevents the merging of dictionaries with overlapping domains. The third side condition prevents any free names of Δ_2 becoming bound by Φ_1. The side condition of the scope extrusion rule prevents free names in K_1 becoming bound by Φ. The real work of the $\stackrel{\blacktriangleright}{\equiv}$ system is done by these two rules, yet it is often the case that an application of these rules will need to be preceded by some use of the $\equiv_\alpha$ relation. This is why we do not consider a structural equivalence separate from α-equivalence, as we did for agents.

We can show that $\stackrel{\blacktriangleright}{\equiv}$ has the following properties.

B.3.13 p265

Lemma 4.2.10 *For two configurations K_1 and K_2,*

$$K_1 \stackrel{\blacktriangleright}{\equiv} K_2 \Rightarrow \mathrm{FN}(K_1) = \mathrm{FN}(K_2)$$

B.3.14 p267

Lemma 4.2.11 *For two configurations K_1 and K_2,*

$$K_1 \stackrel{\blacktriangleright}{\equiv} K_2 \Rightarrow \mathrm{STATE}(K_1) = \mathrm{STATE}(K_2)$$

B.3.15 p268

Lemma 4.2.12 *For two configurations K_1 and K_2,*

$$K_1 \stackrel{\blacktriangleright}{\equiv} K_2 \Rightarrow \mathrm{WF}(K_1) \Leftrightarrow \mathrm{WF}(K_2)$$

B.3.16 p272

Lemma 4.2.13 *For two configurations K_1 and K_2, and a renaming ρ*

$$K_1 \stackrel{\blacktriangleright}{\equiv} K_2 \Rightarrow K_1\rho \stackrel{\blacktriangleright}{\equiv} K_1\rho$$

4.2.4 Semantics

We now give an operational semantics for the configuration-based dynamic system by defining a *labelled transition system* [Plo81] for configurations. This describes the behaviour of a configuration K in the context of a definition set Γ and a global type dictionary Φ. Unlike the reduction relation

we defined for the agent-based dynamic system, this semantics can describe how a subsystem interacts with its environment.

The labelled transition system is a directed graph whose nodes are configurations and whose edges (called *transitions*) are the different actions their source configuration can perform. There are five kinds of transition:

1. The first kind asserts that the configuration K can perform an internal (unobservable) action and reach configuration K'.

$$\Gamma, \Phi \blacktriangleright K \longrightarrow K'$$

2. Next, we have a form that asserts that K can send values $v_1, \ldots v_n$ on channel c.

$$\Gamma, \Phi \blacktriangleright K \xrightarrow{c!\langle v_1,\ldots v_n\rangle} K'$$

3. The third kind asserts that K can receive values $v_1, \ldots v_n$ off channel c.

$$\Gamma, \Phi \blacktriangleright K \xrightarrow{c?\langle v_1,\ldots v_n\rangle} K'$$

4. The fourth kind of action asserts that the configuration K can issue an invocation on method m of object o with values $v_1, \ldots v_n$, where $x{:}T$ is the name and type of a newly created return channel.

$$\Gamma, \Phi \blacktriangleright K \xrightarrow[x:T]{o.m!\langle v_1,\ldots v_n\rangle} K'$$

5. The last form of transition asserts that the K can accept an invocation on method m of object o with values $v_1, \ldots v_n$, where $x{:}T$ is the name and type of a newly created return channel.

$$\Gamma, \Phi \blacktriangleright K \xrightarrow[x:T]{o.m?\langle v_1,\ldots v_n\rangle} K'$$

The labelled transition system is defined by the sixteen rules in tables 4.10, 4.11, 4.12 and 4.13.

There are four general purpose rules given in Table 4.10: the EQV-LFT rule, the EQV-RHT rule, the PAR rule and the SCP rule. The side condition on the SCP rule prevents local (and hence private) information from being exposed.

The six rules in Table 4.11 describe how a single agent can evolve silently in a minimal local environment. Reasoning with structural equivalence is needed to use these rules for more complex systems. The rules are: END, FRK, NEW, CRT, UPD and ACC.

The four rules in Table 4.12 define the labelled transitions. These can be thought of as describing the willingness of a subsystem to participate in a communication. The SND and RCV rules deal with channel based communication. The next two rules are the INV and ACT rules. These are the only rules which involve the choice of a new name, for which a typing is put beneath the transition arrow. This is essentially a free name but its free status is local to the subsystem whose behaviour we are considering (see below).

The two rules in Table 4.13 describe how a system consisting of two parallel subsystems can silently evolve when those subsystems interact. The COM rule supports channel-based communication between two subsystems which can perform appropriate send and receive actions. The INV rule supports method invocation between subsystems which can perform appropriate invocation and activation actions. When these subsystems are brought together, their local name becomes bound in the resulting system[6].

[6]The behaviour of this local name is reminiscent of a bound-output action in the π-calculus [SW01].

Table 4.10: General Rules for the Labelled Transition System

Rule	
EQV-LFT:	$$\frac{K_1 \overset{\blacktriangleright}{\equiv} K_2 \qquad \Gamma, \Phi \blacktriangleright K_1 \xrightarrow[l_2]{l_1} K_1'}{\Gamma, \Phi \blacktriangleright K_2 \xrightarrow[l_2]{l_1} K_1'}$$
EQV-RHT:	$$\frac{K_1' \overset{\blacktriangleright}{\equiv} K_2 \qquad \Gamma, \Phi \blacktriangleright K_1 \xrightarrow[l_2]{l_1} K_1'}{\Gamma, \Phi \blacktriangleright K_1 \xrightarrow[l_2]{l_1} K_2}$$
PAR:	$$\frac{\Gamma, \Phi \blacktriangleright K_1 \xrightarrow[l_2]{l_1} K_1'}{\Gamma, \Phi \blacktriangleright K_1 \vert K_2 \xrightarrow[l_2]{l_1} K_1' \vert K_2}$$
SCP:	$$\frac{\Gamma, \Phi \cup \Phi' \blacktriangleright K \xrightarrow[l_2]{l_1} K'}{\Gamma, \Phi \blacktriangleright K\{^{\Phi'}_{\Delta} \xrightarrow[l_2]{l_1} K'\{^{\Phi'}_{\Delta}}$$

where $\mathrm{FN}(l_1), \mathrm{FN}(l_2) \notin \mathrm{Dom}(\Phi')$.

Table 4.11: Local Rules for the Labelled Transition System

Rule	
END:	$\Gamma, \Phi \blacktriangleright o[\texttt{end}] \longrightarrow \mathrm{Nil}$
FRK:	$\Gamma, \Phi \blacktriangleright o[\texttt{fork}\{p_0\}\ p_1] \longrightarrow o[p_0] \vert o[p_1]$
NEW:	$\Gamma, \Phi \blacktriangleright o[\texttt{new}\ c\colon \mathrm{ChT}\ p] \longrightarrow o[p]\{^{\{c:\mathrm{ChT}\}}_{\emptyset}$
CRT:	$\Gamma, \Phi \blacktriangleright o[\texttt{create}\ o'\colon \mathrm{ClT}\ p] \longrightarrow o[p]\{^{\{o':\mathrm{ClT}\}}_{\{o':=\emptyset\}}$
UPD:	$\Gamma, \Phi \blacktriangleright o[a!v\ p]\{^{\emptyset}_{\{o:=f\}} \longrightarrow o[p]\{^{\emptyset}_{\{o:=f\dagger[a\mapsto v]\}}$
ACC:	$\Gamma, \Phi \blacktriangleright o[a?r\ p]\{^{\emptyset}_{\{o:=f\}} \longrightarrow o[p\{\vert^{v}/r\vert\}]\{^{\emptyset}_{\{o:=f\}}$

where $a \in \mathrm{Dom}(f)$ and $f(a) = v$.

Table 4.12: Labelled Rules for the Labelled Transition System

SND:

$$\Gamma, \Phi \blacktriangleright o[c!\langle v_1, \ldots v_n\rangle\ p] \xrightarrow{c!\langle v_1,\ldots v_n\rangle} o[p]$$

RCV:

$$\Gamma, \Phi \blacktriangleright o[c?(r_1\colon T_1, \ldots r_n\colon T_n)\ p] \xrightarrow{c?\langle v_1,\ldots v_n\rangle} o[p\{\!|^{v_1}/r_1, \ldots {}^{v_n}/r_n|\!\}]$$

INV:

$$\Gamma, \Phi \blacktriangleright$$

$$o[o'.m!\langle v_1, \ldots v_n\rangle?(r_1, \ldots r_{n'})\ p] \xrightarrow[x:\mathtt{Chan}\langle T_1,\ldots T_{n'}\rangle]{o'.m!\langle v_1,\ldots v_n\rangle} o[x?(r_1\colon T_1, \ldots r_{n'}\colon T_{n'})\ p]$$

where x is a new name and o''s class in Φ has a method whose signature has the form $m?(s_1\colon S_1, \ldots s_n\colon S_n)!\langle d_1\colon T_1, \ldots d_{n'}\colon T_{n'}\rangle\{p'\}$.

ACT:

$$\Gamma, \Phi \blacktriangleright \mathrm{Nil} \xrightarrow[x:\mathtt{Chan}\langle S_1,\ldots S_{n'}\rangle]{o.m?\langle v_1,\ldots v_n\rangle} o[p\{\!|^{v_1}/r_1, \ldots {}^{v_1}/r_n, {}^{o}/\mathtt{this}, {}^{x}/\mathtt{return}|\!\}]$$

where x is a new name and o's class in Φ has a method whose signature has the form $m?(r_1\colon T_1, \ldots r_n\colon T_n)!\langle d_1\colon S_1, \ldots d_{n'}\colon S_{n'}\rangle\{p\}$.

Table 4.13: Interaction Rules for the Labelled Transition System

COM:

$$\frac{\Gamma, \Phi \blacktriangleright K_1 \xrightarrow{c!\langle v_1,\ldots v_n\rangle} K_1' \qquad \Gamma, \Phi \blacktriangleright K_2 \xrightarrow{c?\langle v_1,\ldots v_n\rangle} K_2'}{\Gamma, \Phi \blacktriangleright K_1|K_2 \longrightarrow K_1'|K_2'}$$

MTH:

$$\frac{\Gamma, \Phi \blacktriangleright K_1 \xrightarrow[x:\mathrm{ChT}]{o.m!\langle v_1,\ldots v_n\rangle} K_1' \qquad \Gamma, \Phi \blacktriangleright K_2 \xrightarrow[x:\mathrm{ChT}]{o.m?\langle v_1,\ldots v_n\rangle} K_2'}{\Gamma, \Phi \blacktriangleright K_1|K_2 \longrightarrow (K_1'|K_2')\{_{\emptyset}^{\{x:\mathrm{ChT}\}}}$$

Example of the Labelled Transition System

We now give an example of how the labelled transition system is used. In what follows we will omit the definition set and global type dictionary from statements. This example show how the configuration

$$\begin{array}{ll} & \left(o_2[x?(\mathtt{r})\ \mathtt{end}]\{^{\{o_2:\mathtt{Cell_Client}\}}_{\{o_2:=\emptyset\}}\right) \\ | & \left(o_1[\mathtt{contents}?\mathtt{r}\ x!\langle\mathtt{r}\rangle\ \mathtt{end}]\{^{\emptyset}_{\{o_1:=[\mathtt{contents}\mapsto 5]\}}\right) \end{array}$$

can reach the configuration

$$\left(o_2[\mathtt{end}]\{^{\{o_2:\mathtt{Cell_Client}\}}_{\{o_2:=\emptyset\}}\right) \mid \left(o_1[\mathtt{end}]\{^{\emptyset}_{\{o_1:=[\mathtt{contents}\mapsto 5]\}}\right)$$

with two unlabelled transitions.

The first transition is due to the agent at object o_1 accessing the attribute `contents`:

$$\begin{array}{ll} & o_1[\mathtt{contents}?\mathtt{r}\ x!\langle\mathtt{r}\rangle\ \mathtt{end}]\{^{\emptyset}_{\{o_1:=[\mathtt{contents}\mapsto 5]\}} \\ \longrightarrow & \langle \text{ACC} \rangle \\ & o_1[x!\langle 5\rangle\ \mathtt{end}]\{^{\emptyset}_{\{o_1:=[\mathtt{contents}\mapsto 5]\}} \end{array}$$

So,

$$\begin{array}{ll} & \left(o_2[x?(\mathtt{r})\ \mathtt{end}]\{^{\{o_2:\mathtt{Cell_Client}\}}_{\{o_2:=\emptyset\}}\right) \\ & |\quad \left(o_1[\mathtt{contents}?\mathtt{r}\ x!\langle\mathtt{r}\rangle\ \mathtt{end}]\{^{\emptyset}_{\{o_1:=[\mathtt{contents}\mapsto 5]\}}\right) \\ \longrightarrow & \langle \text{using the above with EQV-LFT, EQV-RHT and PAR} \rangle \\ & \left(o_2[x?(\mathtt{r})\ \mathtt{end}]\{^{\{o_2:\mathtt{Cell_Client}\}}_{\{o_2:=\emptyset\}}\right) \\ & |\quad \left(o_1[x!\langle 5\rangle\ \mathtt{end}]\{^{\emptyset}_{\{o_1:=[\mathtt{contents}\mapsto 5]\}}\right) \end{array}$$

The second transition will be a communication on the channel x. The following two proof trees show that the two configurations can perform the appropriate send and receive actions:

$$\frac{o_1[x!\langle 5\rangle\ \mathtt{end}] \xrightarrow{x!\langle 5\rangle} o_1[\mathtt{end}]}{o_1[x!\langle 5\rangle\ \mathtt{end}]\{^{\emptyset}_{\{o_1:=[\mathtt{contents}\mapsto 5]\}} \xrightarrow{x!\langle 5\rangle} o_1[\mathtt{end}]\{^{\emptyset}_{\{o_1:=[\mathtt{contents}\mapsto 5]\}}}$$

and

$$\frac{o_2[x?(\mathtt{r})\ \mathtt{end}] \xrightarrow{x?\langle 5\rangle} o_2[\mathtt{end}]}{o_2[x?(\mathtt{r})\ \mathtt{end}]\{^{\{o_2:\mathtt{Cell_Client}\}}_{\{o_2:=\emptyset\}} \xrightarrow{x?\langle 5\rangle} o_2[\mathtt{end}]\{^{\{o_2:\mathtt{Cell_Client}\}}_{\{o_2:=\emptyset\}}}$$

Thus, the two configurations can communicate as follows:

$$\left(o_1[x!\langle 5\rangle\ \mathtt{end}]\{^{\emptyset}_{\{o_1:=[\mathtt{contents}\mapsto 5]\}}\right) \mid \left(o_2[x?(\mathtt{r})\ \mathtt{end}]\{^{\{o_2:\mathtt{Cell_Client}\}}_{\{o_2:=\emptyset\}}\right)$$
$$\longrightarrow \langle\text{using above with COM}\rangle$$
$$\left(o_1[\mathtt{end}]\{^{\emptyset}_{\{o_1:=[\mathtt{contents}\mapsto 5]\}}\right) \mid \left(o_2[\mathtt{end}]\{^{\{o_2:\mathtt{Cell_Client}\}}_{\{o_2:=\emptyset\}}\right)$$

Properties of the Labelled Transition System

The next two lemmas show that the labelled transition system preserves the state and well-formedness of a configuration.

B.3.17 p278

Lemma 4.2.14 *Say K_1 and K_2 are configurations such that*

$$\Gamma, \Phi \blacktriangleright K \xrightarrow[l_2]{l_1} K'$$

for any labels l_1 and l_2. Then

$$\text{STATE}(K) = \text{STATE}(K')$$

B.3.18 p279

Lemma 4.2.15 *Say K_1 and K_2 are configurations such that*

$$\Gamma, \Phi \blacktriangleright K \xrightarrow[l_2]{l_1} K'$$

for any labels l_1 and l_2. Then

$$\text{WF}(K) \Rightarrow \text{WF}(K')$$

Table 4.14: Rules for Experiments

$$\Gamma, \Phi \blacktriangleright K \Longrightarrow K$$

$$\frac{\Gamma, \Phi \blacktriangleright K_1 \Longrightarrow K_2 \qquad \Gamma, \Phi \blacktriangleright K_2 \longrightarrow K_3}{\Gamma, \Phi \blacktriangleright K_1 \Longrightarrow K_3}$$

$$\frac{\Gamma, \Phi \blacktriangleright K_1 \Longrightarrow K_2 \qquad \Gamma, \Phi \blacktriangleright K_2 \xrightarrow[l_2]{l_1} K_3}{\Gamma, \Phi \blacktriangleright K_1 \overset{l_1}{\underset{l_2}{\Longrightarrow}} K_3}$$

$$\frac{\Gamma, \Phi \blacktriangleright K_1 \overset{L_1}{\underset{L_2}{\Longrightarrow}} K_2 \qquad \Gamma, \Phi \blacktriangleright K_2 \longrightarrow K_3}{\Gamma, \Phi \blacktriangleright K_1 \overset{L_1}{\underset{L_2}{\Longrightarrow}} K_3}$$

$$\frac{\Gamma, \Phi \blacktriangleright K_1 \overset{L_1}{\underset{L_2}{\Longrightarrow}} K_2 \qquad \Gamma, \Phi \blacktriangleright K_2 \xrightarrow[l_2]{l_1} K_3}{\Gamma, \Phi \blacktriangleright K_1 \overset{L_1\ l_1}{\underset{L_2\ l_2}{\Longrightarrow}} K_3}$$

Experiments

The labelled transition system describes how a configuration can perform a single operation. We also provide a generalisation of that system in order to allow us to describe sequences of behaviour. These sequences are called *experiments* because they can be used to test the possible visible behaviour of a configuration.

The rules of this system are given in Table 4.14. To define the rules, we use the following notation: l_1 is either a send, receive, invoke or activate label and l_2 is a return channel typing or a blank space. L_1 and L_2 are sequences of the above two label types where, in L_2, the blank space occupies a position in the sequence.

4.2.5 Weak Bisimulation

The labelled transition system we have defined provides an operational semantics for Oompa's configuration-based dynamic system. In order to reason usefully about configurations, however, we will need to go further[7]. We will want a relation over configurations to identify those configurations whose behaviour is "the same" in some respect. Unlike the equivalence systems we have already defined, which were primarily concerned with syntactic structure, this will be a *semantic equivalence*.

There are many criteria we could use for this notion of sameness but a suitable starting point would be weak bisimulation [Mil89]. The function of weak bisimulation is to abstract away details of internal behaviour; only observable behaviour is considered. Despite its name, it is quite a strong equivalence and it is unlikely that we would ever want to be more discriminating.

A binary relation $\mathcal{S}$ between configurations is a *weak bisimulation in the context* Γ, Φ if $(K_1, K_2) \in \mathcal{S}$ implies

- whenever $\Gamma, \Phi \blacktriangleright K_1 \xrightarrow[l_2]{l_1} K_1'$ then, for some K_2', $\Gamma, \Phi \blacktriangleright K_2 \xRightarrow[l_2]{l_1} K_2'$ and $(K_1', K_2') \in \mathcal{S}$.
- whenever $\Gamma, \Phi \blacktriangleright K_1 \longrightarrow K_1'$ then, for some K_2', $\Gamma, \Phi \blacktriangleright K_2 \Longrightarrow K_2'$ and $(K_1', K_2') \in \mathcal{S}$.
- whenever $\Gamma, \Phi \blacktriangleright K_2 \xrightarrow[l_2]{l_1} K_2'$ then, for some K_1', $\Gamma, \Phi \blacktriangleright K_1 \xRightarrow[l_2]{l_1} K_1'$ and $(K_1', K_2') \in \mathcal{S}$.

[7] It would also be possible to define a semantic equivalence for the operational semantics of the agent-based dynamic system. As the reductions of that operational semantics are unlabelled, we would need to use a *barbed* equivalence which examines agents to observe their willingness to communicate. We do not consider such an equivalence necessary since the agent-based dynamic system is primarily intended to establish intuitions.

- whenever $\Gamma, \Phi \blacktriangleright K_2 \longrightarrow K_2'$ then, for some K_1', $\Gamma, \Phi \blacktriangleright K_1 \Longrightarrow K_1'$ and $(K_1', K_2') \in \mathcal{S}$.

If there exists a weak bisimulation which contains the pair (K_1, K_2) then we say K_1 and K_2 are *weakly bisimilar in the context* Γ, Φ and write $K_1 \approx_{\Gamma,\Phi} K_2$.

4.3 Relationship

Semantics are what gives meaning to a language and we have given Oompa two (operational) semantics. This suggests an important question: do the two semantics give Oompa programs compatible or incompatible meanings? In this section, we identify a relationship between configuration systems and agent systems and state a result that relates their behaviour. The result assures us that the meanings are compatible.

4.3.1 The Flatten Algorithm

Structural equivalence allows us to restructure a configuration and, in particular, enables us to flatten its structure until it has the form of an agent configuration. Here, we introduce an algorithm which does just that — given a well-formed configuration it returns a structurally equivalent agent configuration. The relationship between configurations and their *flattened form* provides the foundation for the relationship between the configuration-based dynamic system and the agent-based dynamic system.

The *Flatten* algorithm is defined recursively over configurations, so there are four cases:

1. $\text{Flatten}(\text{Nil}) = \text{nil}\{^{\emptyset}_{\emptyset}$

2. $\text{Flatten}(o[p]) = o[p]\{^{\emptyset}_{\emptyset}$

3. $\text{Flatten}(K\{^{\Phi}_{\Delta}) = (g'\delta)\{^{(\Phi'\delta)\cup\Phi}_{(\Delta'\delta)\cup\Delta}$ where

   - $\text{Flatten}(K) = g'\{^{\Phi'}_{\Delta'}$
   - δ shoves $(\text{Dom}(\Phi) \cup \text{FN}(\Delta)) \cap \text{Dom}(\Phi')$

4. $\text{Flatten}(K_1|K_2) = (g_1\delta_1|g_2\delta_2)\{^{(\Phi_1\delta_1)\cup(\Phi_2\delta_2)}_{(\Delta_1\delta_1)\cup(\Delta_2\delta_2)}$ where

   - $\text{Flatten}(K_1) = g_1\{^{\Phi_1}_{\Delta_1}$
   - $\text{Flatten}(K_2) = g_2\{^{\Phi_2}_{\Delta_2}$
   - δ_2 shoves $\text{FN}(g_1\{^{\Phi_1}_{\Delta_1}) \cap \text{Dom}(\Phi_2)$
   - δ_1 shoves $(\text{FN}(g_2\delta_2) \cup \text{Dom}(\Phi_2\delta_2) \cup \text{FN}(\Delta_2\delta_2)) \cap \text{Dom}(\Phi_1)$

Note that our presentation of the algorithm is nondeterministic. Nevertheless, we will treat it as deterministic since we could easily make it so. One obvious way is to put an order on the shoves and insist that the algorithm chooses "the least such shove".

Since state dictionaries are joined without a constraining property that ensures their domains are disjoint, the problem of clashes might suggest itself. In fact, in the proof of Theorem 4.3.1 (in Appendix B on page 282) we see that the dictionaries will be disjoint if the configuration is well-formed. We will restrict our attention to well-formed configurations.

We give an example of a step of the flatten algorithm on following configuration:

$$\left(\begin{array}{l} o[x?(\texttt{r})\ \texttt{end}] \left\{\begin{array}{l}\{o\colon \texttt{Cell_Client}\}\\ \{o := \emptyset\}\end{array}\right. \\ \Bigg|\ \left(\begin{array}{ll} \quad o[\texttt{contents?r}\ x!\langle\texttt{r}\rangle\ \texttt{end}] & \\ \mid\ o[\texttt{end}] & \left\{\begin{array}{l}\emptyset\\ \{o := [\texttt{contents} \mapsto 5]\}\end{array}\right. \\ \mid\ \text{Nil} & \end{array}\right) \end{array}\right)$$

Throughout this example, we will implicitly use the associativity of $|$.

This configuration is of the form $(K_1|K_2)$ so the fourth case of flatten applies. There are two recursive calls, each returning an agent configuration.

$$\text{Flatten}\left(o[x?(\texttt{r})\ \texttt{end}] \quad \begin{cases} \{o\colon \texttt{Cell_Client}\} \\ \{o := \emptyset\} \end{cases} \right)$$

$$= \quad \langle \text{flatten algorithm — no renaming necessary} \rangle$$

$$o[x?(\texttt{r})\ \texttt{end}] \quad \begin{cases} \{o\colon \texttt{Cell_Client}\} \\ \{o := \emptyset\} \end{cases}$$

and

$$\text{Flatten} \left(\begin{array}{ll} & o[\texttt{contents?r}\ x!\langle\texttt{r}\rangle\ \texttt{end}] \\ | & o[\texttt{end}] \\ | & \text{Nil} \end{array} \quad \begin{cases} \emptyset \\ \{o := [\texttt{contents} \mapsto 5]\} \end{cases} \right)$$

$$= \quad \langle \text{flatten algorithm — no renaming necessary} \rangle$$

$$\begin{array}{ll} & o[\texttt{contents?r}\ x!\langle\texttt{r}\rangle\ \texttt{end}] \\ | & o[\texttt{end}] \\ | & \text{nil} \end{array} \quad \begin{cases} \emptyset \\ \{o := [\texttt{contents} \mapsto 5]\} \end{cases}$$

The algorithm requires us to pick two shoves. None of the free names of the first agent configuration occur in the domain of the second agent configuration's type dictionary. Therefore, there are no names that δ_2 need shove. We can let $\delta_2 = 1$.

On the other hand, there is a name free in the second agent configuration's agents and state dictionary which occurs in the domain of the first agent configuration's type dictionary, namely o. We need to shove this name, so we let $\delta_1 = (o\ o_1)$ where o_1 is a new name. The agents of the resulting agent configuration will be:

$$(o[x?(\mathtt{r})\ \mathtt{end}])(o\ o_1)$$
$$\left|\left(\begin{array}{ll} & o[\mathtt{contents?r}\ x!\langle\mathtt{r}\rangle\ \mathtt{end}] \\ | & o[\mathtt{end}] \\ | & \mathrm{Nil}\end{array}\right)1\right.$$

$$\begin{array}{lll} = & \langle\text{apply the renamings and use associativity}\rangle & \\ & & o_1[x?(\mathtt{r})\ \mathtt{end}] \\ & | & o[\mathtt{contents?r}\ x!\langle\mathtt{r}\rangle\ \mathtt{end}] \\ & | & o[\mathtt{end}] \\ & | & \mathrm{Nil}\end{array}$$

The dictionaries of the resulting agent configuration will be:

$$\left\{\begin{array}{l}\{o\colon \mathtt{Cell_Client}\}\,(o\ o_1) \cup (\emptyset)1 \\ \{o := \emptyset\}\,(o\ o_1) \cup \{o := [\mathtt{contents} \mapsto 5]\}\,1\end{array}\right.$$

$$= \quad \langle\text{apply the renamings and join the dictionaries}\rangle$$

$$\left\{\begin{array}{l}\{o_1\colon \mathtt{Cell_Client}\} \\ \{o_1 := \emptyset, o := [\mathtt{contents} \mapsto 5]\}\end{array}\right.$$

Finally, we can give the flattened form of our configuration:

$$\begin{array}{ll} & o_1[x?(\mathtt{r})\ \mathtt{end}] \\ | & o[\mathtt{contents?r}\ x!\langle\mathtt{r}\rangle\ \mathtt{end}] \\ | & o[\mathtt{end}] \\ | & \mathrm{Nil}\end{array} \quad \left\{\begin{array}{l}\{o_1\colon \mathtt{Cell_Client}\} \\ \{o_1 := \emptyset, o := [\mathtt{contents} \mapsto 5]\}\end{array}\right.$$

4.3.2 Properties of Flatten and the Relationship Theorem

It should be clear that agent configurations are syntactically valid configurations. In fact, we will permit agent configurations to be used directly as configurations. Strictly, we should provide an injective function which explicitly denotes this identification. Instead, we will leave the injection implicit.

B.4.1 p282

Theorem 4.3.1 *For a configuration K,*

$$\mathrm{WF}(K) \Rightarrow K \overset{\blacktriangleright}{\equiv} \mathrm{Flatten}(K)$$

B.4.4 p288

Lemma 4.3.2 *For a configuration K and a renaming ρ,*

$$\mathrm{Flatten}(K\rho) \equiv_\alpha \mathrm{Flatten}(K)\rho$$

The Theorems 4.3.3 and 4.3.4 show that the relationship with the flattened form is preserved across $\equiv_\alpha$ and $\overset{\blacktriangleright}{\equiv}$.

B.4.5 p292

Theorem 4.3.3 *Say K_1 and K_2 are configurations such that $K_1 \equiv_\alpha K_2$ and*

$$\begin{aligned} \mathrm{Flatten}(K_1) &= g_1\{^{\Phi_1}_{\Delta_1} \\ \mathrm{Flatten}(K_2) &= g_2\{^{\Phi_2}_{\Delta_2} \end{aligned}$$

Then, for some renaming, ρ, such that $\mathrm{FN}(g_2\{^{\Phi_2}_{\Delta_2}) \pitchfork \mathrm{Change}(\rho)$ *we have*

$$\Phi_2\rho = \Phi_1 \qquad \Delta_2\rho = \Delta_1 \qquad g_2\rho \equiv_\alpha g_1$$

B.4.6 p296

Theorem 4.3.4 *Say K_1 and K_2 are configurations such that $K_1 \overset{\blacktriangleright}{\equiv} K_2$ and*

$$\begin{aligned} \mathrm{Flatten}(K_1) &= g_1\{^{\Phi_1}_{\Delta_1} \\ \mathrm{Flatten}(K_2) &= g_2\{^{\Phi_2}_{\Delta_2} \end{aligned}$$

Then, for some renaming, ρ, such that $\mathrm{FN}(g_2\{^{\Phi}_{\Delta}) \pitchfork \mathrm{Change}(\rho)$ *we have*

$$\Phi_2\rho = \Phi_1 \qquad \Delta_2\rho = \Delta_1 \qquad g_2\rho \overset{\triangleright}{\equiv}_\alpha g_1$$

We now give the main result of this section. Intuitively, it can be thought to say that the internal transitions of a configuration system are realisable within the operational semantics of the agent-based dynamic system. This is a desirable result. Because the agent-based dynamic system was designed

to embody our intentions of how we wish Oompa programs to behave, the result gives us confidence that configuration systems also behave as we would wish.

There are other results we could show that would bring to light different aspects of the relationship between the dynamic systems. We chose this result as the minimum to reassure us of the compatibility of the semantics.

One notable feature of the theorem is that we flatten the scoped configurations, $K_1\{^{\Phi}_{\emptyset}$ and $K_2\{^{\Phi}_{\emptyset}$, rather than just the configurations, K_1 and K_2. This is necessary to ensure all the typing information available in Φ gets rolled into the flattened form.

B.4.11 p308

Theorem 4.3.5 *Say K_1 and K_2 are configurations such that*

$$\Gamma, \Phi \blacktriangleright K_1 \longrightarrow K_2$$

and

$$\begin{aligned} \text{Flatten}(K_1\{^{\Phi}_{\emptyset}) &= g_1\{^{\Phi_1}_{\Delta_1} \\ \text{Flatten}(K_2\{^{\Phi}_{\emptyset}) &= g_2\{^{\Phi_2}_{\Delta_2} \end{aligned}$$

then there is some renaming ρ such that $\text{FN}(g_2\{^{\Phi_2}_{\Delta_2}) \pitchfork \text{Change}(\rho)$ *and*

$$\Gamma \rhd g_1\{^{\Phi_1}_{\Delta_1} \longrightarrow g_2\rho\{^{\Phi_2\rho}_{\Delta_2\rho}$$

4.4 Summary

This chapter set out to define Oompa's two dynamic systems: the agent-based dynamic system and the configuration-based dynamic system. For both of these dynamic system we presented the material in four sections. First we defined their syntax, then some theory of renaming, then their equivalence relations and lastly their semantics. For the configuration-based

dynamic system, we also defined a semantic equivalence called weak bisimulation. The relationship between the two dynamic systems was established by defining an algorithm which converts configurations into structurally equivalent agent configurations. We proved a simulation-like result which shows that this relationship is preserved, up to renaming, by internal actions.

Chapter 5

The Type System

Oompa's operational semantics are defined purely in terms of pattern matching. No limitations are put on the behaviour of an agent beyond syntactic conformance to some rule. As a consequence of this, the operational semantics permit behaviours we never intended and do not want. Many of these behaviours fall into two groups.

An example of the first kind would be an attempt to perform an invocation on an object which lacks a matching method. No rules of the operational semantics will apply to the agent responsible, so we can think of this behaviour as "stalling" the agent. This is an example of a "round hole/square peg" error — a square peg will not fit in a round hole.

An example of the second kind would be the assignment of a temperature reading in Fahrenheit to an attribute intended for temperature in centigrade. Some rule of the operational semantic applies to the agent responsible but the transition leads to the propagation of bad data. This belongs to what we might call a "round hole/finger" error — a finger may actually fit in a round hole but it may then get stuck or electrocuted.

We prevent these misbehaviours by providing Oompa with a *type sys-*

tem [Car96]. Oompa's types, whose syntax we defined in Chapter 3, provide a way of describing the appropriate values a context can accept and the appropriate contexts a value can be used in. The role of Oompa's type system is to identify and exclude those programs whose behaviour will be inconsistent with their types. We will require a definition set to be syntactically correct, well-formed and type safe.

One important aspect of our type system is that it supports subtyping. The types we give to objects are interface types, which describe the names of the methods the objects must support and the arity of their input and output parameter lists. When we mark a context with an interface type, we require that any object used in the context support the methods described by that type. But an object which supports those methods *and others* should also be acceptable. Rather than requiring exact matches between the types of values and contexts, we allow a certain flexibility in the type system. This is formalised by a relation between types called *subtyping*. For Oompa, we define subtyping relations on both interface types and channel types.

Another important feature of our type system is that it supports recursive types. There are two different reasons why we need them. Firstly, without recursive types our channels would be limited to carrying "simpler" values. This severely limits a system's dynamic behaviour, as the most complex channel types in the system would form a static infrastructure — there would be no channels which could be used to transmit their names. Secondly, in an object-oriented system, interface types will often be found to be inter-referential. For example, the following two interfaces have this interdependency:

```
interface A              interface B
{                        {
  m?()!<d:B>               m?()!<d:A>
}                        }
```

Consequently, we acknowledge these two kinds of recursion in our type system. The first kind we consider to be *explicit recursion* and results from the use of the `rec` operator. For example, the type `rec` t.`Chan`$\langle$`Long`$, t\rangle$ which advertises its recursivity. The second kind we consider to be *implicit recursion* and results from the use of class or interface names in a type position, as in the interface A, above.

Oompa's type system is based upon [PS93a], which considers a type system for the π-calculus, and [AC91], which gives a type system with subtyping and recursive types for the lambda calculus. The work we present in this chapter has been published as a technical report [TB02].

In Section 5.1, we formally define type violations and give the rules of the type safety system. Section 5.2 gives the semantics of types, which are a kind of infinite tree. We define Oompa's subtyping system in Section 5.3. In Section 5.4, we prove that a type safe program never commits a type violation. We compare our type system with some others in Section 5.5 and Section 5.6 summarises the chapter.

5.1 Typing, Type Violations and Type Safety

In this section, we give the typing system, define the notion of type violation and give the type safety system.

Table 5.1: Rules of the Typing System

TAS-ASS:	$\Gamma, \Phi \cup \{v{:}\,T\} \vdash v{:}\,T$
TAS-LIT:	$\Gamma, \Phi \vdash \ell_P{:}\,P$
where ℓ_P is a literal of primitive type P.	
TAS-SUB:	$\dfrac{\Gamma \vdash T_1 \leq T_2 \qquad \Gamma, \Phi \vdash v{:}\,T_1}{\Gamma, \Phi \vdash v{:}\,T_2}$

5.1.1 The Typing System

An assertion that a value v is of type T is called a *typing* and is written $v{:}\,T$. Syntactically, values in Oompa are either literal expressions or names. The types of literal expressions are unambiguous since their syntactic classes are required to be disjoint, e.g. 5 is always a `Long`. The types of names, however, will depend on a context consisting of a type dictionary and a definition set. It is the role of a type dictionary to record a list of assumptions about the types of names, although a definition set may be required so that the interface types associated with class or interface names may be looked up.

The *typing system* is a proof system that establishes typing judgements of the form $\Gamma, \Phi \vdash v{:}\,T$ where Γ is a definition set, Φ is a type dictionary, v is a value and T is a type. The statement can be read as "v is a T in the context Γ,Φ". It is defined by the three rules in Figure 5.1.

The first two rules are TAS-ASS and TAS-LIT, which are the axioms from which other statements are derived. Oompa's type system supports subtyping, so deciding whether a value has a certain type may require the use of the subtyping system. This is the role of the third rule, TAS-SUB, which incorporates a subtyping statement of the form $\Gamma \vdash T_1 \leq T_2$ (see Section 5.3).

5.1.2 Type Violations

We now formalise the notion of *type violation* by listing the uses of a value which are not compatible with that value's type. Our choice of these determines the properties we expect of the type system.

In an agent system, $\Gamma \triangleright g\{^{\Phi}_{\Delta}$, an agent $o[p] \in g$ is *type violating* if any of the following conditions hold:

- *Feature Mismatch.* The agent attempts to use a feature which doesn't exist by:
  - invoking a method, m, of an object, o'. $o'\colon C \in \Phi$ and p begins $o'.m!\langle v_1, \ldots v_n\rangle?(r_1, \ldots r_m)$ but either C is not a class or C's definition in Γ has no method m.
  - accessing an attribute, a, that o doesn't possess. So $o\colon C \in \Phi$ and p begins $a?r$ but C's definition in Γ has no attribute a.
  - updating an attribute, a, that o doesn't possess. So $o\colon C \in \Phi$ and p begins $a!v$ but C's definition in Γ has no attribute a.
- *Arity Mismatch.* The agent attempts to use a tuple of the wrong length by:
  - receiving on a channel, c. So $c\colon \mathtt{Chan}\langle T_1, \ldots T_n\rangle \in \Phi$ but p begins $c?(r_1\colon T'_1, \ldots r_m\colon T'_m)$ where $n \neq m$.
  - sending on a channel, c. So $c\colon \mathtt{Chan}\langle T_1, \ldots T_n\rangle \in \Phi$ but p begins $c!\langle v_1, \ldots v_m\rangle$ where $n \neq m$.
  - invoking a method, m, of an object, o'. So $o'\colon C \in \Phi$ and C's definition in Γ gives m the signature type $m?(T_1, \ldots T_n)!\langle T'_1, \ldots T'_m\rangle$. However, p begins $o'.m!\langle v_1, \ldots v_j\rangle?(r_1, \ldots r_k)$ where either $j \neq n$ or $k \neq m$.

- *Value Mismatch.* The agent attempts to use a value which is of the wrong type by:
  - sending on a channel, c. So $c{:}\,\mathtt{Chan}\langle T_1, \dots T_n\rangle \in \Phi$ and p begins $c!\langle v_1, \dots v_n\rangle$ but, for some i, we have $\Gamma, \Phi \nvdash v_i{:}\,T_i$.
  - invoking a method, m, of an object, o'. So $o'{:}\,C \in \Phi$, C's definition in Γ gives m the signature type $m?(T_1, \dots T_n)!\langle T'_1, \dots T'_m\rangle$ and p begins $o'.m!\langle v_1, \dots v_n\rangle?(r_1, \dots r_m)$. However, for some i, we have $\Gamma, \Phi \nvdash v_i{:}\,T_i$.
  - updating an attribute, a. So $o{:}\,C \in \Phi$, C's definition in Γ gives a the type $\mathtt{Attr}\{T\}$ and p begins $a!v$. However, $\Gamma, \Phi \nvdash v{:}\,T$.

An agent system, $\Gamma \rhd g\{^{\Phi}_{\Delta}$, is *type violating* if any agent in g is type violating. It is useful to note that type violations are only associated with channel, method or attribute use.

Another situation to beware of occurs when an agent attempts to create an object of a type that is not a class. This is certainly an error but we choose not to call it a type violation since it does not actually represent the *misuse* of a value. We enable our type system to detect these errors at the cause, even though it would detect any invocation on such an object as a type violation.

5.1.3 Type Safety

The *Type Safety System* is a proof system that establishes type safety judgements of the form $\Gamma, \Phi \vdash x$ where Γ is a definition set, Φ is a type dictionary and x is some Oompa expression[1]. We are saying that x is well-behaved in

[1]Notice that the form of these judgements is different from that of the typing system. In fact, an alternative approach to type safety actually gives an *okay type* to all expres-

the context of Γ and Φ. It is defined by the twelve rules in tables 5.2 and 5.3.

The nine rules in Table 5.2 establish the type safety of a piece of code. Each of the rules is concerned with a single code primitive. The TSS-END, TSS-FRK, TSS-NEW, TSS-SND, TSS-RCV and TSS-CRT rules deal with ends, forking, channel creation, sending, receiving and object creation respectively. The TSS-INV rule only checks the outgoing part of an invocation for safety. It converts the invocation into a receive on a new channel name and allows the TSS-RCV rule to check the safety of its receipt of values. This matches the operational semantics and facilitates convenient proofs later on. The TSS-ACC and TSS-UPD rules check attribute access and update, respectively.

The three rules in Table 5.3 establish the type safety of the definitions in a definition set. The TSS-MTH rule checks method definitions and considers them safe if their code will be safe in the context of an invocation, e.g. when there are appropriate input values and a return channel. The TSS-CLS rule checks class definitions and makes sure that their definitions will be type safe in the context of that class, i.e. with the attributes and special value `this` of the appropriate type.

To rule for checking a definition set requires that all class definitions in the system are type safe with respect to each other. A definition set can contain references to global channel names and the types of these names are not contained within the definition set. Therefore, the type safety of a definition set depends on a type dictionary Φ. This is why we write the particular type safety judgement that asserts the type safety of the complete definition set as $\Phi \vdash \Gamma$.

---

sions which are well-behaved in this way, e.g. $\Gamma, \Phi \vdash x\colon ok$ in [PS93a]. The difference is essentially notational and we prefer the more concise form as used, for example, in [San96].

Table 5.2: Rules for the Type Safety of Code

TSS-END: $\Gamma, \Phi \vdash \texttt{end}$

TSS-FRK: $$\frac{\Gamma, \Phi \vdash p_0 \qquad \Gamma, \Phi \vdash p_1}{\Gamma, \Phi \vdash \texttt{fork}\{p_0\}\ p_1}$$

TSS-NEW: $$\frac{\Gamma, \Phi \cup \{c\colon \mathrm{ChT}\} \vdash p}{\Gamma, \Phi \vdash \texttt{new}\ c\colon \mathrm{ChT}\ p}$$

TSS-SND:
$$\frac{\Gamma, \Phi \vdash c\colon \texttt{OuCh}\langle T_1, \ldots T_n\rangle \quad \Gamma, \Phi \vdash v_1\colon T_1 \cdots \Gamma, \Phi \vdash v_n\colon T_n \quad \Gamma, \Phi \vdash p}{\Gamma, \Phi \vdash c!\langle v_1, \ldots v_n\rangle\ p}$$

TSS-RCV: $$\frac{\Gamma, \Phi \vdash c\colon \texttt{InCh}\langle T_1, \ldots T_n\rangle \qquad \Gamma, \Phi \cup \{r_1\colon T_1, \ldots r_n\colon T_n\} \vdash p}{\Gamma, \Phi \vdash c?(r_1\colon T_1, \ldots r_n\colon T_n)\ p}$$

TSS-CRT: $$\frac{\Gamma, \Phi \cup \{o\colon C\} \vdash p}{\Gamma, \Phi \vdash \texttt{create}\ o\colon C\ p}$$

where C is defined as a class in Γ.

TSS-INV: $$\frac{\left(\begin{array}{c} \Gamma, \Phi \vdash o\colon \texttt{Intf}\{m?(T_1, \ldots T_n)!\langle T'_1 \ldots T'_{n'}\rangle\} \\ \Gamma, \Phi \vdash v_1\colon T_1 \qquad \cdots \qquad \Gamma, \Phi \vdash v_n\colon T_n \\ \Gamma, \Phi \cup \{x\colon \texttt{Chan}\langle T'_1, \ldots T'_{n'}\rangle\} \vdash x?(r_1\colon T'_1, \ldots r_{n'}\colon T'_{n'})p \end{array}\right)}{\Gamma, \Phi \vdash o.m!\langle v_1, \ldots v_n\rangle?(r_1, \ldots r_{n'})\ p}$$

where x is new.

TSS-ACC: $$\frac{\Gamma, \Phi \vdash a\colon \texttt{Attr}\{T\} \qquad \Gamma, \Phi \cup \{r\colon T\} \vdash p}{\Gamma, \Phi \vdash a?r\ p}$$

TSS-UPD: $$\frac{\Gamma, \Phi \vdash a\colon \texttt{Attr}\{T\} \qquad \Gamma, \Phi \vdash v\colon T \qquad \Gamma, \Phi \vdash p}{\Gamma, \Phi \vdash a!v\ p}$$

Table 5.3: Rules for the Type Safety of Definitions

TSS-MTH:
$$\frac{\Gamma, \Phi \cup \{r_1\colon T_1, \ldots r_n\colon T_n, \texttt{return}\colon \texttt{OuCh}\langle T'_1, \ldots T'_{n'}\rangle\} \vdash p}{\Gamma, \Phi \vdash m?(r_1\colon T_1, \ldots r_n\colon T_n)!\langle d_1\colon T'_1, \ldots d_m\colon T'_{n'}\rangle\{p\}}$$

TSS-CLS:
$$\frac{\begin{pmatrix} \Gamma, \Phi \cup \{a_1\colon \mathrm{AtT}_1, \ldots a_n\colon \mathrm{AtT}_n, \texttt{this}\colon C\} \vdash \mathrm{mdef}_1 \\ \vdots \\ \Gamma, \Phi \cup \{a_1\colon \mathrm{AtT}_1, \ldots a_n\colon \mathrm{AtT}_n, \texttt{this}\colon C\} \vdash \mathrm{mdef}_m \end{pmatrix}}{\Gamma, \Phi \vdash \texttt{class}\ C\ \{a_1\colon \mathrm{AtT}_1\ \ldots\ a_n\colon \mathrm{AtT}_n\ \mathrm{mdef}_1\ \ldots\ \mathrm{mdef}_m\}}$$

TSS-DEF:
$$\frac{\Gamma, \Phi \vdash \mathrm{Cdef}_1 \qquad \cdots \qquad \Gamma, \Phi \vdash \mathrm{Cdef}_n}{\Phi \vdash \Gamma}$$

where $\mathrm{Cdef}_1 \ldots \mathrm{Cdef}_n$ are all the class definitions in Γ.

5.2 Oompa Type Trees

The traditional semantics for a type is as a set of values, so $v\colon T$ can be read as $v \in T$. However, in the presence of recursive types, well-founded sets do not form a model of a type system (not in the standard way, at least). Following [AC91], we will instead use infinite trees as a semantics for Oompa's types.

5.2.1 Trees and Subtrees

The nodes on an Oompa type tree are labelled with symbols from the following *ranked alphabet*. The superscript of the symbols denotes the number of children the node it labels should have and the symbols corresponding to signature types carry a method name.

$$\begin{aligned} L \;=\; & \{\texttt{Long}^0, \texttt{Char}^0\} \\ & \cup \{\mathrm{Sig}^2(m) \mid m \in \mathcal{N}\} \\ & \cup \{\mathrm{Intf}^n, \mathrm{Chan}^n, \mathrm{InCh}^n, \mathrm{OuCh}^n, \mathrm{InParam}^n, \mathrm{InParam}^n \mid n \in \mathbb{N}\} \end{aligned}$$

We will use l^n to stand for a general member of L. We take the view that the set of natural numbers, $\mathbb{N}$, includes 0 and we will use the symbol $\mathbb{N}^+$ for the set of strictly positive natural numbers.

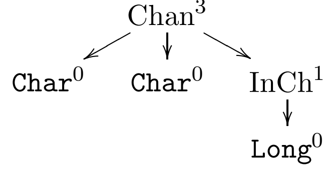
We can descend a tree from its root by specifying, at each node, the child we next want to visit as "the ith child from the left". Therefore, we can view a sequence of positive natural numbers as a path through a tree and represent trees as partial functions from paths to the label on the node reached by that path. We will use $\downarrow$ to mean "is defined", $\pi\sigma$ for the concatenation of π and σ and Λ for the empty sequence.

We define an *Oompa type tree*, A, to be a partial function from $\mathbb{N}^{+*}$ to L, which satisfies the following conditions:

- $A(\Lambda)\downarrow$
- $A(\pi\sigma)\downarrow \implies A(\pi)\downarrow$
- $A(\pi) = l^n \implies A(\pi j)\downarrow$ where $j \leq n$
- $A(\pi) = \mathrm{Intf}^n \implies \begin{array}{l} A(\pi j) = \mathrm{Sig}^2(m) \text{ where } j \leq n \text{ and} \\ \text{if } A(\pi i) = \mathrm{Sig}^2(m') \text{ for } i \neq j \text{ then } m \neq m' \end{array}$
- $A(\pi j) = \mathrm{Sig}^2(m) \implies A(\pi) = \mathrm{Intf}^n,$ some $n \in \mathbb{N}$ such that $n \geq j$
- $A(\pi) = \mathrm{Sig}^2(m) \implies \begin{array}{l} A(\pi 1) = \mathrm{InParam}^n, \text{ some } n \in \mathbb{N} \text{ and} \\ A(\pi 2) = \mathrm{OutParam}^{n'}, \text{ some } n' \in \mathbb{N} \end{array}$
- $A(\pi j) = \mathrm{InParam}^n \implies A(\pi) = \mathrm{Sig}^2(m),$ some m
- $A(\pi j) = \mathrm{OutParam}^n \implies A(\pi) = \mathrm{Sig}^2(m),$ some m

Although formally useful, it is not very intuitive to think of trees as the partial functions defined above. A simple example of an Oompa type tree

represented in a traditional style is the following.



Say that A is an Oompa type tree and π is a sequence from $\mathbb{N}^+$. Then the *subtree* of A at π, written $A[\pi]$, is the partial function such that

$$A[\pi](\sigma) = A(\pi\sigma) \qquad \forall \sigma \in {\mathbb{N}^+}^*$$

Lemma 5.2.1 *If A is an Oompa type tree and A is defined at π, then $A[\pi]$ is also an Oompa type tree.*

Proof: Easy to show by contradiction. ■

5.2.2 Interpreting Types as Trees

We are going to use Oompa type trees as a semantics for Oompa's types, so we need to define how to interpret a type as a tree. Since type trees never involve type variables, when interpreting a type as a tree, we will recursively look up the corresponding definition for each type variable and expand it into the tree.

Say sig is a method signature. Then let sig^- be sig with all its parameter names removed. This simple syntactic step gives the signature type of the signature. For example,

$$m?(r_1\colon T_1, \ldots r_n\colon T_n)!\langle s_1\colon S_1, \ldots s_{n'}\colon S_{n'}\rangle^- = m?(T_1, \ldots T_n)!\langle S_1, \ldots S_{n'}\rangle$$

We define the *look-up function* $\mathrm{e}_\Gamma(\cdot)$ to perform this operation of looking-up and expanding definitions as:

$$\mathrm{e}_\Gamma(t) = \texttt{Intf}\{\text{sig}_1^-, \ldots \text{sig}_n^-\}$$

if Γ contains either an interface of the form

$$\texttt{interface}\ t\ \{\mathrm{sig}_1 \ldots \mathrm{sig}_n\}$$

or a class of the form

$$\texttt{class}\ t\ \{\mathrm{adecl}_1 \ldots \mathrm{adecl}_{n'}\ \ \mathrm{sig}_1\{p_1\} \ldots \mathrm{sig}_n\{p_n\}\}$$

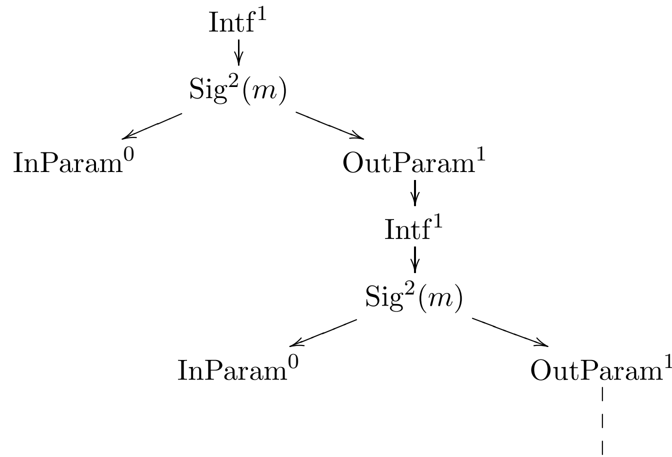
We now define the *interpretation function*, Tree_Γ, which takes an Oompa type and a definition set Γ and gives its corresponding Oompa type tree. The free type variables in the type are completely expanded using the $\mathrm{e}_\Gamma(\cdot)$ function repeatedly.

$$
\begin{array}{lcl}
\mathrm{Tree}_\Gamma(\texttt{Long})(\Lambda) & = & \texttt{Long}^0 \\
\mathrm{Tree}_\Gamma(\texttt{Char})(\Lambda) & = & \texttt{Char}^0 \\
\mathrm{Tree}_\Gamma(\texttt{Chan}\langle T_1, \ldots T_n\rangle)(\Lambda) & = & \mathrm{Chan}^n \\
\mathrm{Tree}_\Gamma(\texttt{Chan}\langle T_1, \ldots T_n\rangle)(i\pi) & = & \mathrm{Tree}_\Gamma(T_i)(\pi) \\
\mathrm{Tree}_\Gamma(\texttt{InCh}\langle T_1, \ldots T_n\rangle)(\Lambda) & = & \mathrm{InCh}^n \\
\mathrm{Tree}_\Gamma(\texttt{InCh}\langle T_1, \ldots T_n\rangle)(i\pi) & = & \mathrm{Tree}_\Gamma(T_i)(\pi) \\
\mathrm{Tree}_\Gamma(\texttt{OuCh}\langle T_1, \ldots T_n\rangle)(\Lambda) & = & \mathrm{OuCh}^n \\
\mathrm{Tree}_\Gamma(\texttt{OuCh}\langle T_1, \ldots T_n\rangle)(i\pi) & = & \mathrm{Tree}_\Gamma(T_i)(\pi) \\
\mathrm{Tree}_\Gamma(\texttt{Intf}\{\mathrm{SgT}_1, \ldots \mathrm{SgT}_n\})(\Lambda) & = & \mathrm{Intf}^n \\
\mathrm{Tree}_\Gamma(\texttt{Intf}\{\mathrm{SgT}_1, \ldots \mathrm{SgT}_n\})(i\pi) & = & \mathrm{Tree}_\Gamma(\mathrm{SgT}_i)(\pi) \\
\mathrm{Tree}_\Gamma(m?(T_1, \ldots T_n)!\langle S_1, \ldots S_{n'}\rangle)(\Lambda) & = & \mathrm{Sig}^2(m) \\
\mathrm{Tree}_\Gamma(m?(T_1, \ldots T_n)!\langle S_1, \ldots S_{n'}\rangle)(1) & = & \mathrm{InParam}^n \\
\mathrm{Tree}_\Gamma(m?(T_1, \ldots T_n)!\langle S_1, \ldots S_{n'}\rangle)(2) & = & \mathrm{OutParam}^n \\
\mathrm{Tree}_\Gamma(m?(T_1, \ldots T_n)!\langle S_1, \ldots S_{n'}\rangle)(1i\pi) & = & \mathrm{Tree}_\Gamma(T_i)(\pi) \\
\mathrm{Tree}_\Gamma(m?(T_1, \ldots T_n)!\langle S_1, \ldots S_{n'}\rangle)(2i\pi) & = & \mathrm{Tree}_\Gamma(S_i)(\pi) \\
\mathrm{Tree}_\Gamma(\texttt{rec}\ t.T)(\pi) & = & \mathrm{Tree}_\Gamma(T\{\!|^{\texttt{rec}\ t.T}/t|\!\})(\pi) \\
\mathrm{Tree}_\Gamma(t)(\pi) & = & \mathrm{Tree}_\Gamma(\mathrm{e}_\Gamma(t))(\pi)
\end{array}
$$

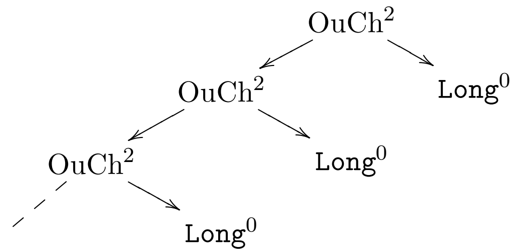
The function is undefined in all other cases. Substitution on types is as expected.

We use this function to define a notion of type equivalence. Two types are *equivalent* if their trees are equal. So we write $\Gamma \models A = B$ if and only if $\mathrm{Tree}_\Gamma(A) = \mathrm{Tree}_\Gamma(B)$.

We give two pictures of the trees of complex types. The type of the interface A, given on page 124, is implicitly recursive, so its tree will be infinite. It will look like this, where the dotted line indicates that the structure repeats following the obvious pattern.



The tree of the explicitly recursive type `rec` $t.$`OuCh`$\langle t,$ `Long`$\rangle$ will also be infinite. It will look like this:



5.2.3 Expanding Definitions

We now introduce the *expansion function*, $\mathrm{E}_\Gamma^\emptyset(\cdot)$, which completely expands an Oompa type so that it no longer depends on the definition set. It does this by converting *implicit recursion*, which is due to recursive look-ups, into *explicit recursion*, which uses the `rec` operator. $\mathrm{E}_\Gamma^\emptyset(\cdot)$ is just a specific case of the recursive function $\mathrm{E}_\Gamma^V(\cdot)$ which looks-up all the variables it encounters other than those in V.

$$
\begin{array}{lcl}
\mathrm{E}_\Gamma^V(\texttt{Long}) & = & \texttt{Long}^0 \\
\mathrm{E}_\Gamma^V(\texttt{Char}) & = & \texttt{Char}^0 \\
\mathrm{E}_\Gamma^V(\texttt{Chan}\langle T_1, \ldots T_n\rangle) & = & \texttt{Chan}\langle \mathrm{E}_\Gamma^V(T_1), \ldots \mathrm{E}_\Gamma^V(T_n)\rangle \\
\mathrm{E}_\Gamma^V(\texttt{InCh}\langle T_1, \ldots T_n\rangle) & = & \texttt{InCh}\langle \mathrm{E}_\Gamma^V(T_1), \ldots \mathrm{E}_\Gamma^V(T_n)\rangle \\
\mathrm{E}_\Gamma^V(\texttt{OuCh}\langle T_1, \ldots T_n\rangle) & = & \texttt{OuCh}\langle \mathrm{E}_\Gamma^V(T_1), \ldots \mathrm{E}_\Gamma^V(T_n)\rangle \\
\mathrm{E}_\Gamma^V(\texttt{Intf}\{\mathrm{SgT}_1, \ldots \mathrm{SgT}_n\}) & = & \texttt{Intf}\{\mathrm{E}_\Gamma^V(\mathrm{SgT}_1), \ldots \mathrm{E}_\Gamma^V(\mathrm{SgT}_n)\} \\
\mathrm{E}_\Gamma^V(m?(T_1, \ldots T_n)!\langle T'_1, \ldots T'_{n'}\rangle) & = & m?(\mathrm{E}_\Gamma^V(T_1), \ldots \mathrm{E}_\Gamma^V(T_n)) \\
 & & \quad !\langle \mathrm{E}_\Gamma^V(T'_1), \ldots \mathrm{E}_\Gamma^V(T'_{n'})\rangle \\
\mathrm{E}_\Gamma^V(t) & = & \begin{cases} t & \text{if } t \in V \\ \texttt{rec}\ t.\mathrm{E}_\Gamma^{V\cup\{t\}}(\mathrm{e}_\Gamma(t)) & \text{otherwise} \end{cases} \\
\mathrm{E}_\Gamma^V(\texttt{rec}\ t.T) & = & \texttt{rec}\ t.\mathrm{E}_\Gamma^{V\cup\{t\}}(T)
\end{array}
$$

In the last of these cases, we must be careful to avoid inappropriate variable capture. We use α-substitutability to rename t different to any other name in the system before applying this step of the expansion function.

We can view the definition set as a set of simultaneous equations and the expansion function as an algorithm for solving them for a particular type variable. As an example, say that Γ contains the two interfaces given on

page 124. If we apply the expansion function to the interface name A we get:

$$\begin{aligned}
\mathrm{E}^{\emptyset}_{\Gamma}(A) &= \texttt{rec}\ A.\mathrm{E}^{\{A\}}_{\Gamma}(\texttt{Intf}\{m?()!\langle B\rangle\}) \\
&= \texttt{rec}\ A.\texttt{Intf}\{\mathrm{E}^{\{A\}}_{\Gamma}(m?()!\langle B\rangle)\} \\
&= \texttt{rec}\ A.\texttt{Intf}\{m?()!\langle \mathrm{E}^{\{A\}}_{\Gamma}(B)\rangle\} \\
&= \texttt{rec}\ A.\texttt{Intf}\{m?()!\langle \texttt{rec}\ B.\mathrm{E}^{\{A,B\}}_{\Gamma}(\texttt{Intf}\{m?()!\langle A\rangle\})\rangle\} \\
&= \texttt{rec}\ A.\texttt{Intf}\{m?()!\langle \texttt{rec}\ B.\texttt{Intf}\{\mathrm{E}^{\{A,B\}}_{\Gamma}(m?()!\langle A\rangle)\}\rangle\} \\
&= \texttt{rec}\ A.\texttt{Intf}\{m?()!\langle \texttt{rec}\ B.\texttt{Intf}\{m?()!\langle \mathrm{E}^{\{A,B\}}_{\Gamma}(A)\rangle\}\rangle\} \\
&= \texttt{rec}\ A.\texttt{Intf}\{m?()!\langle \texttt{rec}\ B.\texttt{Intf}\{m?()!\langle A\rangle\}\rangle\}
\end{aligned}$$

Lemma 5.2.2 $\mathrm{E}^{V}_{\Gamma}(T)$ *is a finite type expression and can be calculated with finite applications of the look-up function* $\mathrm{e}_{\Gamma}(\cdot)$ *for any finite* Γ*,* T *and* V*.*

C.1.1 p315

The next result validates the expansion function, by showing that the type it generates is equivalent to the first, in the context Γ.

C.1.3 p318

Theorem 5.2.3 $\Gamma \models S = \mathrm{E}^{\emptyset}_{\Gamma}(S)$

5.3 Subtyping

In a system without subtyping, a value of type T can only be used in a context which requires values of type S when $T = S$. Subtyping defines a relation between types, $T \leq S$, and allows values of type T to be used in contexts intended for values of type S when $T \leq S$. Subtyping increases the flexibility of a type system while preserving type safety

Our type system will support two general subtype relationships. Interface subtyping allows an object to be used in a context so long as it supports all those methods required by the context's type — it may also support other methods. Channel subtyping allows an input/output channel name to be used where an input channel name or output channel name is required.

However, to properly support subtyping, we need to take account of the fact that values of channel and interface types are also contexts for other values. A channel is a context for those values which can be sent along it and a method of an object can receive and send values.

For example, let's say that T is a subtype of S and that c is a channel of type $\texttt{Chan}\langle T\rangle$. Then c should also be useable as a channel of type $\texttt{InCh}\langle S\rangle$ since, if the values we receive values off c are of type T, the values we receive off c are useable as values of type S. Where there are recursive types, the situation is even more complex. For example, values of type $\texttt{rec}\ t.\texttt{Chan}\langle\texttt{InCh}\langle t\rangle\rangle$ can be used as values of type $\texttt{rec}\ s.\texttt{InCh}\langle s\rangle$.

The complexity of these relationships is such that it is appropriate to consider the subtyping relationship in the semantics[2]. We introduce a relation between Oompa type trees that captures the notion of subtyping. As the trees are possibly infinite structures, the relation is a simulation which compares their structure in terms of labelling and subtrees.

In theory, to test whether two types are subtypes, we could construct their trees and find a simulation between them. This would not be practical so we provide an algorithm, based on one in [PS93a], which operates on types and confirms whether their trees lie in an appropriate tree simulation. Our work differs from the reference in that our types can contain external references, i.e. names of classes and interfaces in the definition set. We manage this by closing the types using our expansion function $\mathrm{E}^{\emptyset}_{\Gamma}(\cdot)$. We prove that the algorithm terminates and is sound and complete with respect to the tree simulation.

[2] In fact, our main motivation for providing a semantics for the type system was to enable us to give meaning to the subtyping relation.

5.3.1 Tree Simulation

A relation $\mathcal{R}$ between Oompa type trees is an *Oompa tree simulation* if $(A, B) \in \mathcal{R}$ implies:

- If $B(\Lambda) = \texttt{Long}^0$ then $A(\Lambda) = \texttt{Long}^0$.
- If $B(\Lambda) = \texttt{Char}^0$ then $A(\Lambda) = \texttt{Char}^0$.
- If $B(\Lambda) = \text{Chan}^n$ then $A(\Lambda) = \text{Chan}^n$ and for each $1 \le i \le n$ both $(A[i], B[i]) \in \mathcal{R}$ and $(B[i], A[i]) \in \mathcal{R}$.
- If $B(\Lambda) = \text{InCh}^n$ then $A(\Lambda) = \text{InCh}^n$ or Chan^n and for each $1 \le i \le n$, $(A[i], B[i]) \in \mathcal{R}$.
- If $B(\Lambda) = \text{OuCh}^n$ then $A(\Lambda) = \text{OuCh}^n$ or Chan^n and for each $1 \le i \le n$, $(B[i], A[i]) \in \mathcal{R}$.
- If $B(\Lambda) = \text{Intf}^n$ then $A(\Lambda) = \text{Intf}^m$ where $m \ge n$ and there exists an injective function $f : [1..n] \to [1..m]$ such that for each $1 \le i \le n$, $(A[f(i)], B[i]) \in \mathcal{R}$.
- If $B(\Lambda) = \text{Sig}^2(m)$ then $A(\Lambda) = \text{Sig}^2(m)$ and for each $1 \le i \le 2$, $(A[i], B[i]) \in \mathcal{R}$
- If $B(\Lambda) = \text{InParam}^n$ then $A(\Lambda) = \text{InParam}^n$ and for each $1 \le i \le n$, $(B[i], A[i]) \in \mathcal{R}$.
- If $B(\Lambda) = \text{InParam}^n$ then $A(\Lambda) = \text{InParam}^n$ and for each $1 \le i \le n$, $(A[i], B[i]) \in \mathcal{R}$.

We use the notation $A \le B$ to indicate that $(A, B) \in \mathcal{R}$ for some tree simulation $\mathcal{R}$. For two types S and T, we use the notation $\Gamma \models S \le T$ if and only if $\text{Tree}_\Gamma(S) \le \text{Tree}_\Gamma(T)$.

As an example, we now show that

$$\models \texttt{rec } t.\texttt{Chan}\langle\texttt{InCh}\langle t\rangle\rangle \leq \texttt{rec } s.\texttt{InCh}\langle s\rangle$$

Let $A = \text{Tree}(\texttt{rec } t.\texttt{Chan}\langle\texttt{InCh}\langle t\rangle\rangle)$ and let $\perp$ stand for undefinedness. It is not difficult to see that

$$A(\pi) = \begin{cases} \perp & \text{if } \pi \notin \{1\}^* \\ \text{Chan}^1 & \text{if } |\pi| \text{ even} \\ \text{InCh}^1 & \text{if } |\pi| \text{ odd} \end{cases}$$

Let

$$A'(\pi) = \begin{cases} \perp & \text{if } \pi \notin \{1\}^* \\ \text{InCh}^1 & \text{if } |\pi| \text{ even} \\ \text{Chan}^1 & \text{if } |\pi| \text{ odd} \end{cases}$$

Then $A[1] = A'$ and $A'[1] = A$. Let $B = \text{Tree}(\texttt{rec } s.\texttt{InCh}\langle s\rangle)$. Then, similarly,

$$B(\pi) = \begin{cases} \perp & \text{if } \pi \notin \{1\}^* \\ \text{InCh}^1 & \text{otherwise} \end{cases}$$

and $B[1] = B$. Then $\{(A, B), (A', B)\}$ is a tree simulation which contains the pair $(\text{Tree}(\texttt{rec } t.\texttt{Chan}\langle\texttt{InCh}\langle t\rangle\rangle), \text{Tree}(\texttt{rec } s.\texttt{InCh}\langle s\rangle))$.

5.3.2 The Subtyping Algorithm

We now define the algorithm which determines whether the subtype relationship holds between two *closed types*, i.e. types which contain no free type variables.

The *subtyping algorithm* is a proof system whose statements have the form $\Sigma \vdash_a S \leq T$, where S and T are closed types and Σ is a set of assumptions, which have the form $U_1 \leq U_2$. It is defined by the eleven rules in Table 5.4. The proof system is considered an algorithm by putting an order on the

rules — when more than one rule are applicable, choose the first as they are presented. We will write $\Sigma \vDash_a S \gtreqless T$ if $\Sigma \vDash_a S \leq T$ and $\Sigma \vDash_a T \leq S$.

There are two axioms: SUB-RFL and SUB-ASS. Next we have the rules for channel subtyping. These are SUB-CHCH, SUB-ININ, SUB-CHIN, SUB-OUOU, SUB-CHOU. The SUB-SIG rule supports signature subtyping and is used by the SUB-INT rule to compare interface types. There are two rules for explicitly recursive types, SUB-RECLFT and SUB-RECRHT.

We extend the subtyping algorithm to all Oompa types by using the expansion function to close types. We write $\Gamma \vdash S \leq T$ if and only if $\emptyset \vDash_a \mathrm{E}^{\emptyset}_{\Gamma}(S) \leq \mathrm{E}^{\emptyset}_{\Gamma}(T)$ is provable from the subtyping algorithm.

The subtyping algorithm is called *terminating* if any application of the algorithm to a pair of types generates a finite proof tree. It is called *complete* if $\Gamma \models S \leq T$ implies $\Gamma \vdash S \leq T$ and called *sound* if $\Gamma \vdash S \leq T$ implies $\Gamma \models S \leq T$. The following results mean we can apply the algorithm to two Oompa types to establish that the simulation relationship holds between the Oompa type trees they represent.

Theorem 5.3.1 ***(Termination)*** *The subtyping algorithm is terminating.* C.2.2 p323

Theorem 5.3.2 ***(Completeness)*** *If* $\Gamma \models S \leq T$ *then* $\Gamma \vdash S \leq T$. C.2.5 p326

Theorem 5.3.3 ***(Soundness)*** *If* $\Gamma \vdash S \leq T$ *then* $\Gamma \models S \leq T$. C.2.10 p330

As an example, we give the proof tree of the following:

$$\vdash \texttt{rec}\ t.\texttt{Chan}\langle\texttt{InCh}\langle t\rangle\rangle \leq \texttt{rec}\ s.\texttt{InCh}\langle s\rangle$$

Table 5.4: Rules for the Subtyping Algorithm

SUB-RFL: $\Sigma \models_a T \leq T$

SUB-ASS: $\Sigma \cup \{S \leq T\} \models_a S \leq T$

SUB-CHCH: $$\frac{\Sigma \models_a S_1 \gtrless T_1 \quad \cdots \quad \Sigma \models_a S_n \gtrless T_n}{\Sigma \models_a \texttt{Chan}\langle S_1, \ldots S_n\rangle \leq \texttt{Chan}\langle T_1, \ldots T_n\rangle}$$

SUB-ININ: $$\frac{\Sigma \models_a S_1 \leq T_1 \quad \cdots \quad \Sigma \models_a S_n \leq T_n}{\Sigma \models_a \texttt{InCh}\langle S_1, \ldots S_n\rangle \leq \texttt{InCh}\langle T_1, \ldots T_n\rangle}$$

SUB-CHIN: $$\frac{\Sigma \models_a S_1 \leq T_1 \quad \cdots \quad \Sigma \models_a S_n \leq T_n}{\Sigma \models_a \texttt{Chan}\langle S_1, \ldots S_n\rangle \leq \texttt{InCh}\langle T_1, \ldots T_n\rangle}$$

SUB-OUOU: $$\frac{\Sigma \models_a T_1 \leq S_1 \quad \cdots \quad \Sigma \models_a T_n \leq S_n}{\Sigma \models_a \texttt{OuCh}\langle S_1, \ldots S_n\rangle \leq \texttt{OuCh}\langle T_1, \ldots T_n\rangle}$$

SUB-CHOU: $$\frac{\Sigma \models_a T_1 \leq S_1 \quad \cdots \quad \Sigma \models_a T_n \leq S_n}{\Sigma \models_a \texttt{Chan}\langle S_1, \ldots S_n\rangle \leq \texttt{OuCh}\langle T_1, \ldots T_n\rangle}$$

SUB-SIG:
$$\frac{\begin{pmatrix} \Sigma \models_a T_1 \leq S_1 & \cdots & \Sigma \models_a T_n \leq S_n \\ \Sigma \models_a S'_1 \leq T'_1 & \cdots & \Sigma \models_a S'_{n'} \leq T'_{n'} \end{pmatrix}}{\Sigma \models_a m?(S_1, \ldots S_n)!\langle S'_1, \ldots S'_{n'}\rangle \leq m?(T_1, \ldots T_n)!\langle T'_1, \ldots T'_{n'}\rangle}$$

SUB-INT: $$\frac{\Sigma \models_a \mathrm{SgT}_{f(1)} \leq \mathrm{SgT}'_1 \quad \cdots \quad \Sigma \models_a \mathrm{SgT}_{f(n)} \leq \mathrm{SgT}'_n}{\Sigma \models_a \begin{array}{l} \texttt{Intf}\{\mathrm{SgT}_1, \ldots \mathrm{SgT}_n, \mathrm{SgT}_{n+1}, \ldots \mathrm{SgT}_{n+m}\} \\ \leq \ \texttt{Intf}\{\mathrm{SgT}'_1, \ldots \mathrm{SgT}'_n\} \end{array}}$$

if there is such an injective function, f, from $[1..n]$ to $[1..(n+m)]$.

SUB-RECLFT: $$\frac{\Sigma \cup \{\texttt{rec}\ s.S \leq T\} \models_a S\{\!|{}^{\texttt{rec}\ s.S}/_{s}|\!\} \leq T}{\Sigma \models_a \texttt{rec}\ s.S \leq T}$$

SUB-RECRHT: $$\frac{\Sigma \cup \{S \leq \texttt{rec}\ t.T\} \models_a S \leq T\{\!|{}^{\texttt{rec}\ t.T}/_{t}|\!\}}{\Sigma \models_a S \leq \texttt{rec}\ t.T}$$

For the sake of readability, we do not give the sets of assumptions in full.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Sigma_2 \vDash_a \texttt{rec}\ t.\texttt{Chan}\langle\texttt{InCh}\langle t\rangle\rangle \leq \texttt{rec}\ s.\texttt{InCh}\langle s\rangle}{\Sigma_2 \vDash_a \texttt{InCh}\langle\texttt{rec}\ t.\texttt{Chan}\langle\texttt{InCh}\langle t\rangle\rangle\rangle \leq \texttt{InCh}\langle\texttt{rec}\ s.\texttt{InCh}\langle s\rangle\rangle}}{\Sigma_1 \vDash_a \texttt{InCh}\langle\texttt{rec}\ t.\texttt{Chan}\langle\texttt{InCh}\langle t\rangle\rangle\rangle \leq \texttt{rec}\ s.\texttt{InCh}\langle s\rangle}}{\Sigma_1 \vDash_a \texttt{Chan}\langle\texttt{InCh}\langle\texttt{rec}\ t.\texttt{Chan}\langle\texttt{InCh}\langle t\rangle\rangle\rangle\rangle \leq \texttt{InCh}\langle\texttt{rec}\ s.\texttt{InCh}\langle s\rangle\rangle}}{\Sigma_0 \vDash_a \texttt{Chan}\langle\texttt{InCh}\langle\texttt{rec}\ t.\texttt{Chan}\langle\texttt{InCh}\langle t\rangle\rangle\rangle\rangle \leq \texttt{rec}\ s.\texttt{InCh}\langle s\rangle}}{\emptyset \vDash_a \texttt{rec}\ t.\texttt{Chan}\langle\texttt{InCh}\langle t\rangle\rangle \leq \texttt{rec}\ s.\texttt{InCh}\langle s\rangle}$$

5.4 Soundness of the Type Safety System

We need to show that if a definition set passes our type safety test, then the running Oompa systems which can evolve from its initial system will never commit a type violation. This is usually called the *soundness* of a type system. It is useful to have a concept of the state of the type system “at run time”. We augment the type safety system to give the dynamic type safety system, which allows us to check that a running Oompa system is obeying the type discipline.

We use this concept to prove the soundness of our type safety system in three stages. First, we show that dynamically type safe systems are not committing a type violation. Then, we show that a type safe definition set gives rise to a dynamically type safe initial system. Finally, we show that dynamic type safety is preserved by the operational semantics.

This technique is a version of *subject reduction* [WF91], which considers “each intermediate state of a program is itself a program [...]. Thus, proving type soundness reduces to proving that well-typed programs yield only well-typed results.” There are two details to consider when applying this approach to Oompa. First, our programs are written in static definitions, which are type checked, and our agents are not of the same form (this is why we will

need a separate *dynamic* type safety system). Second, it is not useful to consider an Oompa expression reducing to a result, so we merely require all the systems arrived at by the operational semantics to be dynamically type safe.

5.4.1 Dynamic Type Safety

Given an agent system $\Gamma \triangleright g\{^{\Phi}_{\Delta}$, we use the notation $\Gamma \models^{\triangleright}_{\mathrm{d}} g\{^{\Phi}_{\Delta}$ to mean that the system is type safe in a dynamic sense, i.e. all values in the system are being used appropriately. To speak about the type safety of agent code (as opposed to static method code) we need to modify the original type safety system slightly to take account of naming issues.

The dynamic type safety system for code is like that of the type safety system for code except four rules, TSS-NEW, TSS-CRT, TSS-RCV and TSS-ACC, have changed. It is given by the nine rules in Table 5.5.

The dynamic type safety of an agent system is defined in terms of its parts, as follows:

- We say $\Gamma \models^{\triangleright}_{\mathrm{d}} g\{^{\Phi}_{\Delta}$ if and only if $\Gamma, \Phi \models^{\triangleright}_{\mathrm{d}} g$ and $\Gamma, \Phi \models^{\triangleright}_{\mathrm{d}} \Delta$.
- We say $\Gamma, \Phi \models^{\triangleright}_{\mathrm{d}} g$ if and only if $g = \mathrm{nil}$ or $g = o_1[p_1] | \ldots o_n[p_n]$ where $\Gamma, \Phi \models^{\triangleright}_{\mathrm{d}} o_i[p_i]$ for all i.
- We say $\Gamma, \Phi \models^{\triangleright}_{\mathrm{d}} o[p]$ if and only if $\Phi(o) = C$ for some class C where

  $$\texttt{class}\ C\ \{a_1\!: \mathrm{AtT}_1\ \cdots\ a_n\!: \mathrm{AtT}_n\ \ \mathrm{mdef}_1 \ldots \mathrm{mdef}_{n'}\} \in \Gamma$$

  such that $\Gamma, \Phi \cup \{a_1\!: \mathrm{AtT}_1 \ldots a_n\!: \mathrm{AtT}_n\} \models^{\triangleright}_{\mathrm{d}} p$.
- We say $\Gamma, \Phi \models^{\triangleright}_{\mathrm{d}} \Delta$ if and only if for all $o \in \mathrm{Dom}(\Delta)$ we have $\Phi(o) = C$ for some class C where

  $$\texttt{class}\ C\ \{a_1\!: \texttt{Attr}\{T_1\}\ \cdots\ a_n\!: \texttt{Attr}\{T_n\}\ \ \mathrm{mdef}_1 \ldots \mathrm{mdef}_{n'}\} \in \Gamma$$

Table 5.5: Rules for the Dynamic Type Safety of Code

DTS-END: $$\Gamma, \Phi \models^{\rhd}_{\mathrm{d}} \texttt{end}$$

DTS-FRK: $$\frac{\Gamma, \Phi \models^{\rhd}_{\mathrm{d}} p_0 \qquad \Gamma, \Phi \models^{\rhd}_{\mathrm{d}} p_1}{\Gamma, \Phi \models^{\rhd}_{\mathrm{d}} \texttt{fork}\{p_0\}\ p_1}$$

DTS-NEW: $$\frac{\Gamma, \Phi \cup \{c'\colon \mathrm{ChT}\} \models^{\rhd}_{\mathrm{d}} p\{\!|{}^{c'}/c|\!\}}{\Gamma, \Phi \models^{\rhd}_{\mathrm{d}} \texttt{new}\ c\colon \mathrm{ChT}\colon\ p}$$

DTS-SND:
$$\frac{\Gamma, \Phi \vdash c\colon \texttt{OuCh}\langle T_1, \dots T_n\rangle \quad \Gamma, \Phi \vdash v_1\colon T_1 \cdots \Gamma, \Phi \vdash v_n\colon T_n \quad \Gamma, \Phi \models^{\rhd}_{\mathrm{d}} p}{\Gamma, \Phi \models^{\rhd}_{\mathrm{d}} c!\langle v_1, \dots v_n\rangle\ p}$$

DTS-RCV:
$$\frac{\Gamma, \Phi \vdash c\colon \texttt{InCh}\langle T_1, \dots T_n\rangle \quad \Gamma, \Phi \cup \{r'_1\colon T_1, \dots r'_n\colon T_n\} \models^{\rhd}_{\mathrm{d}} p\{\!|{}^{r'_1}/r_1 \dots {}^{r'_n}/r_n|\!\}}{\Gamma, \Phi \models^{\rhd}_{\mathrm{d}} c?(r_1\colon T_1, \dots r_n\colon T_n)\ p}$$

DTS-CRT: $$\frac{\Gamma, \Phi \cup \{o'\colon C\} \models^{\rhd}_{\mathrm{d}} p\{\!|{}^{o'}/o|\!\}}{\Gamma, \Phi \models^{\rhd}_{\mathrm{d}} \texttt{create}\ o\colon C\colon\ p}$$

where C is defined as a class in Γ.

DTS-INV:
$$\frac{\left(\begin{array}{c} \Gamma, \Phi \vdash o\colon \texttt{Intf}\{m?(T_1, \dots T_n)!\langle T'_1 \dots T'_{n'}\rangle\} \\ \Gamma, \Phi \vdash v_1\colon T_1 \qquad \cdots \qquad \Gamma, \Phi \vdash v_n\colon T_n \\ \Gamma, \Phi \cup \{x\colon \texttt{Chan}\langle T'_1, \dots T'_{n'}\rangle\} \models^{\rhd}_{\mathrm{d}} x?(r_1\colon T'_1, \dots r_{n'}\colon T'_{n'})p \end{array}\right)}{\Gamma, \Phi \models^{\rhd}_{\mathrm{d}} o.m!\langle v_1, \dots v_n\rangle?(r_1, \dots r_{n'})\ p}$$

where x is new.

DTS-ACC: $$\frac{\Gamma, \Phi \vdash a\colon \texttt{Attr}\{T\} \qquad \Gamma, \Phi \cup \{r'\colon T\} \models^{\rhd}_{\mathrm{d}} p\{\!|{}^{r'}/r|\!\}}{\Gamma, \Phi \models^{\rhd}_{\mathrm{d}} a?r\ p}$$

DTS-UPD: $$\frac{\Gamma, \Phi \vdash a\colon \texttt{Attr}\{T\} \qquad \Gamma, \Phi \vdash v\colon T \qquad \Gamma, \Phi \models^{\rhd}_{\mathrm{d}} p}{\Gamma, \Phi \models^{\rhd}_{\mathrm{d}} a!v\ p}$$

such that for all $a \in \mathrm{Dom}(\Delta(o))$ we have $a = a_i$ for some i and $\Gamma, \Phi \vdash o(a) \colon T_i$.

A key property of dynamic type safety is that an Oompa system which is dynamically type safe is not type violating.

C.3.1 p331

Lemma 5.4.1 *If* $\Gamma \mid\!\overset{\triangleright}{_\mathrm{d}}\, g\{^{\Phi}_{\Delta}$ *then* $\Gamma \triangleright g\{^{\Phi}_{\Delta}$ *is not type violating.*

Another important property is that the initial system of a statically type safe definition set is dynamically type safe. Recall that g_Γ is the initial agent derived from a definition set (see page 83).

C.3.3 p332

Lemma 5.4.2 *If* $\Phi_0 \vdash \Gamma$ *then* $\Gamma \mid\!\overset{\triangleright}{_\mathrm{d}}\, g_\Gamma\{^{\Phi_0}_{\emptyset}$.

5.4.2 Preservation of Dynamic Type Safety

We know that a dynamically type safe system is not type violating and we know that a statically type safe definition set gives rise to an agent system which is dynamically type safe. To guarantee that this agent system will not become type violating, it is sufficient to show that the operational semantics preserve dynamic type safety.

C.3.6 p333

Theorem 5.4.3 *If* $\Phi_0 \vdash \Gamma$ *and* $\Gamma \mid\!\overset{\triangleright}{_\mathrm{d}}\, g\{^{\Phi}_{\Delta}$ *and* $\Gamma \triangleright g\{^{\Phi}_{\Delta} \longrightarrow g'\{^{\Phi'}_{\Delta'}$ *then* $\Gamma \mid\!\overset{\triangleright}{_\mathrm{d}}\, g'\{^{\Phi'}_{\Delta'}$

C.3.7 p341

Theorem 5.4.4 ***(Type Safety)*** *If* $\Phi_0 \vdash \Gamma$ *and* $\Gamma \triangleright g_\Gamma\{^{\Phi_0}_{\emptyset} \Longrightarrow g\{^{\Phi}_{\Delta}$ *then* $\Gamma \triangleright g\{^{\Phi}_{\Delta}$ *is not type violating.*

5.4.3 Configurations and the Type System

A type checked Oompa program has been shown to be type safe for the agent-based operational semantics. We make use of this result and the results of

Section 4.3 to establish the corresponding result for the configuration-based operational semantics.

The concept of type violation for an agent system can be used to identify the same class of unwanted behaviour for a configuration system. Simply flatten $\Gamma \blacktriangleright K$ to give the agent system $\Gamma \triangleright \mathrm{Flatten}(K)$ and test it for a type violation. Similarly, an Oompa configuration system, $\Gamma, \Phi \blacktriangleright K$, is *dynamically type safe* if

$$\Gamma \models^{\triangleright}_{\mathrm{d}} \mathrm{Flatten}(K\{^{\Phi}_{\emptyset})$$

We write this $\Gamma, \Phi \models^{\blacktriangleright}_{\mathrm{d}} K$.

We may want to consider the dynamic type safety of a subsystem which has received values by interacting with other subsystems. Consequently, we need to express the fact that the values exchanged by these interactions were correctly typed.

A *type safe reception in* Γ, Φ is either a receive $c?\langle v_1, \ldots v_n\rangle$ or an activate $o.m?\langle v_1, \ldots v_n\rangle$ where, in the first case we require

$$\begin{array}{lll} \Gamma, \Phi & \vdash & c\colon \mathtt{InCh}\langle T_1, \ldots T_n\rangle \\ \Gamma, \Phi & \vdash & v_i\colon T_i \qquad\qquad \forall i \end{array}$$

for some T_i, and in the second case we require

$$\begin{array}{lll} \Gamma, \Phi & \vdash & o\colon \mathtt{Intf}\{m?(T_1, \ldots T_n)!\langle S_1, \ldots S_{n'}\rangle\} \\ \Gamma, \Phi & \vdash & v_i\colon T_i \qquad\qquad\qquad\qquad \forall i \end{array}$$

for some T_i, S_i.

C.4.6
p343

Theorem 5.4.5 *Say* $\Phi \vdash \Gamma$ *and* $\Gamma, \Phi \models^{\blacktriangleright}_{\mathrm{d}} K$ *then*

1. *If* $\Gamma, \Phi \blacktriangleright K \longrightarrow K'$ *then* $\Gamma, \Phi \models^{\blacktriangleright}_{\mathrm{d}} K'$.

2. *If* $\Gamma, \Phi \blacktriangleright K \xrightarrow{c!\langle v_1, \ldots v_n\rangle} K'$ *then* $\Gamma, \Phi \models^{\blacktriangleright}_{\mathrm{d}} K'$.

3. *If* $\Gamma, \Phi \blacktriangleright K \xrightarrow{c?\langle v_1,\ldots v_n\rangle} K'$ *and* $c?\langle v_1, \ldots v_n\rangle$ *is a type safe reception in* Γ, Φ*, then* $\Gamma, \Phi \models^{\blacktriangleright}_{\mathrm{d}} K'$.

4. *If* $\Gamma, \Phi \blacktriangleright K \xrightarrow[x:T]{o.m!\langle v_1,\ldots v_n\rangle} K'$ *then* $\Gamma, \Phi \models^{\blacktriangleright}_{\mathrm{d}} K'$.

5. *If* $\Gamma, \Phi \blacktriangleright K \xrightarrow[x:T]{o.m?\langle v_1,\ldots v_n\rangle} K'$ *and* $o.m?\langle v_1, \ldots v_n\rangle$ *is a type safe reception in* Γ, Φ*, then* $\Gamma, \Phi \models^{\blacktriangleright}_{\mathrm{d}} K'$.

C.4.7 p346

Corollary 5.4.6 ***(Type Safety)*** *If* $\Gamma, \Phi \models^{\blacktriangleright}_{\mathrm{d}} K_1$ *and* $\Gamma, \Phi \blacktriangleright K_1 \xRightarrow[L_2]{L_1} K_2$ *and all the receptions in* L_1 *are type safe, then* $\Gamma, \Phi \models^{\blacktriangleright}_{\mathrm{d}} K_2$.

5.5 Comparison with Other Work on Type Systems

We have taken the type system of [PS93a] as a basis for our work. This presents a type system for the π-calculus with recursive channel types and which is statically type checkable. Thus, this type system is similar to the subset of ours where primitive types and interface types are dropped. Our proofs of the termination, soundness and completeness of our subtyping system are based directly upon their technique. For a formal definition of trees we choose an approach based on the definitions in [AC91], which considers recursive function types.

Several of the systems we discussed in Section 2.3 have associated type systems and we consider some here.

The type systems of the various object calculi in [AC96] can be related to the one we are using. In particular, the type system of the language $\mathbf{Ob}_{1<:\mu}$ seems similar, as it uses recursion to bind object self-references. A side-effect of the use of method update in their languages is that some seemingly

desirable subtype relationships cannot be established. Within their system methods can be made non-writable and then, like Oompa, these subtype relationships can be derived.

Although predominantly object-based, class structures for some of their languages are given. However, class and interface definitions are used informally and no equivalent notion to our expansion is described. This means there is no explicit way of dealing with inter-referential definitions in the context of subtyping.

Another system whose type system is worth considering is TyCO [Vas94a], which has a statically checked type system. Objects are primitive in TyCO but class-like entities may be defined using a "let" construction. A corresponding notion of subclassing can similarly be defined. TyCO doesn't have an explicit subtyping system but a similar effect is achieved by making the rule which type checks invocation apply to any object which has an appropriate method of the correct arity. Thus an object of one type may stand for an object of another as the receiver of an invocation.

A number of approaches to the typing and subtyping of object systems are considered in the literature. Four approaches are compared in [BCP97]. One uses recursion to deal with interface types and resembles our approach to typing objects. The other approaches use existential types or a combination of existential and recursive types.

The approach to subtyping object languages using existential types to bind object self-references rather than recursive types is illustrated in [PT94]. The advantage is put forward as a simplification of the underlying theory. In fact, the existential quantifier can be encoded using a universal quantifier which seems an obvious simplification in languages with polymorphism where the universal quantifier is already present. Due to the presence of recursive

channel types and the absence of this kind of polymorphism in our system, however, it seems desirable to use the recursive quantifiers for dealing with self-references. Abadi and Cardelli [AC96] use a combination of recursive and existential operators to give what they call the *self quantifier*. This has interesting subclassing properties.

5.6 Summary

In this chapter we defined Oompa's type system. We identified a class of behaviours, called type violations, that we would not wish an Oompa program to perform. We defined a type safety system which can test a definition set to see if it will ever perform a type violation. Those programs which fail the test are excluded.

We gave our type system a sophisticated subtyping system and an algorithm which can test whether two types lie in the subtype relation. We defined a semantics for types in terms of infinite trees and used this semantics to justify the soundness of the subtyping algorithm. We proved that the type safety system was sound for both the agent-based and configuration-based operational semantics.

Chapter 6

Sequential Process Specifications

We intend our formal method to support the modelling and development of distributed object-oriented systems. So far, we have defined a formal language, Oompa, which can be used to express designs for these systems. Although we can use the semantics to describe the behaviour of Oompa programs, as yet we have no formal way of ensuring this behaviour is suitable. We need some additional theory for discussing and manipulating such behaviour.

In this chapter we introduce a simple language which allows us to express specifications of abstract behaviour. Primarily targeted at supporting a development process, it allows us to develop specifications incrementally using a form of refinement. We demonstrate its use by taking an example from the literature, a scheduler, and incrementally developing its specification.

In Section 6.1, we discuss the issue of how abstract behaviour is expressed. Section 6.2 gives a brief introduction to CCS, including a definition of sequential process expressions whose theory we explore further in Section 6.3. We

introduce our specification language in Section 6.4, providing its syntax and semantics. Section 6.5 defines a notion of refinement for the language. In Section 6.6, we demonstrate the use of our approach by developing the specification of a simple scheduler. Section 6.7 describes how we use the language for specifying the behaviour of Oompa systems. We consider some related work in Section 6.8 and summarise the chapter in Section 6.9.

6.1 Expressing Abstract Behaviour

Oompa is a language for expressing designs for distributed object-oriented systems. The designs it expresses are primarily structural — they describe how the system should be built. The requirements of the system, however, are most likely to be properties of the behaviour of the system rather than properties of its structure. It is appropriate, then, to have some intermediate development step between the requirements and the designs for managing this behavioural information. We will provide a simple behavioural specification language for expressing this type of behaviour. There are three features we will require of this language.

Firstly, we will want this language to be able to express the behaviour of the systems we are to develop. The designs of these systems will be specified in Oompa, so the language will need to express behaviour that corresponds to the actions of Oompa programs. Those actions are precisely the transitions of the labelled transition system defined for the configuration-based operational semantics.

Secondly, we will want it to be quite abstract. We will use this language in the development stage *before* design, so it is important for it to express behaviour without inadvertently biasing the development towards certain

designs. It should focus on behaviour patterns and have only limited support for expressing structure.

Thirdly, we will want to be able to *refine* the expressions of our language. Refinement, in general, is the process of evolving an abstract description of a system into a more concrete one. We will use refinement in a more specific way. When a system has to satisfy several requirements, it is difficult to account for them simultaneously. If the development process is incremental, one can allow each of the requirements to influence the specifications separately. It is then much easier to justify the complete specification as the result of a sequence of individually justified steps. A language which supports refinement allows its expressions to be developed incrementally.

In terms of the first requirement, Oompa's labelled transition system resembles the labelled transition system of the π-calculus. Although, the transitions are slightly different, Oompa programs, like π-calculus expressions, can communicate newly created names. With respect to the second requirement, the π-calculus seems like it would have the right level of abstraction. In fact, it seems likely to us that a language similar to the π-calculus would make a suitable language for specifying the abstract behaviour of Oompa programs.

Unfortunately, the π-calculus fails the third requirement; there is no real support for developing abstract descriptions of behaviour. The formal theory of the π-calculus is dominated by equivalences, i.e. symmetric relationships which assert that two processes can be considered the same with respect to some criteria. This kind of reasoning can be useful in development: starting with a specific pattern of behaviour, the π-calculus allows us to consider how more detailed systems can have that behaviour up to various levels of exactness. However, for specification purposes, we also need to justify the

initial pattern of behaviour. This question seems outside the scope of the π-calculus.

The language we present is our first step towards a π-calculus-like language which supports a form of refinement. The π-calculus is based on CCS; our language extends CCS with an extra level of nondeterminism for supporting this kind of refinement.

6.2 CCS

In this section, we give a brief introduction to CCS (the Calculus of Communicating Systems) [Mil89].

Let Act be the set of atomic actions, represented by a, b, c. These will be the smallest units of behaviour we wish to discuss. Let Pid be a set of process identifiers, represented by s. The core of CCS is a sequential language called *sequential process expressions*. The syntax of sequential process expressions and their definitions are given by[1]:

$$p \quad ::= \quad s\langle v_1, \ldots v_n\rangle \quad | \quad \textstyle\sum_{i \in I} a_i.p_i$$

$$\mathrm{Def} \quad ::= \quad s(r_1, \ldots r_n) \stackrel{\mathrm{def}}{=} p'$$

where p' is a summation and I is a finite index set.

The expression $\sum_{i \in I} a_i.p_i$ describes a process which can perform any of the actions a_i, after which it behaves according to the sequential process expression p_i. When s has definition $s(r_1, \ldots r_n) \stackrel{\mathrm{def}}{=} p$, an expression of the form $s\langle v_1, \ldots v_n\rangle$ has the same behaviour as the expression $p\{\!|{}^{v_1}/r_1, \ldots {}^{v_n}/r_n|\!\}$. Re-

[1] We use a form of summation that is defined over *prefixed* processes as found in [Mil99]. In older presentations of CCS, such as [Mil89], summation was between any process expressions.

Table 6.1: Labelled Transition Rules for Sequential Process Expressions

$$\sum_{i \in I} a_i.p_i \xrightarrow{a_j} p_j \qquad \frac{s(r_1, \ldots r_n) \stackrel{\text{def}}{=} p \qquad p\{\!|{}^{v_1}/r_1 \ldots {}^{v_n}/r_n|\!\} \xrightarrow{a} p'}{s\langle v_1, \ldots v_n\rangle \xrightarrow{a} p'}$$

peated behaviour can be achieved with recursion on definitions. We will use the symbol 0 for the empty sum and often abbreviate $a.0$ as just a.

The semantics of sequential process expressions is given by the labelled transition system defined by the two rules in Table 6.1.

To extend the language of sequential process expressions to give the full CCS language, we first need to put some structure on the set of actions. We assume that $\text{Act} = \mathcal{A} \cup \overline{\mathcal{A}} \cup \{\tau\}$ where $\mathcal{A}$ is a set of *names* and $\overline{\mathcal{A}}$ is an isomorphic set of *co-names*. We will use a, b, c for elements of $\mathcal{A}$ and $\overline{a}$, $\overline{b}$, $\overline{c}$ for elements of $\overline{\mathcal{A}}$. Together, the elements of $\mathcal{A}$ and $\overline{\mathcal{A}}$ are the *observable actions* and we will use λ to represent an observable action. The action τ is called the *silent action*. We will use α to represent an element of Act.

The syntax of CCS is given by:

$$p \;::=\; s\langle v_1, \ldots v_n\rangle \;\;|\;\; \textstyle\sum_{i \in I} \alpha_i.p_i \;\;|\;\; p \mid q \;\;|\;\; p \backslash L \;\;|\;\; p[f]$$

$$\text{Def} \;::=\; s(r_1, \ldots r_n) \stackrel{\text{def}}{=} p'$$

Here L is a set of names and f is a relabelling function from names to names. The new expression, $p \mid q$, describes the behaviour of the two processes, p and q, when placed in parallel. The expression $p\backslash L$ makes those actions of L which occur in p unobservable. The expression $p[f]$ describes a process which behaves like p except that the actions it performs are changed according to f.

The significance of the addition of concurrent process expressions is not

Table 6.2: Labelled Transition Rules for CCS

$$\sum_{i \in I} \alpha_i.p_i \xrightarrow{\alpha_j} p_j \qquad \frac{s(r_1, \ldots r_n) \stackrel{\text{def}}{=} p \qquad p\{\!|{}^{v_1}/r_1 \ldots {}^{v_n}/r_n|\!\} \xrightarrow{\alpha} p'}{s\langle v_1, \ldots v_n \rangle \xrightarrow{\alpha} p'}$$

$$\frac{p \xrightarrow{\alpha} p'}{p \mid q \xrightarrow{\alpha} p' \mid q} \qquad \frac{q \xrightarrow{\alpha} q'}{p \mid q \xrightarrow{\alpha} p \mid q'} \qquad \frac{p \xrightarrow{\lambda} p' \qquad q \xrightarrow{\overline{\lambda}} q'}{p \mid q \xrightarrow{\tau} p' \mid q'}$$

$$\frac{p \xrightarrow{\alpha} p'}{p[f] \xrightarrow{f(\alpha)} p'[f]} \qquad \frac{p \xrightarrow{\alpha} p'}{p \backslash L \xrightarrow{\alpha} p' \backslash L}$$

where, in the last rule, $\alpha, \overline{\alpha} \notin L$.

just that we can express interleavings of the processes' behaviours. Also, we can describe the ways in which they can interact: two concurrent processes, one which can perform an action λ from the set of names and another which can perform the action $\overline{\lambda}$ from the set of co-names can interact by performing both actions together. To an observer, the details of this private interaction are not clear and it appears as though the complete system has performed the silent action τ.

The semantics of CCS is given by the labelled transition system defined by the seven rules in Table 6.2.

In order to determine whether two structurally different processes describe the same behaviour, we compare them in the labelled transition system. CCS uses two different equivalence relations for this: strong bisimulation and weak bisimulation. The idea is to distinguish two processes only if some third party interacting with them could distinguish them[2]. The differ-

[2]However, the distinctions made by these equivalences are finer than any third party, expressed as a process in CCS, could make. "In order to distinguish observationally inequivalent processes like $a.b.c+a.b.d$ and $a.(b.c+b.d)$ one has to make fargoing assumptions

ence between strong and weak bisimulation is that with strong bisimulation, the third party treats silent actions like any other action whereas, with weak bisimulation, the third party doesn't observe them. Because it is designed to be sensitive only to observable differences, weak bisimulation is also called observational equivalence.

A binary relation $\mathcal{S}$ between processes is a *strong bisimulation* if $(p, q) \in \mathcal{S}$ implies, for all $\alpha \in \text{Act}$,

- whenever $p \xrightarrow{\alpha} p'$ then, for some q', $q \xrightarrow{\alpha} q'$ and $(p', q') \in \mathcal{S}$.
- whenever $q \xrightarrow{\alpha} q'$ then, for some p', $p \xrightarrow{\alpha} p'$ and $(p', q') \in \mathcal{S}$.

If there exists a *strong bisimulation* which contains the pair (p, q) then we say p and q are strongly bisimilar and write $p \sim q$.

If there is a sequence of zero or more silent actions that will take p to p', then we write $p \Rightarrow p'$. We write $p \stackrel{\alpha}{\Rightarrow} p'$ if $p \Rightarrow \xrightarrow{\alpha} \Rightarrow p'$.

A binary relation $\mathcal{S}$ between processes is a *weak bisimulation*[3] if $(p, q) \in \mathcal{S}$ implies, for all $\alpha \in \text{Act}$,

- whenever $p \xrightarrow{\alpha} p'$ then, for some q', $q \stackrel{\alpha}{\Rightarrow} q'$ and $(p', q') \in \mathcal{S}$.
- whenever $q \xrightarrow{\alpha} q'$ then, for some p', $p \stackrel{\alpha}{\Rightarrow} p'$ and $(p', q') \in \mathcal{S}$.

If there exists a weak bisimulation which contains the pair (p, q) then we say p and q are *weakly bisimilar* (or observationally equivalent) and write $p \approx q$.

For example, given the following two definitions:

$$s_1(a, b) \stackrel{\text{def}}{=} a.s_1\langle a, b\rangle + b.s_2\langle b, a\rangle \qquad s_2(a, b) \stackrel{\text{def}}{=} a.s_2\langle b, a\rangle + b.s_1\langle a, b\rangle$$

on the power of observation. In any case, it cannot be done by any of the CCS operators." [vG97]. We will consider this issue further in the next section.

[3] This definition is compatible with that defined in Section 4.2.5. The differences are due to the context needed by configurations and the labels used in the labelled transition system of the configuration-based dynamic system.

we can show that $s_1\langle a, b\rangle \sim s_2\langle a, b\rangle$. The following relation is a bisimulation which contains that pair:

$$\mathcal{S} = \begin{array}{l} \{\ (s\langle w,x\rangle, s'\langle y,z\rangle)\ \mid s, s' \in \{s_1, s_2\}, (w,x), (y,z) \in \{(a,b),(b,a)\}\ \} \\ \cup\ \{\ (x.s\langle w,x\rangle + w.s'\langle w,y\rangle, y.s'\langle z,y\rangle + z.s\langle y,z\rangle) \mid s, s' \in \{s_1, s_2\}, \\ \qquad\qquad (w,x), (y,z) \in \{(a,b),(b,a)\}\ \} \end{array}$$

6.2.1 Nondeterminism in CCS

CCS allows non-determinism to be expressed using a combination of its summation operator and its silent action. The expression $\sum_{i \in I} \alpha_i.p_i$ describes a process which can perform any of the actions α_i, after which it behaves according to the CCS expression p_i. Although we know it *can* perform these actions, an ambiguity exists about which action it *will* perform. Perhaps the process will allow its environment to select any of the actions; perhaps it will insist on performing only α_i for some i. These two behaviours differ in their degree of nondeterminism and, for interactive systems, this difference is significant.

The approach CCS takes to this issue is to use the silent action to represent an internal action taken by a process independent of, and unobserved by, its environment. Summations which only involve names and co-names allow their environment to choose and hence are deterministic. If there is a silent action in a choice, then the process can choose to perform that silent action without cooperation from the environment. In this case, the process is nondeterministic since the environment cannot rely on its behaviour.

We find this way of expressing nondeterminism somewhat unintuitive.

Consider the following eight processes:

$$a.(b + c) \qquad a.(\tau.b + c) \qquad a.(b + \tau.c) \qquad a.(\tau.b + \tau.c)$$

$$a.b + a.c \qquad \tau.a.b + a.c \qquad a.b + \tau.a.c \qquad \tau.a.b + \tau.a.c$$

All of these processes can perform the same traces (sequences of observable actions), namely ab and ac, but they all behave differently. For example, the first process performs an a action and then offers the environment the choice of whether it performs a b action or a c action. The second performs an a action and reaches a state where it can allow the environment to choose between b and c or it can decide to only allow c. The fourth performs an a action and then decides to perform either a b or a c.

Observational equivalence is the most important of the equivalences for CCS. When we apply it to the above eight processes, we find that it distinguishes all of them. This is appropriate if we are considering how these processes might represent the abstract behaviour of some more detailed processes. It is less appropriate as a way of specifying their behaviour in terms of how it is experienced by another process.

Lets consider the behaviour of the above eight processes and ask how a CCS process in parallel with them would experience their differences. This approach corresponds to the idea of *testing equivalence* [dNH84]. The most powerful way such a process could discriminate between the above processes would be to try to force them to perform one of their traces but not the other. In order to distinguish between the processes, then, we will consider how they behave when we put them in parallel with one of the following processes: $\overline{a}.\overline{b}.w$ and $\overline{a}.\overline{c}.w$. If the composed system can perform a w action then we say that the test has succeeded; otherwise, the test has failed. In order to have the best chance of differentiating between them, we will assume

that we are allowed to repeatedly perform the tests.

The following table gives the results of the tests, asserting whether the each of the two tests always succeeds or only sometimes succeed.

	$\overline{a}.\overline{b}.w$	$\overline{a}.\overline{c}.w$
$a.(b+c)$	always	always
$a.(\tau.b+c)$	always	sometimes
$a.(b+\tau.c)$	sometimes	always
$a.(\tau.b+\tau.c)$	sometimes	sometimes
$a.b+a.c$	sometimes	sometimes
$\tau.a.b+a.c$	sometimes	sometimes
$a.b+\tau.a.c$	sometimes	sometimes
$\tau.a.b+\tau.a.c$	sometimes	sometimes

We discover that, from an observer's point of view, the last five processes are indistinguishable. However, an observer can distinguish each of the first three from each other and the rest.

Beyond merely asserting information about equivalences, the table also suggests an ordering. Say that we have some situation for which we know that the fourth process is suitable. The above table suggests that any of the eight processes can successfully occupy that role because, where the other processes differ, they differ by being *more* reliable. Now say that we have some context where we know the second process is suitable. This time we can only assume that the first two processes will be suitable.

This suggests that the eight processes might stand in some kind of refine-

ment relation, i.e. one which includes[4]:

$$a.(\tau.b + \tau.c) \sqsubseteq a.b + a.c \sqsubseteq \tau.a.b + a.c \sqsubseteq a.b + \tau.a.c \sqsubseteq \tau.a.b + \tau.a.c$$
$$\tau.a.b + \tau.a.c \sqsubseteq a.(\tau.b + \tau.c)$$
$$a.(\tau.b + \tau.c) \sqsubseteq a.(\tau.b + c) \sqsubseteq a.(b + c)$$
$$a.(\tau.b + \tau.c) \sqsubseteq a.(b + \tau.c) \sqsubseteq a.(b + c)$$

This is close to the sort of refinement we are looking for but we find the way CCS expresses nondeterminism makes the refinements hard to see. Instead, we take inspiration from CSP [Hoa84], which expresses this kind of nondeterminism in an alternative way. Rather than using $+$ and τ, CSP has two binary *choice operators* that make the party responsible for the choice explicit. These are *external choice* ($\Box$) and *internal choice* ($\sqcap$). Under the standard interpretation, $P \Box Q$ is a choice between behaviours P and Q over which the environment has control, whereas $P \sqcap Q$ is a similar choice over which the environment has no control.

The language we introduce is called *sequential process specifications*. The intention is to support a kind of refinement close to that present in CSP, but describe CCS-like behaviour. The expressions of our language are specifications because we think of them specifying ranges of behaviour rather than being descriptions of behaviour themselves.

Our system has operators $\Box$ and $\sqcap$ which are similar, but not identical, to their CSP counterparts. $P \Box Q$ specifies a system which is committed to behave according to both of the specifications P and Q. This resembles external choice, since in both cases we can rely on both behaviours. $P \sqcap Q$ specifies a system which is only committed to behave according to one of P and Q (although it may allow both). This resembles internal choice, as we

[4]In fact, this is precisely what we find if we express the eight processes in CSP and use CSP's failure semantics.

can only rely on one of the behaviours actually occurring.

6.3 Some Theory of Sequential Process Expressions

Our system will be built upon sequential process expressions, as defined in Section 6.2, which we will use to expresses specific patterns of behaviour. In this section we define some theory of this simple language that we will need. From this point on, we will assume that p, q, etc. stand for sequential process expressions (i.e. not expressions of full CCS).

Lemma 6.3.1 $\sim$ *is reflexive, symmetric and transitive.*

It follows from this that strong bisimulation is an equivalence relation. We will be considering sets of sequential process expressions and will often consider the quotient set over $\sim$. We will write the $\sim$-equivalence class of a sequential process expression p as $[p]$.

We will need the following useful representative for $\sim$-equivalence classes of strongly bisimilar sequential process expressions. A sequential process expression is said to be in *normal form* if:

1. It is in summation form.

2. Any two summands with the same action prefix have non-bisimilar continuations.

3. All continuations in the summation should be in definition form, where the bodies of those definitions are in normal form.

For example, $a.s_1\langle a,b,c\rangle + c.s_2\langle c\rangle + a.s_3\langle b,b\rangle$ is in normal form, where

$$s_1(a,b,c) \stackrel{\text{def}}{=} a + c \qquad s_2(c) \stackrel{\text{def}}{=} 0 \qquad s_3(b,c) \stackrel{\text{def}}{=} b + c.s_3\langle c,b\rangle$$

In order to give a concise definition of satisfaction (in Section 6.4.2) we will consider a sequential process expression to be in *near normal form* if it is either in normal form or in definition form where the body of the definition is in normal form. For the purpose of proving the Lemma 6.3.3, we will define *pseudo-normal form* to be normal form where instead of condition 3 we insist that continuations be either in definition form or pseudo-normal form.

Lemma 6.3.2 *Every sequential process expression is strongly bisimilar to a sequential process expression in pseudo-normal form.*

Proof: A sequential process expression is clearly strongly bisimilar to some sequential process expression in summation form p_1. Let p_2 be obtained from p_1 by checking the continuations of summands with the same action prefix. If any are found to be strongly bisimilar then remove all but one. Repeat this process at all the remaining continuations not in definition form. The result is a process in pseudo-normal form which is strongly bisimilar to the original process. ■

Lemma 6.3.3 *Every sequential process expression is strongly bisimilar to a sequential process expression in normal form.*

Proof: The body, p, of every definition is strongly bisimilar to a process in pseudo-normal form by Lemma 6.3.2. Starting at the innermost subterms, and working outwards, we can replace the continuations by definition forms, all of whose bodies are in normal form. This process finally gives us a new body which is in normal form and which is strongly bisimilar to p.

We can apply this procedure to a given sequential process expression and, since all definition bodies are strongly bisimilar to a normal form, this given expression will be in normal form. ■

As it stands, normal forms are not necessarily unique. However, there are many ways we could choose a unique normal form, so we do not concern ourselves with this issue here. Thus, we write the normal form of a sequential process expression p as $\overline{p}$ and, given an equivalence class $[p]$, we will use $\overline{p}$ as its representative.

Syntactically, the $+$ operator separates prefixed processes, not processes themselves. Thus, if we want to describe a process which can behave as described by the sequential process expression p *and* behave as described by the sequential process expression q, we cannot use $+$ directly to join the two expressions. Instead, we define an operator, $\star$, on sequential process expressions in summation form to be:

$$\bigstar_{i \in I} \sum_{j \in J_i} a^i_j.p^i_j = \sum_{i \in I,\ j \in J_i} a^i_j.p^i_j$$

We can extend its definition to sequential process expressions of the form $s\langle v_1, \ldots v_n \rangle$ by looking-up definitions.

Lemma 6.3.4

1. *$\star$ is associative and has 0 as an identity up to equality.*

2. *$\star$ is symmetric up to $\sim$.*

Lemma 6.3.5 *Say p and q are strongly bisimilar sequential process expressions. Then*

1. *If r is a sequential process expression, then $p \star r \sim q \star r$.*

2. *If a is an action, then $a.p \sim a.q$.*

3. $p \star p \sim q$.

Proof: Say $\mathcal{R}$ is a strong bisimulation containing the pair (p, q).

1. Everything is bisimilar to itself, so let $\mathcal{R}'$ be a strong bisimulation containing (r, r). Then $\{(p \star r, q \star r)\} \cup \mathcal{R} \cup \mathcal{R}'$ is a strong bisimulation containing the pair $(p \star r, q \star r)$.

2. In this case, $\{(a.p, a.q)\} \cup \mathcal{R}$ is a strong bisimulation containing the pair $(a.p, a.q)$.

3. $\{(p \star p, q)\} \cup \mathcal{R}$ is a strong bisimulation containing the pair $(p \star p, q)$.

■

6.4 Sequential Process Specifications

The role of the language we introduce in this section is to allow patterns of behaviour to be developed in several steps. We are using sequential process expressions to express specific patterns of behaviour, so our language can be thought to specify sequential process expressions.

The actions of sequential process specifications are drawn from the same set, Act, as for sequential process expressions. They will, however, use different process identifiers for which we will use the symbol S. The syntax of *sequential process specifications* and their definitions is given by:

$$P \quad ::= \quad S\langle v_1, \ldots v_n\rangle \quad | \quad a.P \quad | \quad \Box_{i \in I} P_i \quad | \quad \sqcap_{i \in I} P_i$$

$$\text{Def} \quad ::= \quad S(r_1, \ldots r_n) \stackrel{\text{def}}{=} P'$$

where P' is not in definition form.

We call the $\Box$ and $\sqcap$ operators *choice* and *meet* respectively. Notice that, syntactically, they separate process specifications (unlike $+$ for sequential process expressions). We reuse the symbol 0 for the empty choice and use the symbol $\top$ for the empty meet. An important algebraic point about $\Box$ and $\sqcap$ is that they are not idempotent (see page 172). This means we must be careful with how elements i are drawn from the index sets I in the indexed forms of these operators. We will assume that such elements are drawn from the set only once.

The intention is to allow us to specify a process by asserting that it must have certain behaviour and describing other behaviour that it may or may not have. So a process described by $\Box_{i \in I} P_i$ must be capable of behaving according to all the sequential process specifications P_i and a process described by $\bigsqcap_{i \in I} P_i$ must be capable of behaving according to one or more of the sequential process specifications P_i.

We now give some examples of sequential process specifications. The specification $a \Box b$ is intended to specify the process which offers a choice of a and b, i.e. $a+b$. The specification $a \sqcap b$ is intended to specify processes which can do an a or do a b or offer the choice of both an a and a b; i.e. any of the sequential process expressions a, b and $a + b$. A more complex specification illustrates the interplay between the two operators: $a \Box (b \sqcap (c \Box d))$. Three sequential process expressions which satisfy this specification are $a+b$, $a+c+d$ and $a + b + c + d$. Some similar sequential process expressions with do not satisfy the specification are $a + c$ and $a + b + d$.

We primarily use the $\sqcap$ operator to allow decisions to be deferred, so we can use $a \sqcap b$ to describe a system which we know will offer an a or a b or perhaps offer the choice of both. If desired, we can use our specification system to develop processes with the kind of nondeterminism present in CCS.

Table 6.3: Rules for Equivalence

$$P \equiv P \qquad \frac{P \equiv Q}{Q \equiv P} \qquad \frac{P \equiv Q \quad Q \equiv R}{P \equiv R}$$

$$\frac{P \equiv Q}{a.Q \equiv a.P} \qquad \frac{P \equiv Q}{P \;\square\; R \equiv Q \;\square\; R} \qquad \frac{P \equiv Q}{P \sqcap R \equiv Q \sqcap R}$$

$$\begin{array}{rclrcl}
(P \;\square\; Q) \;\square\; R & \equiv & P \;\square\; (Q \;\square\; R) & (P \sqcap Q) \sqcap R & \equiv & P \sqcap (Q \sqcap R) \\
P \;\square\; Q & \equiv & Q \;\square\; P & P \sqcap Q & \equiv & Q \sqcap P \\
P \;\square\; 0 & \equiv & P & P \sqcap \top & \equiv & P \\
P \;\square\; \top & \equiv & \top & & &
\end{array}$$

$$\frac{S(r_1, \ldots r_n) \overset{\text{def}}{=} P}{S\langle v_1, \ldots v_n\rangle \equiv P\{\!\!|{}^{v_1}/r_1, \ldots {}^{v_n}/r_n|\!\!\}}$$

This can be done simply by using τ as we would any other action. For example, $a \sqcap \tau.b$ specifies any of the behaviours a, $a + \tau b$ or $\tau.b$. The second of these is nondeterministic.

6.4.1 Structural Equivalence

Structural equivalence for sequential process specifications is defined by the fourteen rules in Table 6.3.

We will discover in the next section that certain other equivalences which may initially seem believable are not true in general (with respect to our notion of satisfaction).

6.4.2 Satisfaction

We now precisely define the relationship between specifications and the sequential process expressions which satisfy them.

Say $\mathcal{Q}$ is a binary relation between sequential process specifications and sequential process expressions in near normal form. We say that $\mathcal{Q}$ is a *satisfying simulation* if, for all $(P, p) \in \mathcal{Q}$, we have:

- if $p = s\langle v_1, \ldots v_n \rangle$ where $s(r_1, \ldots r_n) \stackrel{\text{def}}{=} p'$ then $(P, p'\{\!|{}^{v_1}/r_1 \ldots {}^{v_n}/r_n|\!\}) \in \mathcal{Q}$.
- else if $P = S\langle v_1, \ldots v_n \rangle$ where $S(r_1, \ldots r_n) \stackrel{\text{def}}{=} P'$ then $(P'\{\!|{}^{v_1}/r_1 \ldots {}^{v_n}/r_n|\!\}, p) \in \mathcal{Q}$.
- else if $P = a.P'$ then $p = a.p'$ and $(P', p') \in \mathcal{Q}$.
- else if $P = \bigsqcup_{i \in I} P_i$ then $p = \bigstar_{j \in J} p_j$ and there exists some bijective function $f : J \to I$ such that for all $j \in J$, $(P_{f(j)}, p_j) \in \mathcal{Q}$.
- else if $P = \bigsqcap_{i \in I} P_i$ then $p = \bigstar_{j \in J} p_j$ where $J \neq \emptyset$ and there exists some injective function $f : J \to I$ such that for all $j \in J$, $(P_{f(j)}, p_j) \in \mathcal{Q}$.

If p is in normal form and there exists a satisfying simulation which contains the pair (P, p) then we say that p *satisfies* P and write $p \ \tilde{\mathsf{sat}}\ P$. We extend the definition to all sequential process expressions, writing $p \ \tilde{\mathsf{sat}}\ P$ if $\overline{p} \ \tilde{\mathsf{sat}}\ P$.

The following lemma provides a useful intuition for the meaning of $\top$: it is the "impossible specification" which no process can satisfy. This also makes sense when considering that $\top$ is the identity of $\sqcap$. If we specify a process with $P \sqcap \top$, we require it to behave according to P or $\top$ or both. As no process can behave as $\top$, it follows that we are just requiring it to behave according to P.

Lemma 6.4.1

1. *If* p $\mathsf{s\tilde{a}t}$ 0 *then* $p \sim 0$.

2. *There is no sequential process expression* p *such that* p $\mathsf{s\tilde{a}t}$ $\top$.

Lemma 6.4.2 *Say* $p \sim q$. *Then* p $\mathsf{s\tilde{a}t}$ P *iff* q $\mathsf{s\tilde{a}t}$ P.

Proof: This is trivial since p and q will have the same normal form. ■

Lemma 6.4.3

1. *Say* $s(r_1, \ldots r_n) \stackrel{\text{def}}{=} p$.
   Then $s\langle v_1, \ldots v_n\rangle$ $\mathsf{s\tilde{a}t}$ P *iff* $p\{\!|{}^{v_1}/r_1 \ldots {}^{v_n}/r_n|\!\}$ $\mathsf{s\tilde{a}t}$ P.

2. *Say* $S(r_1, \ldots r_n) \stackrel{\text{def}}{=} P$.
   Then p $\mathsf{s\tilde{a}t}$ $S\langle v_1, \ldots v_n\rangle$ *iff* p $\mathsf{s\tilde{a}t}$ $P\{\!|{}^{v_1}/r_1 \ldots {}^{v_n}/r_n|\!\}$.

3. p $\mathsf{s\tilde{a}t}$ $a.P$ *iff* $p \sim a.p'$ *where* p' $\mathsf{s\tilde{a}t}$ P.

4. p $\mathsf{s\tilde{a}t}$ $\bigsqcup_{i \in I} P_i$ *iff* $p \sim \bigstar_{i \in I} p_i$ *where for all* $i \in I$, p_i $\mathsf{s\tilde{a}t}$ P_i.

5. p $\mathsf{s\tilde{a}t}$ $\bigsqcap_{i \in I} P_i$ *iff* $p \sim \bigstar_{j \in J} p_j$ *where* $\emptyset \neq J \subseteq I$ *and for all* $j \in J$, p_j $\mathsf{s\tilde{a}t}$ P_j.

Proof:

1. Say $s(r_1, \ldots r_n) \stackrel{\text{def}}{=} p$. Then $\overline{s\langle v_1, \ldots v_n\rangle} = \overline{p\{\!|{}^{v_1}/r_1 \ldots {}^{v_n}/r_n|\!\}}$. Obviously, then, $s\langle v_1, \ldots v_n\rangle$ $\mathsf{s\tilde{a}t}$ p iff $p\{\!|{}^{v_1}/r_1 \ldots {}^{v_n}/r_n|\!\}$ $\mathsf{s\tilde{a}t}$ P as required.

2. Say $S(r_1, \ldots r_n) \stackrel{\text{def}}{=} P$.

   ($\Rightarrow$) Say p $\mathsf{s\tilde{a}t}$ $S\langle v_1, \ldots v_n\rangle$. Then $\overline{p}$ $\mathsf{s\tilde{a}t}$ $S\langle v_1, \ldots v_n\rangle$. Therefore, there exists a satisfying simulation $\mathcal{Q}$ such that $(S\langle v_1, \ldots v_n\rangle, \overline{p}) \in \mathcal{Q}$. But

then, $(P\{\!|^{v_1}/r_1 \ldots {}^{v_n}/r_n|\!\}, \overline{p}) \in \mathcal{Q}$. So $\overline{p}$ $\mathsf{s\tilde{a}t}$ $P\{\!|^{v_1}/r_1 \ldots {}^{v_n}/r_n|\!\}$ and hence p $\mathsf{s\tilde{a}t}$ $P\{\!|^{v_1}/r_1 \ldots {}^{v_n}/r_n|\!\}$, as required.

($\Leftarrow$) Say p $\mathsf{s\tilde{a}t}$ $P\{\!|^{v_1}/r_1 \ldots {}^{v_n}/r_n|\!\}$. Then $\overline{p}$ $\mathsf{s\tilde{a}t}$ $P\{\!|^{v_1}/r_1 \ldots {}^{v_n}/r_n|\!\}$. So there exists a satisfying simulation $\mathcal{Q}$ such that $(P\{\!|^{v_1}/r_1 \ldots {}^{v_n}/r_n|\!\}, \overline{p})$ $\in \mathcal{Q}$. Let $\mathcal{Q}' = \mathcal{Q} \cup (S\langle v_1, \ldots v_n\rangle, \overline{p})$. Then $\mathcal{Q}'$ is a satisfying simulation. Therefore $\overline{p}$ $\mathsf{s\tilde{a}t}$ $S\langle v_1, \ldots v_n\rangle$ and hence p $\mathsf{s\tilde{a}t}$ $S\langle v_1, \ldots v_n\rangle$ as required.

3. ($\Rightarrow$) Say p $\mathsf{s\tilde{a}t}$ $a.P$. Then $\overline{p}$ $\mathsf{s\tilde{a}t}$ $a.P$. Therefore, there exists a satisfying simulation $\mathcal{Q}$ such that $(a.P, \overline{p}) \in \mathcal{Q}$. Then $\overline{p} = a.p'$ and $(P, p') \in \mathcal{Q}$. So p' $\mathsf{s\tilde{a}t}$ P. Well, $p \sim \overline{p} = a.p'$ as required.

   ($\Leftarrow$) Say that $p \sim a.p'$ where p' $\mathsf{s\tilde{a}t}$ P. Then $\overline{p'}$ $\mathsf{s\tilde{a}t}$ P. Therefore, there exists a satisfying simulation $\mathcal{Q}$ such that $(P, \overline{p'}) \in \mathcal{Q}$. Let $\mathcal{Q}' =$ $\mathcal{Q} \cup (a.P, a.\overline{p'})$. Then $\mathcal{Q}'$ is a satisfying simulation and $a.\overline{p'}$ $\mathsf{s\tilde{a}t}$ $a.P$. But $p \sim a.p' \sim a.\overline{p'}$, so p $\mathsf{s\tilde{a}t}$ $a.P$ as required.

4. ($\Rightarrow$) Say p $\mathsf{s\tilde{a}t}$ $\Box_{i\in I} P_i$. Then $\overline{p}$ $\mathsf{s\tilde{a}t}$ $\Box_{i\in I} P_i$. Therefore, there exists a satisfying simulation $\mathcal{Q}$ such that $(\Box_{i\in I} P_i, \overline{p}) \in \mathcal{Q}$. This means that $\overline{p} = \bigstar_{j\in J} p_j$ and there exists a bijective function $f : J \to I$ such that, for all $j \in J$, $(P_{f(j)}, p_j) \in \mathcal{Q}$. So, for each $j \in J$, p_j $\mathsf{s\tilde{a}t}$ $P_{f(j)}$. For each $i \in I$, let $p'_i = p_{f^{-1}(i)}$. Then $p \sim \overline{p} = \bigstar_{i\in I} p'_i$ and, for all $i \in I$, p'_i $\mathsf{s\tilde{a}t}$ P_i, as required.

   ($\Leftarrow$) Say $p \sim \bigstar_{i\in I} p_i$ where, for each $i \in I$, p_i $\mathsf{s\tilde{a}t}$ P_i. Then, for each $i \in$ I, $\overline{p_i}$ $\mathsf{s\tilde{a}t}$ P_i. Therefore, for each $i \in I$, there exists satisfying simulations $\mathcal{Q}_i$ such that $(P_i, \overline{p_i}) \in \mathcal{Q}_i$. Let $\mathcal{Q} = \bigcup_{i\in I} \mathcal{Q}_i \cup (\Box_{i\in I} P_i, \bigstar_{i\in I}\overline{p_i})$. Then $\mathcal{Q}$ is a satisfying simulation and $\bigstar_{i\in I}\overline{p_i}$ $\mathsf{s\tilde{a}t}$ $\Box_{i\in I} P_i$. But $p \sim \bigstar_{i\in I}\overline{p_i}$, so p $\mathsf{s\tilde{a}t}$ $\Box_{i\in I} P_i$ as required.

5. ($\Rightarrow$) Say p $\mathsf{s\tilde{a}t}$ $\sqcap_{i\in I} P_i$. Then $\overline{p}$ $\mathsf{s\tilde{a}t}$ $\sqcap_{i\in I} P_i$. Therefore, there exists

a satisfying simulation $\mathcal{Q}$ such that $(\bigsqcap_{i\in I} P_i, \overline{p}) \in \mathcal{Q}$. This means that $\overline{p} = \bigstar_{j\in J} p_j$ where $J \neq \emptyset$ and there exists an injective function $f : J \to I$ such that, for all $j \in J$, $(P_{f(j)}, p_j) \in \mathcal{Q}$. So, for each $j \in J$, p_j $\mathsf{s\tilde{a}t}$ $P_{f(i)}$. Let $K = f(J)$. Then $\emptyset \neq K \subseteq I$. For all $k \in K$, let $p'_k = p_{f^{-1}(k)}$. Then, $p \sim \overline{p} = \bigstar_{k\in K} p'_k$ and, for all $k \in K$, p'_k $\mathsf{s\tilde{a}t}$ P_k as required.

($\Leftarrow$) Say that $p \sim \bigstar_{j\in J} p_j$ where $\emptyset \neq J \subseteq I$ and, for all $j \in J$, p_j $\mathsf{s\tilde{a}t}$ P_j. Then, for all $j \in J$, $\overline{p_j}$ $\mathsf{s\tilde{a}t}$ P_j. Therefore, for all $j \in J$, there exists satisfying simulations $\mathcal{Q}_j$ such that $(p_j, \overline{p'_j}) \in \mathcal{Q}_j$. Let $\mathcal{Q} = \bigcup_{j\in J} \mathcal{Q}_j \cup (\bigsqcap_{i\in I} P_i, \bigstar_{j\in J} \overline{p_j})$. Then $\mathcal{Q}$ is a satisfying simulation and $\bigstar_{j\in J} \overline{p_j}$ $\mathsf{s\tilde{a}t}$ $\bigsqcap_{i\in I} P_i$. But $p \sim \bigstar_{j\in J} p_j \sim \bigstar_{j\in J} \overline{p_j}$, so p $\mathsf{s\tilde{a}t}$ $\bigsqcap_{i\in I} P_i$ as required.

■

We will need a result which shows that satisfaction is not sensitive to equivalence.

Lemma 6.4.4 *Say that $\mathcal{Q}$ is a satisfying simulation containing the pair (P, p). Say that $Q \equiv P$. Then there exists a satisfying simulation, $\mathcal{Q}'$, such that $\mathcal{Q} \subseteq \mathcal{Q}'$ and $(Q, p) \in \mathcal{Q}'$.*

Proof: We will, in fact, proof the following equivalent result:

> Say that $\mathcal{Q}$ is a satisfying simulation containing the pair (P, p). Say that $Q \equiv P$ or $P \equiv Q$. Then there exists a satisfying simulation, $\mathcal{Q}'$ such that $\mathcal{Q} \subseteq \mathcal{Q}'$ and $(Q, p) \in \mathcal{Q}'$.

We use induction on the proof tree of $Q \equiv P$ or $P \equiv Q$ (whichever was used). We provide four cases which illustrate the arguments needed.

Say that the proof tree ends with the following step:

$$\frac{Q \equiv P}{P \equiv Q}$$

where (P, p) was the original pair in $\mathcal{Q}$. By the induction hypothesis, there exists a satisfying simulation $\mathcal{Q}'$ such that $\mathcal{Q} \subseteq \mathcal{Q}'$ and $(Q, p) \in \mathcal{Q}$. This is precisely the satisfying simulation we need. The argument also works if (Q, p) was the original pair, so this case has been proved. Note that it was for this case that we needed to use the alternative version of the result.

Say that the proof tree ends with the following step:

$$\frac{P \equiv Q}{a.P \equiv a.Q}$$

where $(a.P, p)$ was the original pair in $\mathcal{Q}$. Since $\mathcal{Q}$ is a satisfying simulation, $p = a.p'$ where $(P, p') \in \mathcal{Q}$. By the induction hypothesis, there exists a satisfying simulation $\mathcal{Q}'$ such that $\mathcal{Q} \subseteq \mathcal{Q}'$ and $(Q, p') \in \mathcal{Q}'$. Then $\{(a.Q, a.p')\} \cup \mathcal{Q}'$ will be a satisfying simulation with the required properties. The argument also works if $(a.Q, p)$ was the original pair, so this case has been proved.

Say that the proof tree is simply an instance of:

$$(P \;\Box\; Q) \;\Box\; R \equiv P \;\Box\; (Q \;\Box\; R)$$

where $((P \;\Box\; Q) \;\Box\; R, p)$ was the original pair in $\mathcal{Q}$. Since $\mathcal{Q}$ is a satisfying simulation, we know that $p = p_1 \star p_2$ where either $(P \;\Box\; Q, p_1), (R, p_2) \in \mathcal{Q}$ or $(P \;\Box\; Q, p_2), (R, p_1) \in \mathcal{Q}$. We will consider only the first of these cases, as the argument is virtually the same for both. Then, since $\mathcal{Q}$ is a satisfying simulation, we know that $p_1 = p_3 \star p_4$ where either $(P, p_3), (Q, p_4) \in \mathcal{Q}$ or $(P, p_4), (Q, p_3) \in \mathcal{Q}$. Again, we consider only the first case. Let

$$\mathcal{Q}' = \mathcal{Q} \cup \{(Q \;\Box\; R, p_4 \star p_2), (P \;\Box\; (Q \;\Box\; R), p)\}$$

Then $\mathcal{Q}'$ is a satisfying simulation with the required properties. The argument also works if $(P \,\Box\, (Q \,\Box\, R), p)$ was the original pair in $\mathcal{Q}$, so this case has been proved.

Say that the proof tree is simply an instance of:

$$P \,\Box\, \top \equiv \top$$

In fact, this case cannot occur, since neither $(P \,\Box\, \top, p)$ nor $(\top, p)$ could be elements of any satisfying simulation. ■

Corollary 6.4.5 *Say $P \equiv Q$. Then p $\tilde{\mathsf{sat}}$ P iff p $\tilde{\mathsf{sat}}$ Q.*

As a consequence of this, we can "close" a satisfying simulation with respect to equivalence.

Corollary 6.4.6 *Say that $\mathcal{Q}$ is a satisfying simulation. Then there exists a satisfying simulation $\overline{\mathcal{Q}}$ such that $\mathcal{Q} \subseteq \overline{\mathcal{Q}}$ and for every $(P, p) \in \overline{\mathcal{Q}}$ and $Q \equiv P$, we have $(Q, p) \in \overline{\mathcal{Q}}$.*

Proof: Say $f(\mathcal{Q}, (P, p), Q)$ returns a satisfying simulation as described in Lemma 6.4.4. Then let

$$\overline{\mathcal{Q}} = \bigcup_{(P,p) \in \mathcal{Q}} f(\mathcal{Q}, (P, p), Q)$$

This will have the required properties. ■

Consider the following candidates for equivalence laws:

$$\begin{aligned}
P &\equiv P \,\Box\, P \\
P &\equiv P \sqcap P \\
P \,\Box\, (Q \sqcap R) &\equiv (P \,\Box\, Q) \sqcap (P \,\Box\, R) \\
P \sqcap (Q \,\Box\, R) &\equiv (P \sqcap Q) \,\Box\, (P \sqcap R)
\end{aligned}$$

They all fail under our notion of satisfaction:

$$a.b + a.c \quad \not\tilde{\mathsf{sat}} \quad a.(b \sqcap c)$$
$$a.b + a.c \quad \tilde{\mathsf{sat}} \quad (a.(b \sqcap c)) \,\Box\, (a.(b \sqcap c))$$

$$a.b + a.c \quad \not\tilde{\mathsf{sat}} \quad a.(b \sqcap c)$$
$$a.b + a.c \quad \tilde{\mathsf{sat}} \quad (a.(b \sqcap c)) \sqcap (a.(b \sqcap c))$$

$$a.b + b + a.c + c \quad \not\tilde{\mathsf{sat}} \quad (a.(b \sqcap c)) \,\Box\, (b \sqcap c)$$
$$a.b + b + a.c + c \quad \tilde{\mathsf{sat}} \quad (a.(b \sqcap c) \,\Box\, b) \sqcap (a.(b \sqcap c) \,\Box\, c))$$

$$a + b \quad \not\tilde{\mathsf{sat}} \quad a \sqcap (b \,\Box\, c)$$
$$a + b \quad \tilde{\mathsf{sat}} \quad (a \sqcap b) \,\Box\, (a \sqcap c)$$

Some of these may, at first, seem slightly strange; for example, excluding the expression $a.b + a.c$ from satisfying the specification $a.(b \sqcap c)$. However, if we compare it to other sequential process expressions which do satisfy the specification, $a.b$, $a.c$ and $a.(b + c)$, we discover that it has a significant difference: it is nondeterministic. Furthermore, from a CCS perspective, it represents nondeterminism that cannot be explained away as the external behaviour of some internal decision making. Without τs to indicate when the choice is made, there will be no more complex processes to which this will be observationally equivalent.

Nevertheless, our sequential process specifications do allow such processes to be specified. For example, with a specification such as $a.b \,\Box\, a.c$.

6.4.3 Semantics

Intuitively, the semantics of a sequential process specification is a set of "acceptable" sequential process expressions — i.e. those expressions which

satisfy it. For technical reasons[5], we define the actual semantics in terms of the quotient set over bisimulation. Thus, the semantics of sequential process specifications are defined as follows:

$$[\![P]\!] = \{q \mid q \ \tilde{\mathsf{sat}}\ P\}/_\sim$$

An equivalent definition, explicitly in terms of $\sim$-equivalence classes, would be:

$$[\![P]\!] = \{[q] \mid q \ \tilde{\mathsf{sat}}\ P\}$$

Lemma 6.4.7 $[\![0]\!] = \{[0]\}$ *and* $[\![\top]\!] = \emptyset$.

Lemma 6.4.8 *If* $P \equiv Q$ *then* $[\![P]\!] = [\![Q]\!]$.

The following lemma suggests an alternative definition of the semantics which would not depend on satisfaction. The difficulty with such an approach is managing recursive processes. They necessitate fixed point solutions. To use fixed points would require us to prove various technical results showing that the fixed points exist and are nontrivial.

Lemma 6.4.9

1. $[\![S\langle v_1, \ldots v_n\rangle]\!] = [\![P\{\!|{}^{v_1}/r_1 \ldots {}^{v_n}/r_n|\!\}]\!]$ *where* $S(r_1, \ldots r_n) \stackrel{\text{def}}{=} P$
2. $[\![a.P]\!] = \{a.p \mid [p] \in [\![P]\!]\}/_\sim$
3. $[\![\Box_{i\in I} P_i]\!] = \{\bigstar_{i\in I} p_i \mid [p_i] \in [\![P_i]\!]\}/_\sim$
4. $[\![\sqcap_{i\in I} P_i]\!] = \{\bigstar_{j\in J} p_j \mid \emptyset \neq J \subseteq I, [p_j] \in [\![P_j]\!]\}/_\sim$

[5] By considering expressions up to bisimulation, we need only consider normal-forms. This simplifies proofs.

Proof: Follows from Lemma 6.4.3. ■

The semantics of even simple specifications can be surprisingly complex. For example, consider the recursive sequential process specification:

$$S(a,b) \stackrel{\text{def}}{=} a \sqcap b.S\langle a,b\rangle$$

If we allow the sets of process identifiers and definitions to be large enough, then the cardinality of $[\![S\langle a,b\rangle]\!]$ is at least that of $\mathbb{R}$. To see this, we can construct non-bisimilar sequential process expressions for every real number x, all of which satisfy $S\langle a,b\rangle$. For each $x \in \mathbb{R}$, $n \in \mathbb{N}$, let x_n be the nth digit of the binary expansion of x and define a sequential process expression

$$s^x_n(a,b) \stackrel{\text{def}}{=} \begin{cases} a + b.s^x_{n+1}\langle a,b\rangle & \text{if } x_n = 1 \\ b.s^x_{n+1}\langle a,b\rangle & \text{otherwise} \end{cases}$$

Then $[s^x_0\langle a,b\rangle] \in [\![S\langle a,b\rangle]\!]$ for all $x \in \mathbb{R}$. But if $x_1 \neq x_2 \in \mathbb{R}$, we have $s^{x_1}_0\langle a,b\rangle \not\approx s^{x_2}_0\langle a,b\rangle$, i.e. $[s^{x_1}_0\langle a,b\rangle] \neq [s^{x_2}_0\langle a,b\rangle]$.

6.5 Refinement

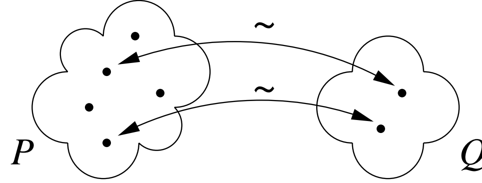
Our notion of refinement is based on the same intuition as our semantics. A sequential process specification describes a set of acceptable sequential process expressions. If this is a set of acceptable expressions, then so is any subset. Hence we give the following definition.

Say P and Q are two sequential process specifications. We say that Q *refines* P if all sequential process expressions which satisfy Q also satisfy P. We write this as $P \sqsubseteq Q$.

Lemma 6.5.1 *$P \sqsubseteq Q$ iff $[\![Q]\!] \subseteq [\![P]\!]$.*

If (ignoring the quotient) we view specifications as representing sets of sequential process expressions, then $P \sqsubseteq Q$ means that any expression which

satisfies Q is strongly bisimilar to some specification which satisfies P. The following picture illustrates this:



6.5.1 Simple Refinement Laws

We can show that this refinement obeys a number of simple *refinement laws*. The first two refinement laws of the following lemma are called *minimum strengthening*.

Lemma 6.5.2

1. $P \sqcap Q \sqsubseteq P \mathbin{\square} (Q \sqcap 0)$
2. $P \sqcap Q \sqsubseteq (P \sqcap 0) \mathbin{\square} Q$
3. $P \sqcap Q \sqsubseteq P$
4. $P \sqcap Q \sqsubseteq Q$
5. $P \sqcap Q \sqsubseteq P \mathbin{\square} Q$

Proof: These can be proved using the representation of sequential process expressions satisfying specifications given by Lemma 6.4.3. ■

Lemma 6.5.3

6. $$\frac{P \equiv P' \quad P' \sqsubseteq Q' \quad Q' \equiv Q}{P \sqsubseteq Q}$$

Proof: Follows from the definitions and Lemma 6.4.5. ■

Lemma 6.5.4

7. $P \sqsubseteq P$

8. $$\frac{P \sqsubseteq Q \qquad Q \sqsubseteq R}{P \sqsubseteq R}$$

Lemma 6.5.5

9. $$\frac{P \sqsubseteq Q}{P \,\Box\, R \sqsubseteq Q \,\Box\, R}$$

10. $$\frac{P \sqsubseteq Q}{P \sqcap R \sqsubseteq Q \sqcap R}$$

11. $$\frac{P \sqsubseteq Q}{a.P \sqsubseteq a.Q}$$

Proof: These can be proved using Lemma 6.4.3. ■

As in other syntactic systems, we can define a syntax of sequential process specification contexts — i.e. sequential process specifications which may contain holes into which other sequential process specifications can be substituted. We don't give a definition of contexts here but they are defined as expected. We can use contexts to elegantly summarises laws 9–11 as follows.

Corollary 6.5.6 (Monotonicity)

12. $$\frac{P \sqsubseteq Q}{C[P] \sqsubseteq C[Q]} \qquad C[\cdot] \textit{ is a sequential process specification context}$$

We also note that the failed equivalence laws discussed above give rise to refinement laws.

Lemma 6.5.7

13. $P \mathbin{\Box} P \sqsubseteq P$

14. $P \sqcap P \sqsubseteq P$

15. $(P \mathbin{\Box} Q) \sqcap (P \mathbin{\Box} R) \sqsubseteq P \mathbin{\Box} (Q \sqcap R)$

16. $(P \sqcap Q) \mathbin{\Box} (P \sqcap R) \sqsubseteq P \sqcap (Q \mathbin{\Box} R)$

Proof: Law 14 is a corollary of Lemma 6.5.2 (law 3). For the other cases, use Lemma 6.4.3. ■

6.5.2 Refining Simulation

We would like a way of checking whether two recursively defined sequential process specifications are in the refinement relation. Although useful, the simple refinement laws we have established do not support this directly. We will need a co-inductive approach.

We introduce a simulation, called a refining simulation, to deal with these cases. We show that it is sound but do not attempt to show that it is complete. Soundness means that it correctly establishes refinement; this is of course necessary. Completeness would mean that any refinement relation between sequential process specifications can be found using a refining simulation. While desirable, we will not need refining simulation to be able to prove any given refinement statement; only those which are likely to occur during development. The refining simulation we provide will be adequate for these cases.

A binary relation $\mathcal{R}$ over sequential process specifications is a *refining simulation* if for all pairs $(P, Q) \in \mathcal{R}$ we have

- if $P = S\langle v_1, \ldots v_n\rangle$ where S has definition $S(r_1, \ldots r_n) \stackrel{\text{def}}{=} P'$ then $(P'\{\!|^{v_1}/r_1 \ldots {}^{v_n}/r_n|\!\}, Q) \in \mathcal{R}$.
- else if $P = a.P'$ then $Q \equiv a.Q'$ and $(P', Q') \in \mathcal{R}$.
- else if $P = \Box_{i \in I} P_i$ then $Q \equiv \Box_{j \in J} Q_j$ and there exists some bijective function $f : J \rightarrow I$ such that for all $j \in J$, $(P_{f(j)}, Q_j) \in \mathcal{R}$.
- else if $P = \bigsqcap_{i \in I} P_i$ where $I \neq \emptyset$, then either:
  - $Q \equiv \bigsqcap_{j \in J} Q_j$ where $J \neq \emptyset$ and there exists some injective function $f : J \rightarrow I$ such that for all $j \in J$, $(P_{f(j)}, Q_j) \in \mathcal{R}$.
  - $Q \equiv \Box_{j \in J} Q_j \,\Box\, (\bigsqcap_{k \in K} Q_k \sqcap 0)$ where $J \neq \emptyset$, $J \pitchfork K$ and there exists some injective function $f : J \cup K \rightarrow I$ such that for all $l \in J \cup K$, $(P_{f(l)}, Q_l) \in \mathcal{R}$.
- else if $P = \top$ then $Q = \top$.

If there exists a refining simulation which contains the pair (P, Q), then we will write $P \lesssim Q$.

The rule for refining meet terms is complex and needs some justification. We allow meets to be refined in two ways. Firstly, by a meet that supports the same or fewer of the optional behaviours. Secondly, by a choice of terms which have become compulsory (those in J) and terms which are still optional (those in K).

Theorem 6.5.8 (Soundness) *If $P \lesssim Q$ then $P \sqsubseteq Q$.*

Proof: Say that $P \lesssim Q$. Clearly, if nothing satisfies Q then $P \sqsubseteq Q$ as required. Alternatively, say that p $\mathsf{s\tilde{a}t}$ Q. We will prove that p $\mathsf{s\tilde{a}t}$ P, from which the main result follows.

Let $q = \overline{p}$. Then q $\mathsf{s\tilde{a}t}$ Q so there must exist a satisfying simulation $\mathcal{Q}$ which contains (Q, q). Then $\overline{\mathcal{Q}}$ (as defined in Corollary 6.4.6) is also a satisfying simulation which contains (Q, q). Since $P \lesssim Q$ there must exists a refining simulation $\mathcal{R}$ which contains (P, Q). Let $\mathcal{Q}'$ be the relational composition of $\mathcal{R}$ and $\overline{\mathcal{Q}}$, i.e.

$$\mathcal{Q}' = \{(R, r) \mid \exists R'.(R, R') \in \mathcal{R}, (R', r) \in \overline{\mathcal{Q}}\}$$

Clearly, $(P, q) \in \mathcal{Q}'$. We will prove that $\mathcal{Q}'$ is a satisfying simulation, which means that q $\mathsf{s\tilde{a}t}$ P implying p $\mathsf{s\tilde{a}t}$ P as required. In what follows we will assume that (P, q) is a general pair from $\mathcal{Q}'$ and assume that Q is such that $(P, Q) \in \mathcal{R}$ and $(Q, q) \in \overline{\mathcal{Q}}$.

Say $q = s\langle v_1, \ldots v_n\rangle$ where s is given the definition $s(r_1, \ldots r_n) \stackrel{\text{def}}{=} q'$. By the definition of satisfying simulation, we know $(Q, q'\{\!|{}^{v_1}\!/r_1 \ldots {}^{v_n}\!/r_n|\!\}) \in \overline{\mathcal{Q}}$. But then $(P, q'\{\!|{}^{v_1}\!/r_1 \ldots {}^{v_n}\!/r_n|\!\}) \in \mathcal{Q}'$ as required in this case.

Say $P = S\langle v_1, \ldots v_n\rangle$ where S is given the definition $S(r_1, \ldots r_n) \stackrel{\text{def}}{=} P'$. By the definition of refining simulation, we know $(P'\{\!|{}^{v_1}\!/r_1 \ldots {}^{v_n}\!/r_n|\!\}, Q) \in \mathcal{R}$, thus $(P'\{\!|{}^{v_1}\!/r_1 \ldots {}^{v_n}\!/r_n|\!\}, q) \in \mathcal{Q}'$ as required in this case.

Say P is of the form $a.P'$. Then $Q \equiv a.Q'$ where $(P', Q') \in \mathcal{R}$. But since $(Q, q) \in \overline{\mathcal{Q}}$ we have $(a.Q', q) \in \overline{\mathcal{Q}}$. Thus $q = a.q'$ where $(Q', q') \in \overline{\mathcal{Q}}$. It follows that $(P', q') \in \mathcal{Q}'$ as required in this case.

Say P is of the form $\Box_{i \in I} P_i$. Then $Q \equiv \Box_{j \in J} Q_j$ and there exists a bijective function $f : J \to I$ such that for all $j \in J$, $(P_{f(j)}, Q_j) \in \mathcal{R}$. But since $(Q, q) \in \overline{\mathcal{Q}}$ we have $(\Box_{j \in J} Q_j, q) \in \overline{\mathcal{Q}}$. Thus $q = \bigstar_{k \in K} q_k$ and there

exists a bijective function $g : K \to J$ such that for all $k \in K$, $(Q_{g(k)}, q_k) \in \overline{\mathcal{Q}}$. But then $f \circ g$ is a bijective function from K to I and for all $k \in K$, $(P_{f \circ g(k)}, q_k) \in \mathcal{Q}'$. This is as required in this case.

Say P is of the form $\bigsqcap_{i \in I} P_i$. First consider the case where $Q \equiv \bigsqcap_{j \in J} Q_j$, $J \neq \emptyset$ and there exists an injective function $f : J \to I$ such that for all $j \in J$, $(P_{f(j)}, Q_j) \in \mathcal{R}$. Since $(Q, q) \in \overline{\mathcal{Q}}$ we have $(\bigsqcap_{j \in J} Q_j, q) \in \overline{\mathcal{Q}}$. Thus $q = \bigstar_{k \in K} q_k$ and there exists an injective function $g : K \to J$ such that for all $k \in K$, $(Q_{g(k)}, q_k) \in \overline{\mathcal{Q}}$. Well $f \circ g$ is an injective function from K to I and for all $k \in K$, $(P_{f \circ g(k)}, q_k) \in \mathcal{Q}'$. This is as required in this case.

Next, we consider the case where $Q \equiv \bigsqcup_{j \in J} Q_j \sqcup (\bigsqcap_{k \in K} Q_k \sqcap 0)$, $J \neq \emptyset$, $J \pitchfork K$ and there exists an injective function $f : J \cup K \to I$ such that for all $l \in J \cup K$, $(P_{f(l)}, Q_l) \in \mathcal{R}$.

Since $(Q, q) \in \overline{\mathcal{Q}}$ we have $(\bigsqcup_{j \in J} Q_j \sqcup (\bigsqcap_{k \in K} Q_k \sqcap 0), q) \in \overline{\mathcal{Q}}$. Thus $q = q_1 \star q_2$ where either $(\bigsqcup_{j \in J} Q_j, q_1)$ and $(\bigsqcap_{k \in K} Q_k \sqcap 0, q_2)$ are in $\overline{\mathcal{Q}}$ or $(\bigsqcup_{j \in J} Q_j, q_2)$ and $(\bigsqcap_{k \in K} Q_k \sqcap 0, q_1)$ are in $\overline{\mathcal{Q}}$. We will only consider the first case, as the other is virtually the same. Then $p_1 = \bigstar_{j' \in J'} q_{j'}$ where there exists a bijective function $g_1 : J' \to J$ such that for all $j' \in J'$, $(Q_{g_1(j')}, q_{j'}) \in \overline{\mathcal{Q}}$. Since g_1 is bijective, we note that $J' \neq \emptyset$. Also, $p_2 = \bigstar_{k' \in K'} q_{k'}$ where there exists an injective function $g_2 : K' \to K$ such that for all $k' \in K'$, $(Q_{g_2(k')}, q_{k'}) \in \overline{\mathcal{Q}}$. Here, we account for the 0 by allowing K' to be $\emptyset$. But then $q = \bigstar_{j' \in J'} q_{j'} \star \bigstar_{k' \in K'} q_{k'} = \bigstar_{l \in J' \cup K'} q_l$. We note that $J' \cup K' \neq \emptyset$. Let $g : J' \cup K'$ be the function which acts like g_1 on J' and acts like g_2 on K'. Then $f \circ g$ is injective and for all $l \in J' \cup K'$ we have $(P_{f \circ g(l)}, q_l) \in \mathcal{Q}'$. This is as required in this case. ■

6.6 Example: A Scheduler

We demonstrate the use of sequential process specifications by taking an example from the literature [Mil89, Mil99]: a simple scheduler. We alter the informal requirements, as given in the references, in order to emphasise the way a specification can emerge in our system. To motivate the discussion, we give the problem an anecdotal setting. In what follows we will use a standard abbreviated notation, leaving out parameter lists in definitions and processes in definition form.

6.6.1 Informal Requirements

We are hired to design a scheduler for a company which has n machines and wishes to use them efficiently. The scheduler is responsible for starting the machines when they are free and acknowledging their request to stop when have finished working. The company requests two properties of the scheduler we provide:

Time is money: If possible, the machines should run in parallel so more work is done is less time.

Electricity costs money: The scheduler should be willing to acknowledge a machine's signal to stop at any stage, so that it can be switched off. This saves electricity.

6.6.2 Model

We will assign each of the n machines a number between 1 and n. Our scheduler has two responsibilities, issuing some kind of signal to start machines and acknowledging their signals to stop. We will model these with

actions $a_1, \ldots a_n, b_1, \ldots b_n$, where a_i signals machine i to start and action b_j acknowledges machine j's signal to stop.

An obvious consequence of our model is:

1. The scheduler will only ever perform one of the actions $a_1, \ldots a_n$ or $b_1, \ldots b_n$.

Let X be the set of machines currently working. Then two similarly obvious requirements are:

2. The scheduler will only perform action a_i if $i \notin X$.

3. The scheduler will only perform action b_j if $j \in X$.

The above three requirements describe the general class of schedulers the company is interested in.

Of the two properties the company want the scheduler to have, the second is obvious and becomes our condition 4.

4. The scheduler should at all times be willing to perform b_j actions for each $j \in X$.

The first property they mention is more difficult to formalise, as it expresses a optimisation rather than a constraint.

5. Whenever possible, the scheduler should perform some a_i action for $i \notin X$.

6.6.3 Specification

We now develop our specification using sequential process expressions. The simulations involved in this section are given in Appendix D.

We begin by stating the obvious: by requirement 1, any suitable scheduler will have to satisfy the following specification:

$$\mathrm{Sched}^0 \stackrel{\mathrm{def}}{=} \bigsqcap_{j\in 1..n} b_j.\mathrm{Sched}^0 \sqcap \bigsqcap_{i\in 1..n} a_i.\mathrm{Sched}^0$$

We can immediately improve on this, using requirements 2 and 3 to give us:

$$\mathrm{Sched}^1_X \stackrel{\mathrm{def}}{=} \bigsqcap_{j\in X} b_j.\mathrm{Sched}^1_{X-j} \sqcap \bigsqcap_{i\notin X} a_i.\mathrm{Sched}^1_{X\cup i}$$

where $X \subseteq \{1..n\}$ and the scheduler is thought to start in state $\mathrm{Sched}^1_\emptyset$. It is easy to show that $\mathrm{Sched}^0 \sqsubseteq \mathrm{Sched}^1_\emptyset$. The specification $\mathrm{Sched}^1_\emptyset$ guarantees that at any stage either an appropriate start action or an appropriate acknowledgement will be performed.

The specification $\mathrm{Sched}^1_\emptyset$ will not guarantee requirements 4 or 5. For example,

$$\mathrm{design}^1 \stackrel{\mathrm{def}}{=} a_1.a_2.b_2.b_1.\mathrm{design}^1$$

fails requirement 4 since, after the a_1 and a_2 actions, a request by machine 1 to stop will be ignored. Yet $\mathrm{design}^1\ \widetilde{\mathsf{sat}}\ \mathrm{Sched}^1_\emptyset$. Also, the following design satisfies $\mathrm{Sched}^1_\emptyset$, yet fails requirement 5:

$$\mathrm{design}^2 \stackrel{\mathrm{def}}{=} a_1.b_1.\mathrm{design}^2$$

Requirement 4 means that there is, in fact, no freedom for which actions b_j for $j \in X$ we perform — we must perform them all. The following represents the minimum strengthening of the specification to achieve this:

$$\mathrm{Sched}^2_X \stackrel{\mathrm{def}}{=} \Box_{j\in X} b_j.\mathrm{Sched}^2_{X-j} \,\Box\, (\bigsqcap_{i\notin X} a_i.\mathrm{Sched}^2_{X\cup i} \sqcap 0)$$

We have $\mathrm{Sched}^1_\emptyset \sqsubseteq \mathrm{Sched}^2_\emptyset$ and $\mathrm{design}^1\ \not\widetilde{\mathsf{sat}}\ \mathrm{Sched}^2_\emptyset$. Unfortunately, we still have $\mathrm{design}^2\ \widetilde{\mathsf{sat}}\ \mathrm{Sched}^2_\emptyset$ so we must further constrain the specification.

To satisfy requirement 5 we can insist that, in any suitable state, some available machine is started:

$$\text{Sched}^3_X \stackrel{\text{def}}{=} \begin{cases} \bigsqcup_{j \in X} b_j.\text{Sched}^3_{X-j} \;\Box\; \bigsqcap_{i \notin X} a_i.\text{Sched}^3_{X \cup i} & \text{if } X \neq \{1..n\} \\ \bigsqcup_{j \in X} b_j.\text{Sched}^3_{X-j} & \text{otherwise} \end{cases}$$

We can easily show that $\text{Sched}^2_\emptyset \sqsubseteq \text{Sched}^3_\emptyset$. In fact, we now have a specification for a suitable scheduler. As the company want an implementable design, we give them the following:

$$\text{design}^3_X \stackrel{\text{def}}{=} \begin{cases} \sum_{j \in X} b_j.\text{design}^3_{X-j} + a_{(\min \overline{X})}.\text{design}^3_{X \cup (\min \overline{X})} & \text{if } X \neq \{1..n\} \\ \sum_{j \in X} b_j.\text{design}^3_{X-j} & \text{otherwise} \end{cases}$$

which satisfies $\text{Sched}^3_\emptyset$ and, at any stage, will start the lowest numbered available machine.

6.6.4 Further Requirements

After some time, the company contacts us again. Although they accept that our scheduler has the properties that they requested, they have encountered a problem. Over time, our scheduler was placing a heavier load on the lowered numbered machines. This lead to unpredictable reliability and an expensive maintenance cycle. They requested a new scheduler with the following property:

Maintenance costs money: In order to have a predictable maintenance cycle, the work should be shared amongst the machines as evenly as possible.

By discussing the possibilities with the company, it was decided that condition 6, below, describes a suitable approach to their new request.

6. A scheduler performs the a_i actions in cyclic order only, i.e.

$$a_1, \ldots a_n, a_1 \ldots a_n, a_1 \ldots$$

If necessary, this requirement may take precedence over condition 5.

6.6.5 Respecifying the Scheduler

To consider how to specify this new scheduler, we return to specification $\mathrm{Sched}^2_\emptyset$. This step backwards is necessary to ensure the correctness condition 6 influences the specification before the optimisation condition 5.

To manage condition 6, our scheduler must keep track of which machine it is meant to start next. We index its states with the set of working machines, X, and the number of the machine to start next, i. In the definition, we consider addition on the integer index to be mod n.

$$\mathrm{Sched}^4_{i,X} \stackrel{\mathrm{def}}{=} \Box_{j\in X} b_j.\mathrm{Sched}^4_{i,X-j} \;\Box\; (\textstyle\bigsqcap_{(k\notin X, k=i)} a_k.\mathrm{Sched}^4_{k+1,X\cup k} \sqcap 0)$$

We can show that $\mathrm{Sched}^2_\emptyset \sqsubseteq \mathrm{Sched}^4_{1,\emptyset}$. However, we now have to take account of condition 5. The following design satisfies $\mathrm{Sched}^4_{i,X}$ but fails to satisfy this requirement:

$$\mathrm{design}^4_i \stackrel{\mathrm{def}}{=} a_i.b_i.\mathrm{design}^4_{i+1}$$

We need to strengthen the specification so that when the scheduler had an option to perform the a_i action, it now must be willing to do so:

$$\mathrm{Sched}^5_{i,X} \stackrel{\mathrm{def}}{=} \begin{cases} \Box_{j\in X} b_j.\mathrm{Sched}^5_{i,X-j} \;\Box\; a_i.\mathrm{Sched}^5_{i+1,X\cup i} & i \notin \mathrm{X} \\ \Box_{j\in X} b_j.\mathrm{Sched}^5_{i,X-j} & i \in \mathrm{X} \end{cases}$$

Naturally $\mathrm{Sched}^4_{1,\emptyset} \sqsubseteq \mathrm{Sched}^5_{1,\emptyset}$. As this process specification has no meet subterms, an obvious transformation gives us the following sequential process

expression which satisfies it:

$$\text{design}^5_{i,X} \stackrel{\text{def}}{=} \begin{cases} \sum_{j\in X} b_j.\text{design}^5_{i,X-j} + a_i.\text{design}^5_{i+1,X\cup i} & i \notin \text{X} \\ \sum_{j\in X} b_j.\text{design}^5_{i,X-j} & i \in \text{X} \end{cases}$$

We deliver this design to the company.

6.7 Using Sequential Process Specifications with Oompa

Currently, we have considered our set of actions, Act, to contain atomic actions, written a, b, c etc. The actions of Oompa are much richer and correspond to the transitions of its labelled transition system. There is a silent action, a send action, a receive action, an invoke action and an activate action:

$$\longrightarrow \qquad \xrightarrow{c!\langle v_1,\ldots v_n\rangle} \qquad \xrightarrow{c?\langle v_1,\ldots v_n\rangle} \qquad \xrightarrow[x:T]{o.m!\langle v_1,\ldots v_n\rangle} \qquad \xrightarrow[x:T]{o.m?\langle v_1,\ldots v_n\rangle}$$

In order to discuss these actions, we follow a standard approach. *Message-passing CCS* is a variant of CCS that extends the atomic actions of CCS to communication actions of the form $c!\langle a\rangle$ and $c?(b)$. Assuming that those values communicated belong to a finite type, we can build message-passing into ordinary CCS using the following sugarings:

$$\begin{aligned} c!\langle a\rangle.p &= \overline{c_a}.p \\ c?(b).p &= \textstyle\sum_{a\in T} c_a.p\{\!|^a/b|\!\} \end{aligned}$$

where for each a, c_a is thought to be a single atomic CCS action.

For Oompa's actions we intend to use the same trick. To allow these actions to appear in sequential process expressions, we will write transitions

in the style of the labelled transition system and use the following sugarings to interpret them either as atomic actions or summations.

$$\begin{aligned}
\xrightarrow{} .p &= \tau.p \\
\xrightarrow{c!\langle v_1,\ldots v_n\rangle} .p &= \overline{c_v_1_\cdots_v_n}.p \\
\xrightarrow{c?\langle r_1,\ldots r_n\rangle} .p &= \textstyle\sum_{v_1\in T_1,\ldots v_n\in T_n} c_v_1_\cdots_v_n.p\{\!|{}^{v_1}/r_1 \ldots {}^{v_n}/r_n|\!\} \\
\xrightarrow[x:T]{o.m!\langle v_1,\ldots v_n\rangle} .p &= \overline{o_m_v_1_\cdots_v_n_x}.p \\
\xrightarrow[x:T]{o.m?\langle r_1,\ldots r_n\rangle} .p &= \\
&\textstyle\sum_{v_1\in T_1,\ldots v_n\in T_n,x'\in x} o_m_v_1_\cdots_v_n_x'.p\{\!|{}^{v_1}/r_1 \ldots {}^{v_n}/r_n, {}^{x'}/x|\!\}
\end{aligned}$$

Similarly, to allow these actions to appear in sequential process specifications, we use the following sugarings.

$$\begin{aligned}
\xrightarrow{} .p &= \tau.p \\
\xrightarrow{c!\langle v_1,\ldots v_n\rangle} .p &= \overline{c_v_1_\cdots_v_n}.p \\
\xrightarrow{c?\langle r_1,\ldots r_n\rangle} .p &= \Box_{v_1\in T_1,\ldots v_n\in T_n} c_v_1_\cdots_v_n.p\{\!|{}^{v_1}/r_1 \ldots {}^{v_n}/r_n|\!\} \\
\xrightarrow[x:T]{o.m!\langle v_1,\ldots v_n\rangle} .p &= \overline{o_m_v_1_\cdots_v_n_x}.p \\
\xrightarrow[x:T]{o.m?\langle r_1,\ldots r_n\rangle} .p &= \\
&\Box_{v_1\in T_1,\ldots v_n\in T_n,x'\in x} o_m_v_1_\cdots_v_n_x'.p\{\!|{}^{v_1}/r_1 \ldots {}^{v_n}/r_n, {}^{x'}/x|\!\}
\end{aligned}$$

Sequential process expressions and specifications do not have a sufficiently powerful mechanism for the communication of new names (which is why, ultimately, a language like the π-calculus is needed). To manage the return channels in method invocations, we will assume that they are generated by the invoking process and received by the activated process.

Using these sugarings, we can specify processes with sequential process specifications whose actions correspond to the transitions of an Oompa configuration. To ensure an Oompa configuration actually does have a behaviour which satisfies a specification, we will show that its behaviour is weakly bisimilar to a sequential process expression which satisfies the specification. We

choose to use weak bisimulation since it allows the configuration to perform extra internal actions in order to satisfy the specification. Also, it is our current choice of semantic equivalence for Oompa configurations.

When using weak bisimulation as a semantic equivalence for Oompa configurations, a context consisting of a definition set and a type dictionary is required (see Section 4.2.5). This is also required for our implementation relation but we will typically allow this context be left implicit. Thus, our *implementation relation* between Oompa configurations and sequential process expressions will be $\approx\tilde{\mathsf{sat}}$:

$$K \approx\tilde{\mathsf{sat}}\ P = \exists p.(K \approx p) \wedge (p\ \tilde{\mathsf{sat}}\ P)$$

6.8 Related Work

Our language is a first step towards a refinable specification language of π-calculus-like expressions. This is the overriding reason why we have built it upon (the core of) CCS. Nevertheless, there are potential alternatives to our approach and we consider them here: CCS itself, CSP and TCCS.

Our prime objection to the π-calculus was its focus on equivalence. Like the π-calculus, the formal theory of CCS is mainly based on equivalences. However, other development techniques have been proposed for CCS which do give support for a incremental development.

Two such methods are discussed in [Mil89]. *Process logic* is a modal logic which can express behavioural properties of processes. The method can be used incrementally: by conjoining expressions of the logic together we get new, more demanding properties. The other method uses the fact that we can obtain an abstraction of the behaviour of a process by restricting its visible actions to a limited set. We can specify a property of a process by

insisting that its behaviour, abstracted in this way, is equivalent to some other process. By increasing the set of such properties we require, we obtain a method somewhat like process logic. The similarity of these approaches to our own is that the outcome of a specification will be sequential process expression[6]. One difference is that these techniques are further removed from the behaviour being specified then ours. This can lead to a problem: with these approaches it is possible to unknowingly specify unrealisable behaviour.

Theory of testing [dNH84] which considers processes in terms of how they can interact with an observer. This is achieved by viewing an observer, coupled with a satisfactory notion of successful completion, as a test. Processes are considered equivalent if they pass exactly the same set of tests. In fact, this notion of equivalence is defined in terms of two preorders: "the first is formulated in terms of the ability to respond positively to a test, the second in terms of the inability not to respond positively to a test". The intersection of the two preorders generates a third preorder which might be useable as a form of refinement relation similar to ours.

CSP does have support for refinement — a process may be refined by another more deterministic process. The interpretation of CSP's language is fixed because the algebra is designed to match a specific set of models. This makes the meaning of CSP's expressions very rigid which, for our purposes, is a disadvantage; we want our language to express behaviour as abstractly as possible.

Testing CCS (TCCS[7]) [dNH87], is a variant of CCS[8] which is designed

[6]More accurately, these approaches should be considered as specifying a full CCS expression. However, the behaviour (as opposed to structure) of that CCS expression would be describable with a sequential process expression.

[7]The name TCCS is sometimes used for a different CCS variant called Timed CCS.

[8]From the perspective of testing equivalence, TCCS is semantically equivalent to CCS

specifically for the notion of testing equivalence. Rather than using CCS' $+$ and τ to express nondeterminism, TCCS uses pair of operators $\Box$ and $\oplus$ to express internal and external nondeterminism. Its theory is based on the testing preorders discussed above.

A similarity between sequential process specifications, CSP and TCCS is the resemblances of their choice operators (syntactically and, to a lesser extent, semantically). One difference between our method and the other two is that the laws we reject in Section 6.4.2 are true in both those languages.

One practical advantage of our system over TCCS and CSP is that our meet operator, which corresponds to their internal choice, has an identity: $\top$. We regularly make use of this fact. For instance, in the scheduler example the second specification was given as:

$$\mathrm{Sched}^1_X \stackrel{\mathrm{def}}{=} \bigsqcap_{j \in X} b_j.\mathrm{Sched}^1_{X-j} \sqcap \bigsqcap_{i \notin X} a_i.\mathrm{Sched}^1_{X \cup i}$$

If the set X is either empty or full (contains all the indices 1..n) then the left or right subterm of this expression becomes an empty meet and hence equals $\top$. Given that $\top$ is the identity of $\sqcap$, that subterm can be viewed as absent from the complete expression. Without a way of discussing empty meet expressions, this simple specification would need three cases: one for when X is empty, one for when X is neither empty nor full and one for when X is full.

6.9 Summary

In this chapter, we presented sequential process specifications; a new formalism which we will use as a behavioural specification language for Oompa.

itself. This not the case if observational equivalence is used.

We defined sequential process expressions and discussed some of their theory. These constitute the core of CCS and we consider them to be suitable objects for expressing abstract behaviour. We then defined the syntax of sequential process specifications and defined their semantics in terms of a notion of satisfaction: a sequential process specification specifies those sequential process expressions which satisfy it. In order that our language allows specifications to be incrementally developed, we defined a refinement relation between specifications. We demonstrated the use of our method by deriving the specification of a simple scheduler from informal requirements. We showed how sequential process specifications can be used to specify the behaviour of Oompa configurations. We also considered some alternatives to our specification language.

Chapter 7

A Development in Oompa

In this chapter, we give an example which suggests how sequential process specifications and Oompa might be used in actual development. We apply our approach to a standard example from the literature, namely the development of a concurrent dictionary design.

A *dictionary* is a structure which allows information to be stored and retrieved based on an index[1]. A dictionary provides its service to clients by providing two operations: get, which returns the current value associated with an index, and set, which associates a new value with an index. We will develop an Oompa object which provides this service to multiple concurrent clients.

The purpose of this chapter is to consider how designs might be developed within our formal method and not to develop new ones. In fact, the dictionary is a very standard structure and it is likely that the designs we develop have been studied comprehensively. Nevertheless, in order to gain a clear appreciation of our formal method, we followed through the development in this chapter without particular reference to existing work.

[1]Other names for this type of structure include look-up table, symbol table and array.

In Section 7.1 we determine the behavioural requirements of a concurrent dictionary. We use sequential process specifications to develop two behavioural specifications for such a dictionary object in Section 7.2. In Section 7.3 we provide two Oompa classes whose objects will satisfy the specifications. We summarise the chapter in Section 7.4.

7.1 Behavioural Requirements of a Concurrent Dictionary

In this section, we establish the behavioural requirements on Oompa objects which implement a concurrent dictionary. The first step is to give a precise characterisation of what a dictionary actually is. Next, we consider what it means to be a dictionary object. We will provide an interface that such an object must implement and describe the semantics of the dictionary's methods. This is non-trivial, so we simplify the task by first giving the behavioural requirements of a sequential dictionary. We then define the behaviour of a concurrent dictionary object partially in terms of the behaviour of the sequential dictionary object.

7.1.1 Abstract Data Type for a Dictionary

In order to characterise the notion of a dictionary formally, but abstractly, we will use an abstract data type. This captures the pure functional behaviour of the operations without any implementation details. We will assume that the information is of type Value and is indexed by elements of type Key. For simplicity, we assume that Value includes an element $\perp$ which we will use to

indicate "no value". Then, the abstract data type Dict is given by:

$$\begin{aligned} \mathsf{D_{init}} &: \mathsf{Dict} \\ \mathsf{get} &: \mathsf{Dict} \times \text{Key} \to \text{Value} \\ \mathsf{set} &: \mathsf{Dict} \times \text{Key} \times \text{Value} \to \mathsf{Dict} \end{aligned}$$

$$\begin{aligned} \mathsf{get}(\mathsf{D_{init}}, k) &= \bot & \forall k \in \text{Key} \\ \mathsf{get}(\mathsf{set}(D, k, v), k') &= \begin{cases} v & k' = k \\ \mathsf{get}(D, k') & \text{otherwise} \end{cases} & \forall k, k' \in \text{Key}, v \in \text{Value} \end{aligned}$$

This definition defines a type, Dict, and an instance of that type, $\mathsf{D_{init}}$, which corresponds to a dictionary in its initial state. It also gives the specification of two functions, get and set.

7.1.2 Sequential Object-Oriented Dictionary

Before attempting to describe the rather complex behaviour of a concurrent object-oriented dictionary, it is preferable to consider the consequences of object-orientation without concurrency. Unlike the abstract data type, which was defined in pure functional terms, a dictionary object has independent state and side-effecting operations.

We will use the following Oompa interface for dictionary objects:

```
interface Dictionary
{
  // returns the current value associated with key k
  get?(k:Key)!<v:Value>
  // associates value v with key k
  set?(k:Key,v:Value)!<>
}
```

The behaviour of objects which implement this interface can be expressed using notation similar to Oompa's labelled transition system: An invocation on the dictionary's get method can be written $\xrightarrow[x]{o.\mathtt{get}?\langle k\rangle}$ and its reply is written $\xrightarrow{x!\langle v\rangle}$. An invocation on the dictionary's set method can be written $\xrightarrow[x]{o.\mathtt{set}?\langle k,v\rangle}$ and its reply is written $\xrightarrow{x!\langle\rangle}$. Although, an actual dictionary is likely to have internal behaviour represented by silent actions, our requirements will only need to discuss these four observable actions. Rather than using sequences of actions, therefore, we will use *traces*, which are sequences of observable actions only.

Our behavioural requirements are of two sorts. First, there are requirements that insist that the dictionary object be able and willing to perform certain actions. For example, we will want the dictionary object to eventually reply to an invocation on its get method. Second, there are requirements that constrain the sequences of actions that the dictionary object may perform. For example, we will want the reply that the dictionary object sends to be a meaningful response.

As can be seen from the abstract data type, a set operation on one key does not affect the behaviour of the dictionary with respect to other keys. Moreover, performing a get operation should not affect the behaviour of the dictionary. One expected behaviour of a dictionary object, therefore, is that that operations on different keys do not affect each other. This observation helps us simplify the constraints, since we can write them in terms of the subsequences for each key.

Given a trace t, let $\pi_k(t)$ be the subsequence consisting of just the invocations on key k and their replies. We will want the following predicates on

a trace t on key k:

$$\begin{aligned}
\mathrm{GetGet}_k(t) &= \forall s, s', x, x', v, v'. \\
&\quad ((t = s \xrightarrow[x]{\texttt{o.get?}\langle k\rangle} \xrightarrow{x!\langle v\rangle} \xrightarrow[x']{\texttt{o.get?}\langle k\rangle} \xrightarrow{x'!\langle v'\rangle} s') \Rightarrow v = v')
\end{aligned}$$

$$\begin{aligned}
\mathrm{SetGet}_k(t) &= \forall s, s', x, x', v, v'. \\
&\quad ((t = s \xrightarrow[x]{\texttt{o.set?}\langle k,v\rangle} \xrightarrow{x!\langle\rangle} \xrightarrow[x']{\texttt{o.get?}\langle k\rangle} \xrightarrow{x'!\langle v'\rangle} s') \Rightarrow v = v')
\end{aligned}$$

$$\mathrm{InitGet}_k(t) = \forall s, x, v.((t = \xrightarrow[x]{\texttt{o.get?}\langle k\rangle} \xrightarrow{x!\langle v\rangle} s) \Rightarrow v = \bot)$$

The following are the requirements we put on the dictionary object's behaviour. The first and fourth requirements constrain the behaviour and the second and third requirements oblige the dictionary to perform certain behaviours.

1. The dictionary will only ever perform one of the actions:

$$\xrightarrow[x]{\texttt{o.get?}\langle k\rangle}, \xrightarrow[x]{\texttt{o.set?}\langle k,v\rangle}, \xrightarrow{x!\langle v\rangle}, \xrightarrow{x!\langle\rangle}$$

   where, in the first two actions, x is required to be a new name.

2. If the dictionary performs the action $\xrightarrow[x]{\texttt{o.get?}\langle k\rangle}$ then it next performs $\xrightarrow{x!\langle v\rangle}$ for some v.

3. If the dictionary performs the action $\xrightarrow[x]{\texttt{o.set?}\langle k,v\rangle}$ then it next performs $\xrightarrow{x!\langle\rangle}$.

4. If the dictionary can perform trace t, then

$$\forall k.(\mathrm{GetGet}_k(\pi_k(t)) \wedge \mathrm{SetGet}_k(\pi_k(t)) \wedge \mathrm{InitGet}_k(\pi_k(t)))$$

A sequential object which behaves according to all these requirements should be considered an acceptable sequential dictionary object.

7.1.3 Concurrent Object-Oriented Dictionary

In this section we extend the approach and describe the behavioural requirements of a concurrent object-oriented dictionary. Again, we can consider the behaviour of the dictionary in terms of traces of the four actions $\xrightarrow[x]{o.\mathtt{get}?\langle k\rangle}$, $\xrightarrow{x!\langle v\rangle}$, $\xrightarrow[x]{o.\mathtt{set}?\langle k,v\rangle}$ and $\xrightarrow{x!\langle\rangle}$. Intuitively, we will be requiring the same properties of this dictionary; those expressed by the predicates $\mathrm{GetGet}(\cdot)$, $\mathrm{SetGet}(\cdot)$ and $\mathrm{InitGet}(\cdot)$. However, there are two main reasons why the predicates will not apply directly to the concurrent dictionary objects.

Firstly, the predicates we defined in the previous section only apply to traces where none of the invocation-reply pairs overlap. We will call such traces *lined-up* and write $\mathrm{LinedUp}(t)$ if t is a lined-up trace. All of the traces of a sequential dictionary object are lined-up but we cannot make this assumption about a concurrent dictionary object.

Secondly, the predicates from the previous section only apply to traces where all (or all but the last) of the invocations have been replied to. We will call such a trace *completed* and write $\mathrm{Completed}(t)$ if t is a completed trace. In the concurrent case, we need to take unreplied invocations into consideration: when considering a given trace, it may contain invocations representing operations which have been performed but have not yet replied.

To handle the case where two or more invocation-reply pairs are interleaved, we observe that the processing of a request is atomic from the perspective of traces — no trace of a correct dictionary object would allow a partially performed request to be observed. Consequently, we can view an overlapping trace as representing some lined-up trace where the order of the invocations-reply pairs in the latter represent the order that the requests were processed in the former. To handle incomplete traces, we view them as initial subtraces of some complete trace.

We write $t \rightsquigarrow s$ if t can be transformed into s by interchanging any neighbouring actions except moving an invocation before a reply or moving a reply before its corresponding invocation. We use $\rightsquigarrow$ to find all the possible lined-up traces that it might represent; it allows interleaving invocation-reply pairs to be reordered in any of the possible lined-up ways:

$$\mathrm{PossTraces}(t) = \{s \mid \mathrm{LinedUp}(s) \ \wedge \ t \rightsquigarrow s\}$$

Our requirement on a trace is therefore that it can be completed and interpreted as some lined-up trace which satisfies the predicates for sequential dictionary objects. Formally, we will use the following predicate on a trace t:

$$\begin{aligned}&\exists t'.(\mathrm{Completed}(tt') \wedge \exists s \in \mathrm{PossTraces}(tt').\\ &\qquad \forall k.(\mathrm{GetGet}_k(\pi_k(s)) \wedge \mathrm{SetGet}_k(\pi_k(s)) \wedge \mathrm{InitGet}_k(\pi_k(s))))\end{aligned}$$

The requirements we give for the concurrent dictionary resemble those we gave for the sequential case. Aside from using the new predicate, there are a few other differences. Firstly, we can no longer require the dictionary object to reply directly after receiving a request. Instead, requirements 2 and 3 insist that it replies eventually. Secondly, a new concurrent request may arrive at any stage, so requirement 5 insists that the dictionary object is willing to receive them.

The requirements are as follows:

1. The dictionary will only ever perform one of the actions:

$$\xrightarrow[x]{\mathtt{o.get}?\langle k\rangle}, \ \xrightarrow[x]{\mathtt{o.set}?\langle k,v\rangle}, \ \xrightarrow{x!\langle v\rangle}, \ \xrightarrow{x!\langle\rangle}$$

   where, in the first two actions, x is required to be a new name.

2. If the dictionary performs the action $\xrightarrow[x]{o.\mathtt{set}?\langle k,v\rangle}$, then it *will eventually be willing to* perform the action $\xrightarrow{x!\langle\rangle}$.

3. If the dictionary performs the action $\xrightarrow[x]{o.\texttt{get}?\langle k\rangle}$, then it *will eventually be willing to* perform the action $\xrightarrow{x!\langle v\rangle}$.

4. If the dictionary can perform trace t, then

$$\exists t'.(\text{Completed}(tt') \wedge \exists s \in \text{PossTraces}(tt').$$
$$\forall k.(\text{GetGet}_k(\pi_k(s)) \wedge \text{SetGet}_k(\pi_k(s)) \wedge \text{InitGet}_k(\pi_k(s))))$$

5. The dictionary should always be willing to perform both $\xrightarrow[x]{o.\texttt{get}?\langle k\rangle}$ and $\xrightarrow[x]{o.\texttt{set}?\langle k,v\rangle}$ actions.

An Oompa object which behaves according to all these requirements should be considered an acceptable concurrent dictionary object.

7.2 Behavioural Specification for a Concurrent Dictionary

Our next step is to derive sequential process specifications which describe the behaviour of an Oompa object implementing a concurrent dictionary.

7.2.1 Specification

We now specify processes whose behaviour obeys the requirements by providing a suitable sequential process specification. As before, we derive this incrementally, considering the requirements one by one.

Our initial specification accommodates requirement 1 by limiting the possible form of actions of our system and ensuring that new channel names are chosen. An important aspect of our current approach to specification becomes clear at this point: we must also provide for the possibility for internal

reconfigurations. The dictionary is likely to need some internal processing and this may change its future observable behaviour. We cannot know what ways the dictionary will need to configure itself, so we initially allow for an arbitrary number. So our initial specification is:

$$\begin{array}{rlll} \mathrm{Dspec}^0 \stackrel{\text{def}}{=} & \prod_{k\in\mathrm{Key},x\in\mathcal{N}} & \xrightarrow[x]{\texttt{o.get?}\langle k\rangle} & \mathrm{Dspec}^0 \\ \sqcap & \prod_{(k,v)\in\mathrm{Key}\times\mathrm{Value},x\in\mathcal{N}} & \xrightarrow[x]{\texttt{o.set?}\langle k,v\rangle} & \mathrm{Dspec}^0 \\ \sqcap & \prod_{v\in\mathrm{Value},x\in\mathcal{N}} & \xrightarrow{x!\langle v\rangle} & \mathrm{Dspec}^0 \\ \sqcap & \prod_{x\in\mathcal{N}} & \xrightarrow{x!\langle\rangle} & \mathrm{Dspec}^0 \\ \sqcap & \prod_{i\in\mathbb{N}} & \longrightarrow & \mathrm{Dspec}^0 \end{array}$$

Our second specification accommodates requirements 2 and 3 by enforcing the correct use of return channels. We need to tie requests to their replies, so the dictionary will need to carry around some kind of state. We will index states with two sets which store the channel names on which replies must be made.

$$\mathrm{R}_1 \subseteq \mathcal{N} \qquad \mathrm{R}_2 \subseteq \mathcal{N}$$

Our specification becomes:

$$\begin{array}{rlll} \mathrm{Dspec}^1_{\mathrm{R}_1,\mathrm{R}_2} \stackrel{\text{def}}{=} & \prod_{k\in\mathrm{Key},x\ \mathrm{new}} & \xrightarrow[x]{\texttt{o.get?}\langle k\rangle} & \mathrm{Dspec}^1_{\mathrm{R}_1\cup\{x\},\mathrm{R}_2} \\ \sqcap & \prod_{(k,v)\in\mathrm{Key}\times\mathrm{Value},x\ \mathrm{new}} & \xrightarrow[x]{\texttt{o.set?}\langle k,v\rangle} & \mathrm{Dspec}^1_{\mathrm{R}_1,\mathrm{R}_2\cup\{x\}} \\ \sqcap & \prod_{v\in\mathrm{Value},x\in\mathrm{R}_1} & \xrightarrow{x!\langle v\rangle} & \mathrm{Dspec}^1_{\mathrm{R}_1\setminus\{x\},\mathrm{R}_2} \\ \sqcap & \prod_{x\in\mathrm{R}_2} & \xrightarrow{x!\langle\rangle} & \mathrm{Dspec}^1_{\mathrm{R}_1,\mathrm{R}_2\setminus\{x\}} \\ \sqcap & \prod_{i\in\mathbb{N}} & \longrightarrow & \mathrm{Dspec}^1_{\mathrm{R}_1,\mathrm{R}_2} \end{array}$$

In Appendix D, we show that $\mathrm{Dspec}^0 \sqsubseteq \mathrm{Dspec}^1_{\emptyset,\emptyset}$.

Unfortunately, this specification does not actually guarantee requirements 2 and 3. Because of the outer meet, the terms for replying are still optional.

Also, even if the outer meet was strengthened to a choice, the silent actions mean that a process which repeatedly performs a silent action and no other (called a *livelocking* process) is an implementation. As we continue deriving the specification, however, these problems will be resolved and requirements 2 and 3 will then be guaranteed.

We now wish refine Dspec^1 to give a specification whose processes satisfy requirement 4, i.e. their traces satisfy the predicate:

$$\begin{aligned}&\exists t'.(\text{Completed}(tt') \wedge \exists s \in \text{PossTraces}(tt').\\&\qquad \forall k.(\text{GetGet}_k(\pi_k(s)) \wedge \text{SetGet}_k(\pi_k(s)) \wedge \text{InitGet}_k(\pi_k(s))))\end{aligned}$$

Consider how requests progress in a dictionary object. In general, it will be a three phase process:

We will need the state of the dictionary accommodate the four points in this diagram. We need to store requests which have arrived but have not yet been processed and requests which have been processed but have not yet been replied to. The most general way of storing this information will be with sets, as all information except the presence or absence of an element is abstracted away. We will use the sets

$$\begin{aligned}\text{In}_1 &\subseteq \mathcal{N} \times \text{Key}\\\text{Out}_1 &\subseteq \mathcal{N} \times \text{Value}\\\text{In}_2 &\subseteq \mathcal{N} \times \text{Key} \times \text{Value}\\\text{Out}_2 &\subseteq \mathcal{N}\end{aligned}$$

where In_1 and Out_1 are for get requests and In_2 and Out_2 are for set requests.

We now index specifications with a quadruple of state values:

$$\text{Dstate} = (\text{In}_1, \text{Out}_1, \text{In}_2, \text{Out}_1)$$

There are six ways in which this state can change: A get or set request can arrive, a reply to a get or set request can be issued and a get or set request can be processed. We use the following six functions to describe these state changes.

$$
\begin{array}{lcl}
f_1(\text{Dstate}, x, k) & = & (\text{In}_1 \cup \{(x,k)\}, \text{Out}_1, \text{In}_2, \text{Out}_2) \\
f_2(\text{Dstate}, x, k, v) & = & (\text{In}_1, \text{Out}_1, \text{In}_2 \cup \{x,k,v\}, \text{Out}_2) \\
f_3(\text{Dstate}, x, v) & = & (\text{In}_1, \text{Out}_1 \setminus \{(x,v)\}, \text{In}_2, \text{Out}_2) \\
f_4(\text{Dstate}, x) & = & (\text{In}_1, \text{Out}_1, \text{In}_2, \text{Out}_2 \setminus \{x\}) \\
f_5(\text{Dstate}, x, k, v) & = & (\text{In}_1 \setminus \{(x,k)\}, \text{Out}_1 \cup \{(x,v)\}, \text{In}_2, \text{Out}_2) \\
f_6(\text{Dstate}, x, k, v) & = & (\text{In}_1, \text{Out}_1, \text{In}_2 \setminus \{(x,k,v)\}, \text{Out}_2 \cup \{x\})
\end{array}
$$

Our new specification is:

$$
\begin{array}{llcl}
\text{Dspec}^2_{\text{Dstate}} \stackrel{\text{def}}{=} & \bigsqcap_{k \in \text{Key}, x \text{ new}} & \xrightarrow[x]{\texttt{o.get?}\langle k\rangle} & \text{Dspec}^2_{f_1(\text{Dstate},x,k)} \\
\sqcap & \bigsqcap_{(k,v) \in \text{Key} \times \text{Value}, x \text{ new}} & \xrightarrow[x]{\texttt{o.set?}\langle k,v\rangle} & \text{Dspec}^2_{f_2(\text{Dstate},x,k,v)} \\
\sqcap & \bigsqcap_{(x,v) \in \text{Out}_1} & \xrightarrow{x!\langle v\rangle} & \text{Dspec}^2_{f_3(\text{Dstate},x,v)} \\
\sqcap & \bigsqcap_{x \in \text{Out}_2} & \xrightarrow{x!\langle\rangle} & \text{Dspec}^2_{f_4(\text{Dstate},x)} \\
\sqcap & \bigsqcap_{(x,k) \in \text{In}_1, v \in \text{Value}} & \longrightarrow & \text{Dspec}^2_{f_5(\text{Dstate},x,k,v)} \\
\sqcap & \bigsqcap_{(x,k,v) \in \text{In}_2} & \longrightarrow & \text{Dspec}^2_{f_6(\text{Dstate},x,k,v)}
\end{array}
$$

In Appendix D, we show that $\text{Dspec}^1_{\emptyset,\emptyset} \sqsubseteq \text{Dspec}^2_{\text{Dstate}_0}$ where $\text{Dstate}_0 = (\emptyset, \emptyset, \emptyset, \emptyset)$.

We still haven't specified the actual processing of requests; the value returned to a get request can be any value in Value. However, our new specification is a useful intermediate stage: the traces of a process satisfying this specification do come close to satisfying the main predicate. Consider an action sequence of such a process and the order in which the silent actions representing the processing requests occur in it. A trace of that action sequence can be completed and lined up in such a way that the order of the

invocation-reply pairs in the resulting trace matches the order that the silent actions occurred.

Our next specification must ensure that the silent actions which represent the processing of the requests guarantee the predicates $\mathrm{GetGet}_k(\cdot)$, $\mathrm{SetGet}_k(\cdot)$ and $\mathrm{InitGet}_k(\cdot)$. Because we are working forward from the previous specification, we need only think of behaviour in terms of lined-up traces. The obvious thing to do is to make the dictionary's state include some kind of "dictionary logic". We will represent this by an instance of the dictionary abstract data type.

$$\mathrm{Dict} : \mathsf{Dict}$$

We now index specifications with the quintuple:

$$\mathrm{Dstate} = (\mathrm{Dict}, \mathrm{In}_1, \mathrm{Out}_1, \mathrm{In}_2, \mathrm{Out}_1)$$

The functions which act on this state value are the following:

$$\begin{aligned}
g_1(\mathrm{Dstate}, x, k) &= (\mathrm{Dict}, \mathrm{In}_1 \cup \{(x,k)\}, \mathrm{Out}_1, \mathrm{In}_2, \mathrm{Out}_2)\\
g_2(\mathrm{Dstate}, x, k, v) &= (\mathrm{Dict}, \mathrm{In}_1, \mathrm{Out}_1, \mathrm{In}_2 \cup \{x,k,v\}, \mathrm{Out}_2)\\
g_3(\mathrm{Dstate}, x, v) &= (\mathrm{Dict}, \mathrm{In}_1, \mathrm{Out}_1 \setminus \{(x,v)\}, \mathrm{In}_2, \mathrm{Out}_2)\\
g_4(\mathrm{Dstate}, x) &= (\mathrm{Dict}, \mathrm{In}_1, \mathrm{Out}_1, \mathrm{In}_2, \mathrm{Out}_2 \setminus \{x\})\\
g_5(\mathrm{Dstate}, x, k, v) &= (\mathrm{Dict}, \mathrm{In}_1 \setminus \{(x,k)\}, \mathrm{Out}_1 \cup \{(x,v)\}, \mathrm{In}_2, \mathrm{Out}_2)\\
g_6(\mathrm{Dstate}, x, k, v) &= \\
&\quad (\mathsf{set}(\mathrm{Dict}, k, v), \mathrm{In}_1, \mathrm{Out}_1, \mathrm{In}_2 \setminus \{(x,k,v)\}, \mathrm{Out}_2 \cup \{x\})
\end{aligned}$$

and our specification becomes:

$$
\begin{array}{llll}
\mathrm{Dspec}^3_{\mathrm{Dstate}} \stackrel{\mathrm{def}}{=} & \prod_{k\in\mathrm{Key},x\,\mathrm{new}} & \xrightarrow[x]{\mathtt{o.get?}\langle k\rangle} & \mathrm{Dspec}^3_{g_1(\mathrm{Dstate},x,k)} \\
\sqcap & \prod_{(k,v)\in\mathrm{Key}\times\mathrm{Value},x\,\mathrm{new}} & \xrightarrow[x]{\mathtt{o.set?}\langle k,v\rangle} & \mathrm{Dspec}^3_{g_2(\mathrm{Dstate},x,k,v)} \\
\sqcap & \prod_{(x,v)\in\mathrm{Out}_1} & \xrightarrow{x!\langle v\rangle} & \mathrm{Dspec}^3_{g_3(\mathrm{Dstate},x,v)} \\
\sqcap & \prod_{x\in\mathrm{Out}_2} & \xrightarrow{x!\langle\rangle} & \mathrm{Dspec}^3_{g_4(\mathrm{Dstate},x)} \\
\sqcap & \prod_{(x,k)\in\mathrm{In}_1,v=\mathsf{get}(\mathrm{Dict},k)} & \longrightarrow & \mathrm{Dspec}^3_{g_5(\mathrm{Dstate},x,k,v)} \\
\sqcap & \prod_{(x,k,v)\in\mathrm{In}_2} & \longrightarrow & \mathrm{Dspec}^3_{g_6(\mathrm{Dstate},x,k,v)}
\end{array}
$$

Let $\mathrm{Dict}_\perp$ associate $\perp$ with every key and let $\mathrm{Dstate}_\perp = (\mathrm{Dict}_\perp, \emptyset, \emptyset, \emptyset, \emptyset)$. Clearly, the $\mathrm{InitGet}_k(\cdot)$ predicate is realised by making the initial dictionary value be $\mathrm{Dict}_\perp$. It is quite easy to show that $\mathrm{Dspec}^2_{\mathrm{Dstate}_0} \sqsubseteq \mathrm{Dspec}^3_{\mathrm{Dstate}_\perp}$ and we only briefly discuss it in Appendix D.

To accommodate requirement 5, we need to restructure the specification. The receipt of requests need to be strengthened so they are always available for the environment. The kind of strengthening we do here is a minimum strengthening (similar to that mentioned on page 176), so we leave the other terms optional by adding a 0.

$$
\begin{array}{llll}
\mathrm{Dspec}^4_{\mathrm{Dstate}} \stackrel{\mathrm{def}}{=} & \square_{k\in\mathrm{Key},x\,\mathrm{new}} & \xrightarrow[x]{\mathtt{o.get?}\langle k\rangle} & \mathrm{Dspec}^4_{g_1(\mathrm{Dstate},x,k)} \\
\square & \square_{k\in\mathrm{Key},v\in\mathrm{Value},x\,\mathrm{new}} & \xrightarrow[x]{\mathtt{o.set?}\langle k,v\rangle} & \mathrm{Dspec}^4_{g_2(\mathrm{Dstate},x,k,v)} \\
\square & (\ \prod_{(x,v)\in\mathrm{Out}_1} & \xrightarrow{x!\langle v\rangle} & \mathrm{Dspec}^4_{g_3(\mathrm{Dstate},x,v)} \\
& \sqcap \prod_{x\in\mathrm{Out}_2} & \xrightarrow{x!\langle\rangle} & \mathrm{Dspec}^4_{g_4(\mathrm{Dstate},x)} \\
& \sqcap \prod_{(x,k)\in\mathrm{In}_1,v=\mathsf{get}(\mathrm{Dict},k)} & \longrightarrow & \mathrm{Dspec}^4_{g_5(\mathrm{Dstate},x,k,v)} \\
& \sqcap \prod_{(x,k,v)\in\mathrm{In}_2} & \longrightarrow & \mathrm{Dspec}^4_{g_6(\mathrm{Dstate},x,k,v)} \\
& \sqcap\ 0\ ) & &
\end{array}
$$

One reason why the 0 is necessary is that, when the four sets In_1, Out_1,

In_2 and Out_2 are all empty, the specification would become a choice one of whose arguments was an empty meet. Since $\top$ is a zero of $\Box$, the specification would be unimplementable. In Appendix D, we show that $\text{Dspec}^4_{\text{Dstate}_\perp} \sqsubseteq \text{Dspec}^5_{\text{Dstate}_\perp}$.

Our final specification manages the 0 and splits the specification into two cases. Firstly, if the set In_1, In_2, Out_1, and Out_2 are all empty, then the only thing the dictionary can do is accept requests:

$$\begin{array}{lcll} \text{Dspec}^5_{\text{Dstate}} & \stackrel{\text{def}}{=} & \Box_{k \in \text{Key}, x\ \text{new}} & \xrightarrow[x]{\text{o.}\mathtt{get}?\langle k \rangle} \text{Dspec}^5_{g_1(\text{Dstate},x,k)} \\ & \Box & \Box_{k \in \text{Key}, v \in \text{Value}, x\ \text{new}} & \xrightarrow[x]{\text{o.}\mathtt{set}?\langle k,v \rangle} \text{Dspec}^5_{g_2(\text{Dstate},x,k,v)} \end{array}$$

However, when at least one of the sets In_1, In_2, Out_1 and Out_2 is non-empty, we should be willing to do something other than just wait for new requests. The most general approach to this is simply to leave the meet of all those terms and remove the 0. This ensures that at least one of the available actions can be performed.

$$\begin{array}{lcll} \text{Dspec}^5_{\text{Dstate}} & \stackrel{\text{def}}{=} & \Box_{k \in \text{Key}, x\ \text{new}} & \xrightarrow[x]{\text{o.}\mathtt{get}?\langle k \rangle} \text{Dspec}^5_{g_1(\text{Dstate},x,k)} \\ & \Box & \Box_{k \in \text{Key}, v \in \text{Value}, x\ \text{new}} & \xrightarrow[x]{\text{o.}\mathtt{set}?\langle k,v \rangle} \text{Dspec}^5_{g_2(\text{Dstate},x,k,v)} \\ & \Box & (\prod_{(x,v) \in \text{Out}_1} & \xrightarrow{x!\langle v \rangle} \text{Dspec}^5_{g_3(\text{Dstate},x,v)} \\ & & \sqcap \prod_{x \in \text{Out}_2} & \xrightarrow{x!\langle\rangle} \text{Dspec}^5_{g_4(\text{Dstate},x)} \\ & & \sqcap \prod_{(x,k) \in \text{In}_1, v = \mathtt{get}(\text{Dict},k)} & \longrightarrow \text{Dspec}^5_{g_5(\text{Dstate},x,k,v)} \\ & & \sqcap \prod_{(x,k,v) \in \text{In}_2} & \longrightarrow \text{Dspec}^5_{g_6(\text{Dstate},x,k,v)}) \end{array}$$

It is quite easy to show that $\text{Dspec}^4_{\text{Dstate}_\perp} \sqsubseteq \text{Dspec}^5_{\text{Dstate}_\perp}$ and we only briefly discuss this in Appendix D.

Many different dictionary behaviours fit under this specification. For example, by relevant strengthenings, we could insist that if the dictionary

has a reply ready, then it must allow the environment to choose that action. By further strengthenings, we could make the dictionary process a value and reply before any other values are processed.

Regrettably, a deficiency of our notion of refinement means that our specification does not cover all acceptable dictionaries. The problem is that we have not given silent actions any special treatment — they are considered as normal actions. This makes our specification a little too strict, since a refining process cannot perform new (but insignificant) internal actions. In the next section we present an alternative specification whose behaviours also satisfy the requirements. Although both the specification above and the alternative can be expressed in sequential process specifications, neither is a refinement of the other and there is no common ancestor of which both are refinements. We discuss this in more detail in Section 7.4.

7.2.2 Alternative Specification

In this section we give an alternative specification for concurrent dictionary objects. The difference in the specification is that the processing of set requests is broken into two silent actions. Most significantly, we can allow get requests to be processed while a set request is underway. The derivation of the specification is very similar to the previous case, so we omit most of the details.

We use the specifications Dspec^0 and $\mathrm{Dspec}^1_{\mathrm{R}_1,\mathrm{R}_2}$ as before. We do need a new version of Dspec^2 to reflects the extra silent actions we are assigning to the processing of set requests. However, it is the fourth specification that reflects the most interesting change: the state carried by the specifications need to record the details of a set request which is underway. To the state values carried by Dspec^3, we add a new state component to accommodate

this information. As there can only be one pending set request, this is a set which either contains a single triple or is empty[2]:

$$\text{Pending} \in \mathcal{N} \times \text{Key} \times \text{Value}$$

We index specifications with the following state value:

$$\text{Dstate} = (\text{Dict}, \text{In}_1, \text{Out}_1, \text{In}_2, \text{Pending}, \text{Out}_2)$$

and define the following functions on those state values:

$$\begin{array}{lcl}
h_1(\text{Dstate}, x, k) & = & (\text{Dict}, \text{In}_1 \cup \{(x,k)\}, \text{Out}_1, \text{In}_2, \text{Pending}, \text{Out}_2) \\
h_2(\text{Dstate}, x, k, v) & = & (\text{Dict}, \text{In}_1, \text{Out}_1, \text{In}_2 \cup \{(x,k,v)\}, \text{Pending}, \text{Out}_2) \\
h_3(\text{Dstate}, x, v) & = & (\text{Dict}, \text{In}_1, \text{Out}_1 \setminus \{(x,v)\}, \text{In}_2, \text{Pending}, \text{Out}_2) \\
h_4(\text{Dstate}, x) & = & (\text{Dict}, \text{In}_1, \text{Out}_1, \text{In}_2, \text{Pending}, \text{Out}_2 \setminus \{x\}) \\
h_5(\text{Dstate}, x, k, v) & = & \\
\multicolumn{3}{c}{(\text{Dict}, \text{In}_1 \setminus \{(x,k)\}, \text{Out}_1 \cup \{x,v\}, \text{In}_2, \text{Pending}, \text{Out}_2)} \\
h_6(\text{Dstate}, x, k, v) & = & \\
\multicolumn{3}{c}{(\text{Dict}, \text{In}_1, \text{Out}_1, \text{In}_2 \setminus \{(x,k,v)\}, \text{Pending} \cup \{(x,k,v)\}, \text{Out}_2)} \\
h_7(\text{Dstate}, x, k, v) & = & \\
\multicolumn{3}{c}{(\mathsf{set}(\text{Dict}, k, v), \text{In}_1, \text{Out}_1, \text{In}_2, \text{Pending} \setminus \{(x,k,v)\}, \text{Out}_2 \cup \{x\})}
\end{array}$$

[2]Alternatively, we could have used a sum type, e.g. $\text{Pending} \in \mathcal{N} \times \text{Key} \times \text{Value} + \text{Unit}$, corresponding to a *maybe value* as used, for example, in Haskell [Jon03]

Our new fourth specification will be:

$$
\begin{aligned}
\mathrm{Dspec}^{3'}_{\mathrm{Dstate}} \stackrel{\mathrm{def}}{=} & \quad \bigsqcap_{k\in\mathrm{Key},x\,\mathrm{new}} \xrightarrow[x]{\mathrm{o.}\mathtt{get}?\langle k\rangle} \mathrm{Dspec}^{3'}_{h_1(\mathrm{Dstate},x,k)} \\
\sqcap & \quad \bigsqcap_{(k,v)\in\mathrm{Key}\times\mathrm{Value},x\,\mathrm{new}} \xrightarrow[x]{\mathrm{o.}\mathtt{set}?\langle k,v\rangle} \mathrm{Dspec}^{3'}_{h_2(\mathrm{Dstate},x,k,v)} \\
\sqcap & \quad \bigsqcap_{(x,v)\in\mathrm{Out}_1} \xrightarrow{x!\langle v\rangle} \mathrm{Dspec}^{3'}_{h_3(\mathrm{Dstate},x,v)} \\
\sqcap & \quad \bigsqcap_{x\in\mathrm{Out}_2} \xrightarrow{x!\langle\rangle} \mathrm{Dspec}^{3'}_{h_4(\mathrm{Dstate},x)} \\
\sqcap & \quad \bigsqcap_{(x,k)\in\mathrm{In}_1,v=\mathtt{get}(\mathrm{Dict},k)} \longrightarrow \mathrm{Dspec}^{3'}_{h_5(\mathrm{Dstate},x,k,v)} \\
\sqcap & \quad \bigsqcap_{(x,k,v)\in\mathrm{In}_2,\mathrm{Pending}=\emptyset} \longrightarrow \mathrm{Dspec}^{3'}_{h_6(\mathrm{Dstate},x,k,v)} \\
\sqcap & \quad \bigsqcap_{(x,k,v)\in\mathrm{Pending}} \longrightarrow \mathrm{Dspec}^{3'}_{h_7(\mathrm{Dstate},x,k,v)}
\end{aligned}
$$

Following a sequence of specifications similar to in that in Section 7.2, we derive our final specification. If the sets In_1, Out_1, In_2, Pending, Out_2 are all empty, then

$$
\begin{aligned}
\mathrm{Dspec}^{5'}_{\mathrm{Dstate}} \stackrel{\mathrm{def}}{=} & \quad \Box_{k\in\mathrm{Key},x\,\mathrm{new}} \xrightarrow[x]{\mathrm{o.}\mathtt{get}?\langle k\rangle} \mathrm{Dspec}^{5'}_{h_1(\mathrm{Dstate},x,k)} \\
\Box & \quad \Box_{(k,v)\in\mathrm{Key}\times\mathrm{Value},x\,\mathrm{new}} \xrightarrow[x]{\mathrm{o.}\mathtt{set}?\langle k,v\rangle} \mathrm{Dspec}^{5'}_{h_2(\mathrm{Dstate},x,k,v)}
\end{aligned}
$$

Otherwise,

$$
\begin{aligned}
\mathrm{Dspec}^{5'}_{\mathrm{Dstate}} \stackrel{\mathrm{def}}{=} & \quad \Box_{k\in\mathrm{Key},x\,\mathrm{new}} \xrightarrow[x]{\mathrm{o.}\mathtt{get}?\langle k\rangle} \mathrm{Dspec}^{5'}_{h_1(\mathrm{Dstate},x,k)} \\
\Box & \quad \Box_{(k,v)\in\mathrm{Key}\times\mathrm{Value},x\,\mathrm{new}} \xrightarrow[x]{\mathrm{o.}\mathtt{set}?\langle k,v\rangle} \mathrm{Dspec}^{5'}_{h_2(\mathrm{Dstate},x,k,v)} \\
\Box & \quad (\bigsqcap_{(x,v)\in\mathrm{Out}_1} \xrightarrow{x!\langle v\rangle} \mathrm{Dspec}^{5'}_{h_3(\mathrm{Dstate},x,v)} \\
& \quad \sqcap \bigsqcap_{x\in\mathrm{Out}_2} \xrightarrow{x!\langle\rangle} \mathrm{Dspec}^{5'}_{h_4(\mathrm{Dstate},x)} \\
& \quad \sqcap \bigsqcap_{(x,k)\in\mathrm{In}_1,v=\mathtt{get}(\mathrm{Dict},k)} \longrightarrow \mathrm{Dspec}^{5'}_{h_5(\mathrm{Dstate},x,k,v)} \\
& \quad \sqcap \bigsqcap_{(x,k,v)\in\mathrm{In}_2,\mathrm{Pending}=\emptyset} \longrightarrow \mathrm{Dspec}^{5'}_{h_6(\mathrm{Dstate},x,k,v)} \\
& \quad \sqcap \bigsqcap_{(x,k,v)\in\mathrm{Pending}} \longrightarrow \mathrm{Dspec}^{5'}_{h_7(\mathrm{Dstate},x,k,v)})
\end{aligned}
$$

7.3 Designs for a Concurrent Dictionary

In this section, we consider how Oompa objects might satisfy the specification for the dictionary. Our method is not sufficiently developed to support a formal refinement that will take us from sequential process specifications to Oompa programs. However, we can use our theory to "invent and verify" such programs.

We will provide two implementations of a concurrent dictionary, one for each of the specifications. In fact, we will use an extended Oompa which supports a few features we have chosen not to formalise. We believe that a version in pure Oompa would obscure the designs we present.

1. We will allow the abstract data type Dict defined in Section 7.1.1 to be used as an Oompa type.

2. We will allow the expressions representing operations on the abstract data type, $\mathsf{get}(d, k)$ and $\mathsf{set}(d, k, v)$, to be used in value positions. We will imagine that they get reduced to the corresponding result exactly when all their parameters are replaced by values.

3. We will use an informal approach to locking. We will provide primitives "`acquire` *lockname*" and "`release` *lockname*", both of which take a single silent action when performed.

Both designs are quite simple, as we intend them to illustrate the relationship between Oompa programs and sequential process specifications. In Section 7.3.5, we consider how we might provide more sophisticated designs.

7.3.1 A Standard Design

We first provide a very standard design for the dictionary. Our design is a class with a pure functional dictionary attribute, access to which is locked in both get and set requests. The Oompa class which embodies our design is:

```
class Dictionary_Design_1
{
  dict:Attr{Dict}

  get?(k:Key)!<v:Value>
  {
    acquire lock
    dict?d
    release lock
    return!<get(d,k)>
    end
  }

  set?(k:Key,v:Value)!<>
  {
    acquire lock
    return!<>
    dict?d
    dict!set(d,k,v)
    release lock
    end
  }
}
```

A notable feature of this design is that the set method replies before the actual request has been processed. This is a technique called an *early return* [Jon92]. In Oompa, when a client invokes a method, it blocks until it receives a reply; an early return means that the client is released as soon as possible and can, therefore, get more work done. This technique has the disadvantage that there is more code in the locked region and, hence, less parallelism in the dictionary object. Of course, a design where the return was put after the locked region would also satisfy the specification.

7.3.2 Justifying the Standard Design

Our approach to justifying that objects of this class actually implement the dictionary specification is as follows. First, we consider the possible configurations which represent the states of such an object. Then, we present sequential process expressions which are weakly bisimilar to the configurations but also formally satisfy the specification. Thus, we know the configurations and the specification lie in the implementation relation $\approx\tilde{\mathsf{sat}}$.

Consider a configuration which represents an instance of a dictionary object, o. An obvious observation is that all of its agents must be code from either a get or a set method. Thus, its agents fit into the following

classification:

$$
\begin{array}{ll}
\text{gets} & \left\{ \begin{array}{lcll}
K_i^1 & = & \mathrm{o}[\texttt{acquire}\ \mathit{lock} \ldots] & i \in 1..n_1 \\
K_i^2 & = & \mathrm{o}[\texttt{dict}?d \ldots] & i \in 1..n_2 \\
K_i^3 & = & \mathrm{o}[\texttt{release}\ \mathit{lock} \ldots] & i \in 1..n_3 \\
K_i^4 & = & \mathrm{o}[x_i^4!\langle v_i^4 \rangle \ldots] & i \in 1..n_4 \\
K_i^5 & = & \mathrm{o}[\texttt{end}] & i \in 1..n_5
\end{array} \right. \\
\text{sets} & \left\{ \begin{array}{lcll}
K_i^6 & = & \mathrm{o}[\texttt{acquire}\ \mathit{lock} \ldots] & i \in 1..n_6 \\
K_i^7 & = & \mathrm{o}[x_i^7!\langle\rangle \ldots] & i \in 1..n_7 \\
K_i^8 & = & \mathrm{o}[\texttt{dict}?d \ldots] & i \in 1..n_8 \\
K_i^9 & = & \mathrm{o}[\texttt{dict}!D_i^9 \ldots] & i \in 1..n_9 \\
K_i^{10} & = & \mathrm{o}[\texttt{release}\ \mathit{lock} \ldots] & i \in 1..n_{10} \\
K_i^{11} & = & \mathrm{o}[\texttt{end}] & i \in 1..n_{11}
\end{array} \right.
\end{array}
$$

We can refine this, as we know that at most one agent can be holding the lock on the dictionary at any one time. Thus, we know that there is only one of the agents K_i^2, K_i^3, K_i^7, K_i^8, K_i^9 and K_i^{10}. Moreover, there is no point in distinguishing a K_i^5 agent from a K_i^{11} agent. Using $\overset{\blacktriangleright}{\equiv}$, we can reorganise a configuration of the dictionary object into the following form:

$$
\text{Kstate} = \quad K^1 | K^{\text{get}} | K^4 | K^6 | K^{\text{set}} | K^{\text{end}} \quad \left\{ \begin{array}{l} \emptyset \\ \mathrm{o} := [\texttt{dict} \mapsto D] \end{array} \right.
$$

where the agents in K^1 are newly invoked get methods:

$$
K^1 = \begin{array}{c} \mathrm{o}[\texttt{acquire}\ \mathit{lock}\ \texttt{dict}?d\ \texttt{release}\ \mathit{lock}\ x_1^1!\langle \texttt{get}(d, k_1^1) \rangle\ \texttt{end}] \mid \ldots \\ \mathrm{o}[\texttt{acquire}\ \mathit{lock}\ \texttt{dict}?d\ \texttt{release}\ \mathit{lock}\ x_{n_1}^1!\langle \texttt{get}(d, k_{n_1}^1) \rangle\ \texttt{end}] \end{array}
$$

The agents in K^4 are get methods which have performed their access:

$$
K^4 = \mathrm{o}[x_1^4!\langle v_1^4 \rangle\ \texttt{end}] \mid \ \ldots\ \mathrm{o}[x_{n_4}^4!\langle v_{n_4}^4 \rangle\ \texttt{end}]
$$

The agents in K^6 are newly invoked set methods:

$$K^6 = \begin{array}{l} \mathrm{o}[\texttt{acquire}\ \mathit{lock}\ x_1^6!\langle\rangle\ \texttt{dict}?d\ \texttt{dict!set}(d, k_1^6, v_1^6)\ \texttt{release}\ \mathit{lock}\ \texttt{end}]\ | \\ \vdots \\ \mathrm{o}[\texttt{acquire}\ \mathit{lock}\ x_{n6}^6!\langle\rangle\ \texttt{dict}?d\ \texttt{dict!set}(d, k_{n6}^6, v_{n6}^6)\ \texttt{release}\ \mathit{lock}\ \texttt{end}] \end{array}$$

K^{get} is either Nil or contains a single agent of one of the forms:

$$\begin{array}{ll} \text{(i)} & \mathrm{o}[\texttt{dict}?d\ \texttt{release}\ \mathit{lock}\ x!\langle\mathsf{get}(d,k)\rangle\ \texttt{end}] \\ \text{(ii)} & \mathrm{o}[\texttt{release}\ \mathit{lock}\ x!\langle v\rangle\ \texttt{end}] \end{array}$$

The agent in K^{set} is either Nil or has one of the forms:

$$\begin{array}{ll} \text{(i)} & \mathrm{o}[x!\langle\rangle\ \texttt{dict}?d\ \texttt{dict!set}(d,k,v)\ \texttt{release}\ \mathit{lock}\ \texttt{end}] \\ \text{(ii)} & \mathrm{o}[\texttt{dict}?d\ \texttt{dict!set}(d,k,v)\ \texttt{release}\ \mathit{lock}\ \texttt{end}] \\ \text{(iii)} & \mathrm{o}[\texttt{dict}!D'\ \texttt{release}\ \mathit{lock}\ \texttt{end}] \\ \text{(iv)} & \mathrm{o}[\texttt{release}\ \mathit{lock}\ \texttt{end}] \end{array}$$

The agents in K^{end} are get or set methods which have finished their work.

$$K^{\text{end}} = \mathrm{o}[\texttt{end}]\ |\ \ldots\ \mathrm{o}[\texttt{end}]$$

We will show that our configuration Kstate is weakly bisimilar to the following sequential process expression:

$$\begin{array}{lllll} p_{\text{Dstate}} & \stackrel{\text{def}}{=} & \bigstar_{k\in\text{Key},x\text{ new}} & \xrightarrow[x]{\mathrm{o}.\mathtt{get}?\langle k\rangle} & p_{g1(\text{Dstate},x,k)} \\ & \star & \bigstar_{k\in\text{Key},v\in\text{Value},x\text{ new}} & \xrightarrow[x]{\mathrm{o}.\mathtt{set}?\langle k,v\rangle} & p_{g2(\text{Dstate},x,k,v)} \\ & \star & \bigstar_{(x,v)\in\text{Out}_1} & \xrightarrow{x!\langle v\rangle} & p_{g3(\text{Dstate},x,v)} \\ & \star & \bigstar_{x\in\text{Out}_2} & \xrightarrow{x!\langle\rangle} & p_{g4(\text{Dstate},x)} \\ & \star & \bigstar_{(x,k)\in\text{In}_1,v=\mathsf{get}(\text{Dict},k),\text{Out}_2=\emptyset} & \longrightarrow & p_{g5(\text{Dstate},x,k,v)} \\ & \star & \bigstar_{(x,k,v)\in\text{In}_2,\text{Out}_2=\emptyset} & \longrightarrow & p_{g6(\text{Dstate},x,k,v)} \end{array}$$

It is not difficult to show that $p_{\text{Dstate}_\perp}$ $\tilde{\mathsf{sat}}$ $\text{Dspec}^5_{\text{Dstate}_\perp}$. To show that $\text{Kstate}_\perp \approx p_{\text{Dstate}_\perp}$ we need to construct a bisimulation. To do so, we will define

a projection from the possible configurations of the object into the state components of the sequential process expression above.

Given a configuration, Kstate, we define the following:

$$
\begin{aligned}
\pi_1(\text{Kstate}) &= \begin{cases} \mathsf{set}(D,k,v) & \text{if } K^{\text{set}} \text{ is in state (i) or (ii)} \\ D' & \text{if } K^{\text{set}} \text{ is in state (iii)} \\ D & \text{otherwise} \end{cases} \\
\pi_2(\text{Kstate}) &= \{(x_i^1, k_i^1) \mid i \in 1..n_1\} \\
\pi_3(\text{Kstate}) &= \{(x_i^4, v_i^4) \mid i \in 1..n_4\} \\
& \quad \cup \begin{cases} \{(x, \mathsf{get}(D,k))\} & \text{if } K^{\text{get}} \text{ is in state (i)} \\ \{(x,v)\} & \text{if } K^{\text{get}} \text{ is in state (ii)} \\ \emptyset & \text{otherwise} \end{cases} \\
\pi_4(\text{Kstate}) &= \{(x_i^6, k_i^6, v_i^6) \mid i \in 1..n_6\} \\
\pi_5(\text{Kstate}) &= \begin{cases} \{x\} & \text{if } K^{\text{set}} \text{ is in state (i)} \\ \emptyset & \text{otherwise} \end{cases}
\end{aligned}
$$

We then define π as follows:

$$\pi(\text{Kstate}) = (\pi_1(\text{Kstate}), \pi_2(\text{Kstate}), \pi_3(\text{Kstate}), \pi_4(\text{Kstate}), \pi_5(\text{Kstate}))$$

We now show the following is a weak bisimulation between configurations Kstate and $p_{\pi(\text{Kstate})}$:

$$\mathcal{S} = \{(\text{Kstate}, p_{\pi(\text{Kstate})}) \mid \forall \text{Kstate}\}$$

The definition of weak bisimulation is on page 114. To show that $\mathcal{S}$ is a weak bisimulation, we examine each pair, $(\text{Kstate}, p_{\pi(\text{Kstate})})$, and show that

- For each action that Kstate can perform to reach Kstate$'$, $p_{\pi(\text{Kstate})}$ can perform an experiment with the same visible action to reach $p_{\pi(\text{Kstate}')}$.
- For each action that $p_{\pi(\text{Kstate})}$ can perform to reach p', Kstate can perform an experiment with the same visible action to reach some Kstate$'$ where $p' = p_{\pi(\text{Kstate}')}$.

We reason somewhat informally but everything considered here can be fully formalised.

First we consider the actions of Kstate:

- $\xrightarrow[x]{\texttt{o.get?}\langle k\rangle}$: a new get request arrives and a new K^1 agent is created. This is simulated by $p_{\pi(\text{Kstate})}$ performing $\xrightarrow[x]{\texttt{o.get?}\langle k\rangle}$.

- $\xrightarrow[x]{\texttt{o.set?}\langle k,v\rangle}$: a new set request arrives and a new K^6 agent is created. This is simulated by $p_{\pi(\text{Kstate})}$ performing $\xrightarrow[x]{\texttt{o.set?}\langle k,v\rangle}$.

- $\xrightarrow{x!\langle v\rangle}$: a reply to a get request is issued and a K^4 agent is converted into a K^{end} agent. This is simulated by $p_{\pi(\text{Kstate})}$ performing $\xrightarrow{x!\langle v\rangle}$.

- $\xrightarrow{x!\langle\rangle}$: a reply to a set request is issued. There must have been a K^{set} agent in state (i) which is changed to state (ii). So $\pi_5(\text{Kstate}) = \{x\}$ and the action can be simulated by $p_{\pi(\text{Kstate})}$ performing $\xrightarrow{x!\langle\rangle}$.

- $\longrightarrow$: a K^1 agent acquires the lock and becomes a K^{get} agent in state (i). This can occur only when there are no other K^{get} or K^{set} agents. Let (x, k) be the values in that K^1 agent and let $v = \mathsf{get}(\pi_1(\text{Dstate}), k)$. The action is simulated by $p_{\pi(\text{Kstate})}$ performing a silent action and becoming state $p_{g_5(\pi(\text{Kstate}),x,k,v)}$. It can perform this action because $(x, k) \in \pi_2(\text{Kstate})$ and $\pi_5(\text{Kstate}) = \emptyset$.

- $\longrightarrow$: a K^{get} agent in state (i) becomes an K^{get} agent in state (ii). This is simulated by $p_{\pi(\text{Kstate})}$ doing nothing.

- $\longrightarrow$: a K^{get} agent in state (ii) becomes an K^4 agent. This is simulated by $p_{\pi(\text{Kstate})}$ doing nothing.

- $\longrightarrow$: a K^6 agent acquires the lock and becomes a K^{set} agent in state (i). This can occur only when there are no other K^{get} or K^{set} agents.

Let (x, k, v) be the values in that K^6 agent. The action is simulated by $p_{\pi(\text{Kstate})}$ performing a silent action and becoming state $p_{g_6(\pi(\text{Kstate}),x,k,v)}$. It can perform this action because $(x, k, v) \in \pi_4(\text{Kstate})$ and $\pi_5(\text{Kstate}) = \emptyset$.

- $\longrightarrow$: a K^{set} agent in state (ii) or (iii) becomes a K^{set} agent in state (iii) or (iv) respectively. This is simulated by $p_{\pi(\text{Kstate})}$ doing nothing.

- $\longrightarrow$: a K^{set} agent in state (iv) becomes a K^{end} agent. This is simulated by $p_{\pi(\text{Kstate})}$ doing nothing.

- $\longrightarrow$: a K^{end} agent is removed. This is simulated by $p_{\pi(\text{Kstate})}$ doing nothing.

Now we must reason that Kstate can simulate any of the actions performed by $p_{\pi(\text{Kstate})}$.

- $\xrightarrow[x]{\texttt{o.get}?\langle k\rangle}$ leading to state $p_{g_1(\pi(\text{Kstate}),x,k)}$. This is simulated by Kstate performing $\xrightarrow[x]{\texttt{o.get}?\langle k\rangle}$, whereby a new K^1 agent is created.

- $\xrightarrow[x]{\texttt{o.set}?\langle k,v\rangle}$ leading to state $p_{g_2(\pi(\text{Kstate}),x,k,v)}$. This is simulated by Kstate performing $\xrightarrow[x]{\texttt{o.set}?\langle k,v\rangle}$, whereby a new K^6 agent is created.

- $\xrightarrow{x!\langle v\rangle}$ leading to state $p_{g_3(\pi(\text{Kstate}),x,v)}$. If the pair (x, v) is in $\pi_3(\text{Kstate})$ by virtue of belonging to a K^{get} agent, then Kstate may need to perform one or two silent actions to process this get request. After this, or if the values already belonged to a K^4 agent, Kstate can now perform the action $\xrightarrow{x!\langle v\rangle}$, converting the K^4 agent into a K^{end} agent.

- $\xrightarrow{x!\langle\rangle}$ leading to state $p_{g_4(\pi(\text{Kstate}),x)}$. So $\pi_5(\text{Kstate}) = \{x\}$ and there is a K^{set} agent in state (i) containing the corresponding x. Therefore

Kstate can perform the action $\xrightarrow{x!\langle\rangle}$, converting this K^{set} agent from state (i) to state (ii).

- $\longrightarrow$ leading to state $p_{g_5(\pi(\text{Kstate}),x,k,v)}$ where $v = \mathsf{get}(\pi_1(\text{Kstate}), k)$. For this to occur, $\pi_5(\text{Kstate}) = \emptyset$. If there is any current K^{get} or K^{set} agent (the latter must be in a state subsequent to (i)), then Kstate can perform a number of silent actions to remove that agent and thereby release the lock. Since $(x, k) \in \pi_2(\text{Kstate})$, we know there is a K^1 agent with the appropriate values. Therefore, Kstate can perform a silent action and convert that K^1 agent into a K^{get} agent in state (i).

- $\longrightarrow$ leading to state $p_{g_6(\pi(\text{Kstate}),x,k,v)}$. For this to occur, $\pi_5(\text{Kstate}) = \emptyset$. If there is any current K^{get} or K^{set} agent (the latter must be in a state subsequent to (i)), then Kstate can perform a number of silent actions to remove that agent and thereby release the lock. Since $(x, k, v) \in \pi_4(\text{Kstate})$, we know there is a K^6 agent with the appropriate values. Therefore, Kstate can perform a silent action and convert that K^6 agent into a K^{set} agent in state (i).

Thus, $\mathcal{S}$ is a weak bisimulation. This means that the implementation relation holds between the configurations of the objects of the class we presented and the specifications $\text{Dspec}^5_{\text{Dstate}}$.

7.3.3 An Alternative Design

A design which implements the second specification emphasises the internal parallelism of Oompa objects. We choose not to lock access to the dictionary in get requests at all, although this prohibits the use of early returns[3].

[3] An implementation where get requests could occur in parallel with a set request which has not been processed, but has already replied, has a clearly different behaviour than any

Nevertheless, by moving the return outside the locked region, we gain more parallelism in the dictionary object. This design is suitable for situations where the frequency of get requests exceeds the frequency of set requests (which seems plausible). Our alternate design is as follows.

```
class Dictionary_Design_2
{
  dict:Attr{KeyValueDict}

  get?(k:Key)!<v:Value>
  {
    dict?d
    return!<get(d,k)>
    end
  }

  set?(k:Key,v:Value)!<>
  {
    acquire lock
    dict?d
    dict!set(d,k,v)
    release lock
    return!<>
    end
  }
}
```

allowed by our requirements.

7.3.4 Justifying the Alternative Design

Consider a configuration which represents an instance of a dictionary object, o. Its agents fit into the following classification:

$$
\begin{array}{l}
\text{gets}\left\{\begin{array}{lcll}
K_i^1 & = & \text{o}[\texttt{dict}?d\ldots] & i \in 1..n_1 \\
K_i^2 & = & \text{o}[x_i^2!\langle v_i^2\rangle\ldots] & i \in 1..n_2 \\
K_i^3 & = & \text{o}[\texttt{end}] & i \in 1..n_3
\end{array}\right. \\
\text{sets}\left\{\begin{array}{lcll}
K_i^4 & = & \text{o}[\texttt{acquire}\ \mathit{lock}\ldots] & i \in 1..n_4 \\
K_i^5 & = & \text{o}[\texttt{dict}?d\ldots] & i \in 1..n_5 \\
K_i^6 & = & \text{o}[\texttt{dict}!D_i^6\ldots] & i \in 1..n_6 \\
K_i^7 & = & \text{o}[\texttt{release}\ \mathit{lock}\ldots] & i \in 1..n_7 \\
K_i^8 & = & \text{o}[x_i^8!\langle\rangle\ldots] & i \in 1..n_8 \\
K_i^9 & = & \text{o}[\texttt{end}] & i \in 1..n_9
\end{array}\right.
\end{array}
$$

We can refine this, as we know that at most one agent can be holding the lock on the dictionary at any one time. Thus, we know that there is only one of the agents K_i^5, K_i^6, and K_i^7. Moreover, there is no point in distinguishing a K_i^3 agent from a K_i^9 agent. Using $\overset{\blacktriangleright}{\equiv}$, we can reorganise a configuration of the dictionary object into the following form:

$$
\text{Kstate} = \quad K^1|K^2|K^4|K^{\text{set}}|K^8|K^{\text{end}} \quad \left\{\begin{array}{l} \emptyset \\ \text{o} := [\texttt{dict} \mapsto D] \end{array}\right.
$$

where the agents in K^1 are newly invoked get methods:

$$
K^1 = \text{o}[\texttt{dict}?d\ x_1^1!\langle\texttt{get}(d, k_1^1)\rangle\ \texttt{end}] \mid \ \ldots\ \text{o}[\texttt{dict}?d\ x_{n_1}^1!\langle\texttt{get}(d, k_{n_1}^1)\rangle\ \texttt{end}]
$$

The agents in K^2 are get methods which have performed their access:

$$
K^2 = \text{o}[x_1^2!\langle v_1^2\rangle\ \texttt{end}] \mid \ \ldots\ \text{o}[x_{n_2}^2!\langle v_{n_2}^2\rangle\ \texttt{end}]
$$

The agents in K^4 are newly invoked set methods:

$$K^4 = \begin{array}{l} \mathrm{o}[\texttt{acquire}\ \mathit{lock}\ \texttt{dict}?d\ \texttt{dict!set}(d, k^4_1, v^4_1)\ \texttt{release}\ \mathit{lock}\ x^4_1!\langle\rangle\ \texttt{end}]\ | \\ \vdots \\ \mathrm{o}[\texttt{acquire}\ \mathit{lock}\ \texttt{dict}?d\ \texttt{dict!set}(d, k^4_{n_4}, v^4_{n_4})\ \texttt{release}\ \mathit{lock}\ x^4_{n_4}!\langle\rangle\ \texttt{end}] \end{array}$$

The agent in K^{set} is either Nil or has one of the forms:

(i) $\mathrm{o}[\texttt{dict}?d\ \texttt{dict!set}(d, k, v)\ \texttt{release}\ \mathit{lock}\ x!\langle\rangle\ \texttt{end}]$

(ii) $\mathrm{o}[\texttt{dict}!D'\ \texttt{release}\ \mathit{lock}\ x!\langle\rangle\ \texttt{end}]$

(iii) $\mathrm{o}[\texttt{release}\ \mathit{lock}\ x!\langle\rangle\ \texttt{end}]$

The agents in K^8 are get methods which have performed their access:

$$K^8 = \mathrm{o}[x^8_1!\langle\rangle\ \texttt{end}]\ |\ \ldots\ \mathrm{o}[x^8_{n_8}!\langle v^8_{n_8}\rangle\ \texttt{end}]$$

The agents in K^{end} are get or set methods which have finished their work.

$$K^{\text{end}} = \mathrm{o}[\texttt{end}]\ |\ \ldots\ \mathrm{o}[\texttt{end}]$$

We will show that our configuration Kstate is weakly bisimilar to the following sequential process expression:

$$\begin{array}{llll} q_{\text{Dstate}} \stackrel{\text{def}}{=} & \bigstar_{k \in \text{Key}, x\ \text{new}} & \xrightarrow[x]{\mathrm{o}.\texttt{get}?\langle k\rangle} & q_{h_1(\text{Dstate}, x, k)} \\ \star & \bigstar_{k \in \text{Key}, v \in \text{Value}, x\ \text{new}} & \xrightarrow[x]{\mathrm{o}.\texttt{set}?\langle k, v\rangle} & q_{h_2(\text{Dstate}, x, k, v)} \\ \star & \bigstar_{(x,v) \in \text{Out}_1} & \xrightarrow{x!\langle v\rangle} & q_{h_3(\text{Dstate}, x, v)} \\ \star & \bigstar_{x \in \text{Out}_2} & \xrightarrow{x!\langle\rangle} & q_{h_4(\text{Dstate}, x)} \\ \star & \bigstar_{(x,k) \in \text{In}_1, v = \texttt{get}(\text{Dict}, k)} & \longrightarrow & q_{h_5(\text{Dstate}, x, k, v)} \\ \star & \bigstar_{(x,k,v) \in \text{In}_2, \text{Pending} = \emptyset} & \longrightarrow & q_{h_6(\text{Dstate}, x, k, v)} \\ \star & \bigstar_{(x,k,v) \in \text{Pending}} & \longrightarrow & q_{h_7(\text{Dstate}, x, k, v)} \end{array}$$

where the definitions of functions $h_1, \ldots h_7$ are given on page 208.

It is not difficult to show that $q_{\text{Dstate}_\perp}\ \mathsf{s\tilde{a}t}\ \text{Dspec}^{5'}_{\text{Dstate}_\perp}$. To show that $\text{Kstate}_\perp \approx q_{\text{Dstate}_\perp}$ we need to define a weak bisimulation. We now show that

we can obtain a meaningful projection from configurations of the above form into the state components of the sequential process expression above. Given such a configuration Kstate, we define the following:

$$
\begin{aligned}
\pi_1(\text{Kstate}) &= D \\
\pi_2(\text{Kstate}) &= \{(x_i^1, k_i^1) \mid i \in 1..n_1\} \\
\pi_3(\text{Kstate}) &= \{(x_i^2, v_i^2) \mid i \in 1..n_2\} \\
\pi_4(\text{Kstate}) &= \{(x_i^4, k_i^4, v_i^4) \mid i \in 1..n_6\} \\
\pi_5(\text{Kstate}) &= \{x^8 \mid i \in 1..n_8\} \\
& \quad \cup \begin{cases} \{x\} & \text{if } K^{\text{set}} \text{ is in state (iii)} \\ \emptyset & \text{otherwise} \end{cases} \\
\pi_6(\text{Kstate}) &= \begin{cases} \{(x, k, v)\} & \text{if } K^{\text{set}} \text{ is in state (i)} \\ f(D, K^{\text{set}}) & \text{if } K^{\text{set}} \text{ is in state (ii)} \\ \emptyset & \text{otherwise} \end{cases}
\end{aligned}
$$

In the definition of π_6, we must be careful if the K^{set} is in state (ii), since there are no explicit k and v values in the agent. We use the function $f(D, K^{\text{set}})$ to obtain such values as follows: First compare the dictionary value D' in the agent with the value D. If they differ, then the key value pair at which they differ (this occurs at most once) provides our k and v. If they are identical, then the update won't affect the dictionary, so we can pick an arbitrary key value pair, k and v, which agrees with D. $f(D, K^{\text{set}})$ returns (x, k, v) where x is obtained from the agent in K^{set}.

We then define π as follows:

$$
\pi(\text{Kstate}) = \begin{array}{l} (\pi_1(\text{Kstate}), \pi_2(\text{Kstate}), \pi_3(\text{Kstate}), \\ \quad \pi_4(\text{Kstate}), \pi_5(\text{Kstate}), \pi_6(\text{Kstate})) \end{array}
$$

We now show the following is a weak bisimulation between Kstate and $q_{\pi(\text{Kstate})}$:

$$
\mathcal{S} = \{(\text{Kstate}, q_{\pi(\text{Kstate})}) \mid \forall \text{Kstate}\}
$$

Given a pair $(\text{Kstate}, q_{\pi(\text{Kstate})})$, we check weak simulation by considering all actions of one component and making sure that the other component can perform an experiment with the same visible action.

First we consider the actions of Kstate:

- $\xrightarrow[x]{\texttt{o.get?}\langle k\rangle}$: a new get request arrives and a new K^1 agent is created. This is simulated by $q_{\pi(\text{Kstate})}$ performing $\xrightarrow[x]{\texttt{o.get?}\langle k\rangle}$.

- $\xrightarrow[x]{\texttt{o.set?}\langle k,v\rangle}$: a new set request arrives and a new K^4 agent is created. This is simulated by $q_{\pi(\text{Kstate})}$ performing $\xrightarrow[x]{\texttt{o.set?}\langle k,v\rangle}$.

- $\xrightarrow{x!\langle v\rangle}$: a reply to a get request is issued and a K^2 agent is converted into a K^{end} agent. This is simulated by $q_{\pi(\text{Kstate})}$ performing $\xrightarrow{x!\langle v\rangle}$.

- $\xrightarrow{x!\langle\rangle}$: a reply to a set request is issued and a K^8 agent is converted into a K^{end} agent. This is simulated by $q_{\pi(\text{Kstate})}$ performing $\xrightarrow{x!\langle\rangle}$.

- $\longrightarrow$: a K^1 agent accesses the dictionary and becomes a K^2 agent. Let (x, k) be the values in the K^1 agent and let $v = \mathsf{get}(D, k)$. The action is simulated by $q_{\pi(\text{Kstate})}$ performing a silent action and becoming state $q_{h_5(\pi(\text{Kstate}),x,k,v)}$. It can perform this action because $(x, k) \in \pi_2(\text{Kstate})$ and $v = \mathsf{get}(\pi_1(\text{Kstate}), k)$.

- $\longrightarrow$: a K^4 agent acquires the lock and becomes a K^{set} agent in state (i). This can occur only when there is no other K^{set} agent. Let (x, k, v) be the values in the K^4 agent. The action is simulated by $q_{\pi(\text{Kstate})}$ performing a silent action and becoming state $q_{h_6(\pi(\text{Kstate}),x,k,v)}$. It can perform this action because $(x, k, v) \in \pi_4(\text{Kstate})$ and $\pi_6(\text{Kstate}) = \emptyset$.

- $\longrightarrow$: a K^{set} agent in state (i) becomes a K^{set} agent in state (ii). This is simulated by $q_{\pi(\text{Kstate})}$ doing nothing.

- $\longrightarrow$: a K^{set} agent in state (ii) becomes a K^{set} agent in state (iii). Let $(x, k, v) = f(D, K^{\text{set}})$. The action is simulated by $q_{\pi(\text{Kstate})}$ performing a silent action and becoming state $q_{h_7(\pi(\text{Kstate}),x,k,v)}$. It can perform this action because $(x, k, v) \in \pi_6(\text{Kstate})$.

- $\longrightarrow$: a K^{set} agent in state (iii) becomes a K^8 agent. This is simulated by $q_{\pi(\text{Kstate})}$ doing nothing.

- $\longrightarrow$: a K^{end} agent is removed. This is simulated by $q_{\pi(\text{Kstate})}$ doing nothing.

Now we must reason that Kstate can simulate any of the actions performed by $q_{\pi(\text{Kstate})}$.

- $\xrightarrow[x]{\text{o.}\mathtt{get}?\langle k\rangle}$ leading to state $q_{h_1(\pi(\text{Kstate}),x,k)}$. This is simulated by Kstate performing $\xrightarrow[x]{\text{o.}\mathtt{get}?\langle k\rangle}$, whereby a new K^1 agent is created.

- $\xrightarrow[x]{\text{o.}\mathtt{set}?\langle k,v\rangle}$ leading to state $q_{h_2(\pi(\text{Kstate}),x,k,v)}$. This is simulated by Kstate performing $\xrightarrow[x]{\text{o.}\mathtt{set}?\langle k,v\rangle}$, whereby a new K^4 agent is created.

- $\xrightarrow{x!\langle v\rangle}$ leading to state $q_{h_3(\pi(\text{Kstate}),x,v)}$. This is simulated by Kstate performing $\xrightarrow{x!\langle v\rangle}$, whereby a K^4 agent is converted into a K^{end} agent.

- $\xrightarrow{x!\langle\rangle}$ leading to state $q_{h_4(\pi(\text{Kstate}),x)}$. This is simulated by Kstate performing $\xrightarrow{x!\langle\rangle}$, whereby a K^8 agent is converted into a K^{end} agent.

- $\longrightarrow$ leading to state $q_{h_5(\pi(\text{Kstate}),x,k,v)}$ where $v = \mathsf{get}(\pi_1(\text{Kstate}), k)$. Since $(x, k) \in \pi_2(\text{Kstate})$, we know there is a K^1 agent with the appropriate values. Therefore, Kstate can perform a silent action and convert that K^1 agent into a K^2 agent.

- $\longrightarrow$ leading to state $q_{h_6(\pi(\text{Kstate}),x,k,v)}$. For this to be occur, we know that $\pi_6(\text{Kstate}) = \emptyset$. Therefore, if there is a K^{set} agent, it must be in state (iii) and can perform a silent action to release its lock. Since $(x, k, v) \in \pi_4(\text{Kstate})$, there must be a K^4 agent with the appropriate values. Therefore, Kstate can perform a silent action and convert the that K^4 agent into a K^{set} agent in state (i).

- $\longrightarrow$ leading to state $q_{h_7(\pi(\text{Kstate}),x,k,v)}$. Thus $\pi_6(\text{Kstate}) = \{(x, k, v)\}$ and, consequently, there must be a K^{set} agent in state (i) or (ii). Thus, Kstate can perform one or two silent actions to convert this into a K^{set} agent in state (iii).

Thus, $\mathcal{S}$ is a weak bisimulation. This means that the implementation relation holds between the configurations of the objects of the class we presented and the specifications $\text{Dspec}^{5'}_{\text{Dstate}}$.

7.3.5 Further Development

The designs we presented in Section 7.3.1 and 7.3.3 are intended to illustrate our development method. However, as actual designs for a concurrent dictionary they are quite unsophisticated. Both designs involve a single object which manages the whole dictionary. They require the entire dictionary to be locked all at once. We briefly consider here some more sophisticated designs for a concurrent dictionary that would also satisfy the specification.

Rather than having a single object manage the dictionary, we could split the dictionary into separate objects, each of which would have responsibility for only a subset of the set of keys. This gives us a much finer control over locking and would allow us to share the task of running the dictionary to many distributed machines. Designs of this sort are considered in [Jon92].

One such design would involve a set of Oompa objects arranged as a linked list. Each object in the list would manage a single key value pair. Set and get requests can concurrently traverse the list looking for the object managing their key.

Another very standard design for a dictionary would be a *binary search tree*, although we would need to assume that the set of keys has some kind of ordering on it. In this design, the objects are arranged in a binary tree formation and, as in the previous design, each object is responsible for a single key value pair. The core property of this design is the following: if an object, o, is managing key, k, then

- all the objects in the left subtree beneath o are managing keys less than k and
- all the objects in the right subtree beneath o are managing keys greater than k.

Again, many concurrent requests can be operating on the tree at any instant.

7.4 Summary

In this chapter, we demonstrated how Sequential Process Expressions and Oompa might be used in actual development. We derived two designs for a very standard structure, a concurrent dictionary.

The first step was to discover the behavioural requirements of a concurrent dictionary object. We started with the abstract data type of a dictionary, which defines the abstract behaviour of its get and set operations. Before tackling concurrency, we provided the requirements for a sequential dictionary object. We were then able to give the requirements for a concurrent dictionary object.

Our next step was to give a specification of behaviour that satisfied those requirements. Using sequential process specifications, we derived two specifications for the behaviour of a concurrent dictionary. Unfortunately, there was no generalisation of both specifications in our language.

Lastly, we provided two Oompa classes, one for each specification, which embody our designs for a concurrent dictionary. We considered the objects that these classes define and we were able to show that those objects lie in the implementation relation ($\approx \tilde{\mathsf{sat}}$) with their specification.

Chapter 8

Conclusions and Future Work

The goal of this research was to provide support for the modelling and development of distributed object-oriented systems. We decided to provide a special-purpose formal method to this end. Although our formal method is in an early stage of development, our research has lead to the development of two components, Oompa and sequential process specifications. In this final chapter, we review our achievements and discuss how we might take the research forward.

We review the contents of the thesis in Section 8.1 and consider future work in Section 8.2.

8.1 Overview

In this section, we review the material we presented in this thesis.

Our first task was to frame our problem by choosing an interpretation of "distributed object-oriented systems". Because of its generality, we decided to base our interpretation on the CORBA object model.

We commenced the thesis proper with a review of the current state of

the art. We considered various methods which we divided into three classes: state-based approaches to concurrency, process calculi and concurrent object-oriented approaches. We evaluated the methods as approaches to our problem but, ultimately, found that they did not precisely satisfy our requirements.

The bulk of the material in this thesis was concerned with Oompa. All formal methods need a language for representing the concepts they work with. We intended to discuss designs for distributed object-oriented systems so, in line with our decision to avoid a semantic gap, we defined a language which embodies many of the features of those systems. Oompa is an object-oriented language whose objects support multiple concurrent invocations.

Our presentation of Oompa consisted of three parts: the static system, which defines the syntax of Oompa programs, the dynamic systems, which define their behaviour and the type system, which guarantees their type correctness. In defining Oompa's static and dynamic systems, we gave a thorough consideration to issues of naming and devised some algebraic techniques such as isolated sum (see Appendix A) and ρ^{α}-constructions (see Appendix B, Section B.1). In defining Oompa's type system, we produced a minimal type system suitable for languages with objects and channels. We supplied this type system with a way of handling mutually referential class and interface definitions, the expansion function (see Section 5.2.3).

In order to discuss and manipulate behaviour, we also developed a behavioural specification language called sequential process specifications. We gave its syntax and defined its semantics in terms of a notion of satisfaction. Although similar, we showed that the language is semantically different to CSP (see Section 6.4.2). Most significantly, we defined a form of refinement for our language which allows specifications to be developed incrementally.

The refinement steps of such a development can be formally verified and, importantly, can be justified in terms of requirements (see Section 6.6 and Section 7.2). We described how the behaviour of Oompa programs can be specified with sequential process specifications.

We indicated how our formal method might support development by producing two designs for a concurrent dictionary object. First, we derived behavioural requirements for such a dictionary object by semi-formal reasoning about its traces. From these requirements, we were able to incrementally develop two behavioural specifications using sequential process specifications. Lastly, we provided two Oompa class definitions and proved that their objects were implementations of the specifications. These Oompa classes represent two simple designs for a concurrent dictionary objects.

8.2 Future Work

In this section, we describe some possible future work.

The are many ways that our work on the language Oompa could be extended. Considering first the static system, we note that our current approach to naming requires many very similar results to be proved from scratch. We believe that there should be a good generalisation of our naming techniques. Finding one would hopefully simplify many of the proofs in Appendix B. Alternatively, a full investigation of the approaches we discussed in Section 4.2.2 might lead us to take advantage of their techniques.

A useful extension to Oompa's static system would be the addition of an expression language. Currently, Oompa's computational syntax is minimal and implementing algorithms of any sophistication involves numerous encodings. An expression language would make Oompa programs much easier to

write. There are two ways this might be done. One way would be to extend Oompa itself by providing new syntactic forms and rules in the operational semantics to give them behaviour. Alternatively, we could define sugarings from an enriched Oompa syntax to the base language described in this thesis.

Considering Oompa's dynamic systems, we note that the single result we proved about the relationship between the two operational semantics is only one of many. We chose this result as it guarantees that the labelled transition system is consistent with the reduction system. Proving further relationships that hold between them would strengthen our theory. For example, we could show that all reductions have an analogous silent transition in the labelled transition system.

Oompa's labelled transition system requires the use of an equivalence system. Even though we do take advantage of it in Chapter 7, this feature does make reasoning over proof trees awkward. Many comparable systems in the literature (e.g. the labelled transition system of the π-calculus camstpc) provide a labelled transition system which does not require equivalence. It might be worth developing a third operational semantics without this requirement. The main reason we do require an equivalence system is to ensure that agents have access to their object's state. Without the ability to manipulate configurations using equivalence, we might need to consider transitions which describe access or update operations. Of course, this might be interesting in its own right.

Oompa's configuration-based dynamic system uses weak bisimulation to identify equivalent behaviour. However, weak bisimulation (as we discussed in Section 6.2.1) is only one possibility and there are many alternative behavioural equivalences we could have chosen. Weak bisimulation was a conservative choice, since it is relatively strong and it is unlikely that we would

ever want to be more discriminating. Another way of taking our work further would be a full investigation of alternatives. This might lead us to prefer a weaker equivalence more suited to the systems we wish to discuss.

Although intended to represent designs for distributed object-oriented systems, Oompa has no explicit notion of distribution. We could augment Oompa with explicit distribution and there are examples of how this can be done in the literature [CG98, Car94, RH98]. One way would be to add a notion of location and ensure Oompa objects reside at separate locations. Location names might be values which can then be communicated. Locations become interesting when we limit communication between them or allow a location or an inter-location communication to fail. The particular choice of how we might do this is likely to be heavily influenced by the CORBA standard.

Sequential process specifications, as defined in this thesis, are only a step on the way to the behavioural specification language we are looking for. Ultimately, we plan on developing a language with a similar support for refinement but which has the flexibility and power of the π-calculus. This is likely to require a thorough reworking of the material of Chapter 6.

With regard to the language we have presented, however, there are a few useful extensions we could make. One particularly appropriate change would be to add support for object behaviour as standard. In our example in Chapter 7, we had to explicitly manage the fact that our specifications represented the behaviour of an object.

There are two obvious ways of taking our theory of refining simulation forward. Refining simulation is not complete with respect to refinement. We could seek to strengthen it, perhaps to handle more cases or even to make it complete. Also, we could try to define an algorithm to find refinement

simulations. Inspiration could be taken from our work on Oompa's subtyping system, where an algorithm is provided to determine whether a simulation holds between two infinite structures.

Clearly, this thesis only presents the start of a formal method. Much work will need to be done to build it into a full method. Considered as a development method, a key feature to add would be support for deriving Oompa programs from specifications. We also proposed our method for modelling but we have not, as yet, explored this use. For both development and modelling, further case studies must be done to develop techniques and evaluate the method as a whole.

One task that might greatly increase the usability of our formal method would be the development of tools. Currently, an Oompa parser and parts of a pretty-printer have been written using the Haskell programming language [Jon03]. The automation of well-formedness and type safety checking would be desirable. It would also be useful to be able to animate the dynamic systems.

Appendix A

Isolated Sum

For many of the proofs in Appendix B it will be necessary to build renamings with specific properties, usually by composing other renamings. A situation can arise where two renamings we wish to apply potentially interfere with each other. This short appendix presents our way of managing this, called isolated sum, which constructs a composition of yet more renamings. To avoid long composition expressions, we provide this construction with a notation that we find intuitive.

Section A.1 gives the definition of isolated sum and proves that it has the properties we require. In Section A.2, we elaborate some of the theory of isolated sum. Section A.3 generalises it from two to an arbitrary number of renamings.

A.1 Definition

Say the sets A and B are disjoint. We will often want renamings whose effect is to manipulate the two sets independently. In general, the changes we wish to apply to one set may interfere with the other, so we provide a way of

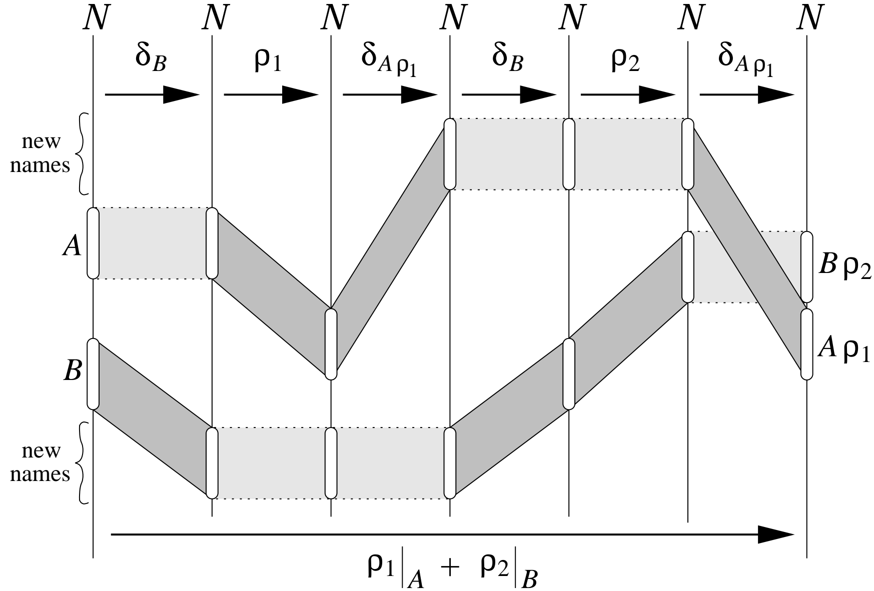
temporarily isolating them.

Say A and B are finite sets of names such that $A \pitchfork B$. Say ρ_1 and ρ_2 are renamings such that $A\rho_1 \pitchfork B\rho_2$. We define the *isolated sum* of ρ_1 and ρ_2 with respect to A and B as

$$\rho_1|_A + \rho_2|_B = \delta_{A\rho_1} \circ \rho_2 \circ \delta_B \circ \delta_{A\rho_1} \circ \rho_1 \circ \delta_B$$

where $\delta_{A\rho_1}$ shoves $A\rho_1$ and δ_B shoves B.

We will refer to the sets A and B in an isolated sum $\rho_1|_A + \rho_2|_B$ as the *isolated sets*. Note that the restrictions in an isolated sum are part of the whole expression and not part of the summands. The following diagram illustrates how isolated sum enables two potentially interfering renamings to be performed "simultaneously".



We say that two renamings ρ_1 and ρ_2 *agree on a set* A if, for all $n \in A$, we have $n\rho_1 = n\rho_2$. We will write this as $\rho_1 \simeq_A \rho_2$.

Lemma A.1.1

1. $\rho_1|_A + \rho_2|_B \simeq_A \rho_1$.

2. $\rho_1|_A + \rho_2|_B \simeq_B \rho_2$.

Proof: Say $n \in A$. Then

$$
\begin{array}{ll}
 & n(\rho_1|_A + \rho_2|_B) \\
= & \langle \text{ definition of isolated sum, choosing } \delta_{A\rho_1} \text{ and } \delta_B \text{ appropriately } \rangle \\
 & n\delta_B\rho_1\delta_{A\rho_1}\delta_B\rho_2\delta_{A\rho_1} \\
= & \langle n \notin \text{Change}(\delta_B) \rangle \\
 & n\rho_1\delta_{A\rho_1}\delta_B\rho_2\delta_{A\rho_1} \\
= & \langle n\rho_1\delta_{A\rho_1} \notin \text{Change}(\delta_B) \rangle \\
 & n\rho_1\delta_{A\rho_1}\rho_2\delta_{A\rho_1} \\
= & \langle n\rho_1\delta_{A\rho_1} \notin \text{Change}(\rho_2) \rangle \\
 & n\rho_1\delta_{A\rho_1}\delta_{A\rho_1} \\
= & \langle {\delta_{A\rho_1}}^{-1} = \delta_{A\rho_1} \rangle \\
 & n\rho_1
\end{array}
$$

The case for $n \in B$ is similar. ■

Corollary A.1.2 $(A \cup B)(\rho_1|_A + \rho_2|_B) = A\rho_1 \cup B\rho_2$

A.2 Some Properties of Isolated Sum

The behaviour of an isolated sum on a name which is outside its isolated sets is unreliable but it has nice properties when we restrict attention to those sets. To support reasoning about isolated sums, we use agreement to capture the notion of "equality on the isolated sets".

When an isolated sum $\rho_1|_A + \rho_2|_B$ and a renaming ρ agree on the sum's isolated sets, i.e. $\rho_1|_A + \rho_2|_B \simeq_{A \cup B} \rho$, then we will just say that they *agree*. Where there is no ambiguity we will drop the set marking on the relation and write this as either $\rho_1|_A + \rho_2|_B \simeq \rho$ or $\rho \simeq \rho_1|_A + \rho_2|_B$.

Lemma A.2.1 $\rho|_A + \rho|_B \simeq \rho$

Lemma A.2.2 $\rho_1|_A + \rho_2|_B \simeq \rho_2|_B + \rho_1|_A$

Lemma A.2.3

1. *If* $\rho_1 \simeq_A \rho_2$ *then* $\rho_1|_A + \rho|_B \simeq \rho_2|_A + \rho|_B$.

2. *If* $\rho_1 \simeq_B \rho_2$ *then* $\rho|_A + \rho_1|_B \simeq \rho|_A + \rho_2|_B$.

Lemma A.2.4 *If* $\rho_1 \simeq \rho_2$ *then* $\rho_2 \simeq \rho_1$.

In general, agreement is not transitive because isolated sums may have different isolated sets. However, if we are careful, the following result allows us to reason in a similar way.

Lemma A.2.5 *Say that* $\rho \simeq \rho_1|_A + \rho_2|_B$ *and* $\rho_1|_A + \rho_2|_B \simeq \rho'$. *Then* $\rho \simeq_{A\cup B} \rho'$.

Lemma A.2.6

1. $\rho \circ (\rho_1|_A + \rho_2|_B) \simeq (\rho \circ \rho_1)|_A + (\rho \circ \rho_2)|_B$

2. $(\rho_1|_A + \rho_2|_B) \circ \rho \simeq (\rho_1 \circ \rho)|_{A\rho^{-1}} + (\rho_2 \circ \rho)|_{B\rho^{-1}}$

Proof:

1. Since $\rho_1|_A + \rho_2|_B$ is well defined, we can conclude that $A \pitchfork B$ and $A\rho_1 \pitchfork B\rho_2$. Thus, since ρ is bijective, we have $A\rho_1\rho \pitchfork B\rho_1\rho$ so the isolated sum $(\rho \circ \rho_1)|_A + (\rho \circ \rho_2)|_B$ is well defined.

   Say $n \in A$. Then

$$
\begin{array}{ll}
& n(\rho \circ (\rho_1|_A + \rho_2|_B)) \\
= & \langle \text{definition of composition} \rangle \\
& n(\rho_1|_A + \rho_2|_B)\rho \\
= & \langle \text{by Lemma A.1.1} \rangle \\
& n\rho_1\rho \\
= & \langle \text{definition of composition} \rangle \\
& n(\rho \circ \rho_1) \\
= & \langle \text{by Lemma A.1.1} \rangle \\
& n((\rho \circ \rho_1)|_A + (\rho \circ \rho_2)|_B)
\end{array}
$$

The case where $n \in B$ is similar.

2. Since $\rho_1|_A + \rho_2|_B$ is well defined, we can conclude that $A \pitchfork B$ and $A\rho_1 \pitchfork B\rho_2$. Thus, since ρ is bijective, we have $A\rho^{-1} \pitchfork B\rho^{-1}$ so the isolated sum $(\rho_1 \circ \rho)|_{A\rho^{-1}} + (\rho_2 \circ \rho)|_{B\rho^{-1}}$ is well defined.

   Say $n \in A\rho^{-1}$. Then

$$
\begin{array}{ll}
& n((\rho_1|_A + \rho_2|_B) \circ \rho) \\
= & \langle \text{definition of composition} \rangle \\
& n\rho(\rho_1|_A + \rho_2|_B) \\
= & \langle \text{by Lemma A.1.1} \rangle \\
& n\rho\rho_1 \\
= & \langle \text{definition of composition} \rangle \\
& n(\rho_1 \circ \rho) \\
= & \langle \text{by Lemma A.1.1} \rangle \\
& n((\rho_1 \circ \rho)|_{A\rho^{-1}} + (\rho_2 \circ \rho)|_{B\rho^{-1}})
\end{array}
$$

   The case where $n \in B\rho^{-1}$ is similar.

■

A.3 n-Ary Isolated Sum

We can define an n-ary version of isolated sum in terms of the binary operation by repeatedly applying the following rule:

$$\rho_1|_{A_1} + \cdots \rho_n|_{A_n} = (\rho_1|_{A_1} + \ldots \rho_{n-1}|_{A_{n-1}})\Big|_{(A_1 \cup \cdots A_{n-1})} + \rho_n|_{A_n}$$

We require the following condition to ensure that the n-ary isolated sum will be well-defined:

$$\forall i \neq j \ . \ ((A_i \pitchfork A_j) \ \wedge \ (A_i\rho_i \pitchfork A_j\rho_j))$$

We can prove n-ary analogues of the properties of binary isolated sum.

Lemma A.3.1 *For $i \in 1..n$,*

$$\rho_1|_{A_1} + \cdots \rho_n|_{A_n} \simeq_{A_i} \rho_i$$

Proof: We use induction on n. The base case is where $n = 2$ and this follows directly from Lemma A.1.1. Say that $n > 2$ and assume that the result is true for all m such that $2 \leq m < n$.

There are two cases for i. First, we consider the case where $i = n$.

$$\begin{array}{ll} & \rho_1|_{A_1} + \cdots \rho_n|_{A_n} \\ = & \langle \text{definition of } n\text{-ary isolated sum} \rangle \\ & (\rho_1|_{A_1} + \ldots \rho_{n-1}|_{A_{n-1}})\Big|_{(A_1 \cup \cdots A_{n-1})} + \rho_n|_{A_n} \\ \simeq_{A_n} & \langle \text{by Lemma A.1.1} \rangle \\ & \rho_n \end{array}$$

as required in this case.

Second, we consider the case where $1 \leq i < n$.

$$
\begin{aligned}
& \rho_1|_{A_1} + \cdots \rho_n|_{A_n} \\
= \quad & \langle \text{definition of } n\text{-ary isolated sum} \rangle \\
& (\rho_1|_{A_1} + \ldots \rho_{n-1}|_{A_{n-1}})\Big|_{(A_1 \cup \cdots A_{n-1})} + \rho_n|_{A_n} \\
\simeq_{A_1 \cup \cdots A_{n-1}} & \langle \text{by Lemma A.1.1} \rangle \\
& \rho_1|_{A_1} + \ldots \rho_{n-1}|_{A_{n-1}}
\end{aligned}
$$

Therefore we have:

$$
\begin{aligned}
& \rho_1|_{A_1} + \cdots \rho_n|_{A_n} \\
\simeq_{A_i} \quad & \langle A_i \subseteq A_1 \cup \cdots A_{n-1} \rangle \\
& \rho_1|_{A_1} + \ldots \rho_{n-1}|_{A_{n-1}} \\
\simeq_{A_i} \quad & \langle \text{induction hypothesis} \rangle \\
& \rho_i
\end{aligned}
$$

The transitivity of agreement on A_i gives the result. ■

Corollary A.3.2 $(A_1 \cup \cdots A_n)(\rho_1|_{A_1} + \cdots \rho_n|_{A_n}) = A_1\rho_1 \cup \cdots A_n\rho_n$

Lemma A.3.3 $\rho|_{A_1} + \cdots \rho|_{A_n} \simeq \rho$

The n-ary version of Lemma A.2.2 allows the permuting of summands.

Lemma A.3.4 *If α is a permutation of $1..n$, then*

$$\rho_1|_{A} + \cdots \rho_n|_{A_n} \simeq \rho_{1\alpha}|_{A_{1\alpha}} + \cdots \rho_{n\alpha}|_{A_{n\alpha}}$$

Lemma A.3.5 *If $\rho_i \simeq_{A_i} \rho'_i$ for $i \in 1..n$ then*

$$\rho_1|_{A_1} + \cdots \rho_i|_{A_i} + \cdots \rho_n|_{A_n} \simeq \rho_1|_{A_1} + \cdots \rho'_i|_{A_i} + \cdots \rho_n|_{B_n}$$

Lemma A.3.6 *Say that $\rho \simeq \rho_1|_{A_1} + \cdots \rho_n|_{A_n}$ and $\rho_1|_{A_1} + \cdots \rho_n|_{A_n} \simeq \rho'$. Then $\rho \simeq_{A_1 \cup \cdots A_n} \rho'$.*

Lemma A.3.7

1. $\rho \circ (\rho_1|_{A_1} + \cdots \rho_n|_{A_n}) \simeq (\rho \circ \rho_1)|_{A_1} + \cdots (\rho \circ \rho_n)|_{A_n}$

2. $(\rho_1|_{A_1} + \cdots \rho_n|_{A_n}) \circ \rho \simeq (\rho_1 \circ \rho)|_{A_1\rho^{-1}} + \cdots (\rho_n \circ \rho)|_{A_n\rho^{-1}}$

Appendix B

Technical Results for Oompa

In this appendix, we elaborate on the technical aspects of our presentation of Oompa. Most importantly, the proofs of many results stated in chapters 3 and 4 are given here. When a result is restated, it is annotated with a reference to the corresponding result in the body of the thesis and the page number where it occurs.

Section B.1 provides the proofs of several properties of the α-equivalence system for code. Also in this section is a description of a proof technique we make use of many times called the ρ^α-construction. In Section B.2, we prove some results about agents necessary for the rest of the appendix. Section B.3 gives the proofs for many of the results about configurations. We prove the relationship between the two dynamic systems in Section B.4.

B.1 Results about Code

In this section we give the proofs of lemmas 3.5.1, 3.5.2 and 3.5.3.

3.5.1 p75

Lemma B.1.1 *For two pieces of code p_1 and p_2,*

$$p_1 \equiv_\alpha p_2 \Rightarrow \mathrm{FN}(p_1) = \mathrm{FN}(p_2)$$

Proof: We use induction on the proof tree that ends with $p_1 \equiv_\alpha p_2$. If the last step is justified by an equivalence rule or congruence rule then the argument is obvious. We provide one example where the last step is a change of bound names rule.

Say the proof tree is an instance of CHNG-INV:

$$o.m!\langle v_1, \ldots v_n \rangle ?(r_1, \ldots r_{n'})\ p \equiv_\alpha o.m!\langle v_1, \ldots v_n \rangle ?(r_1\rho, \ldots r_{n'}\rho)\ p\rho$$

where

$$\mathrm{FN}(o.m!\langle v_1, \ldots v_n \rangle ?(r_1, \ldots r_{n'})\ p) \pitchfork \mathrm{Change}(\rho)$$

Then

$$\begin{array}{ll} & \mathrm{FN}(o.m!\langle v_1, \ldots v_n \rangle ?(r_1\rho, \ldots r_{n'}\rho)\ p\rho) \\ = & \langle \text{definition of } \mathrm{FN}(\cdot) \rangle \\ & (\mathrm{FN}(p\rho) \setminus \{r_1\rho, \ldots r_{n'}\rho\}) \cup \{o\} \cup \mathrm{FN}(v_1) \cup \ldots \mathrm{FN}(v_n) \\ = & \langle \text{by Lemma 3.4.5 and set theory} \rangle \\ & (\mathrm{FN}(p)\rho \setminus \{r_1, \ldots r_{n'}\}\rho) \cup \{o\} \cup \mathrm{FN}(v_1) \cup \ldots \mathrm{FN}(v_n) \\ = & \langle \text{since } \rho \text{ is bijective} \rangle \\ & (\mathrm{FN}(p) \setminus \{r_1, \ldots r_{n'}\})\rho \cup \{o\} \cup \mathrm{FN}(v_1) \cup \ldots \mathrm{FN}(v_n) \\ = & \langle (\mathrm{FN}(p) \setminus \{r_1, \ldots r_{n'}\}) \pitchfork \mathrm{Change}(\rho) \rangle \\ & (\mathrm{FN}(p) \setminus \{r_1, \ldots r_{n'}\}) \cup \{o\} \cup \mathrm{FN}(v_1) \cup \ldots \mathrm{FN}(v_n) \\ = & \langle \text{definition of } \mathrm{FN}(\cdot) \rangle \\ & \mathrm{FN}(o.m!\langle v_1, \ldots v_n \rangle ?(r_1, \ldots r_{n'})\ p) \end{array}$$

The second last step is justified by the fact that

$$\mathrm{FN}(p) \setminus \{r_1, \ldots r_{n'}\} \subseteq \mathrm{FN}(o.m!\langle v_1, \ldots v_n \rangle ?(r_1, \ldots r_{n'})\ p)$$

■

We use the proof of the following lemma to exemplify a technique we will use repeatedly. For this first use, we will give all the details but, later, we will leave out most of the steps. We call this technique a ρ^α*-construction*.

A ρ^α-construction will be applicable when we are using structural induction to show that a pair of Oompa structures are α-equivalent. In each case, we will construct a renaming, written ρ^α, which has two key properties. Firstly, it will map the bound names of the outermost binder of one structure to the bound names of the outermost binder of the other. Secondly, it will leave the free names of the structures alone. In simple cases, ρ^α can be built using composition but, in more complicated cases, ρ^α will be an isolated sum.

3.5.2 p75

Lemma B.1.2 *For a piece of code* p *and two renamings* ρ_1 *and* ρ_2*,*

$$p\rho_1\rho_2 \equiv_\alpha p(\rho_2 \circ \rho_1)$$

Proof: We use induction on the construction of the piece of code p. We give two examples of the cases, which illustrate the argument in general.

Consider the case where $p = c!\langle v_1, \ldots v_n\rangle\ p'$. Then

$$\begin{array}{ll} & p\rho_1\rho_2 \\ = & \quad\langle\text{expanding } p\rangle \\ & (c!\langle v_1, \ldots v_n\rangle\ p')\rho_1\rho_2 \\ = & \quad\langle\text{renaming procedure on } \rho_1\rangle \\ & (c\rho_1!\langle v_1\rho_1, \ldots v_n\rho_1\rangle\ p'\rho_1)\rho_2 \\ = & \quad\langle\text{renaming procedure on } \rho_2\rangle \\ & c\rho_1\rho_2!\langle v_1\rho_1\rho_2, \ldots v_n\rho_1\rho_2\rangle\ p'\rho_1\rho_2 \\ = & \quad\langle\text{composition}\rangle \\ & c(\rho_2 \circ \rho_1)!\langle v_1(\rho_2 \circ \rho_1), \ldots v_n(\rho_2 \circ \rho_1)\rangle\ p'\rho_1\rho_2 \\ \equiv_\alpha & \quad\langle\text{induction hypothesis and CONG-SND}\rangle \end{array}$$

$$
\begin{aligned}
& c(\rho_2 \circ \rho_1)!\langle v_1(\rho_2 \circ \rho_1), \ldots v_n(\rho_2 \circ \rho_1)\rangle\ p'(\rho_2 \circ \rho_1) \\
= & \quad \langle \text{renaming procedure for } (\rho_2 \circ \rho_1) \rangle \\
& (c!\langle v_1, \ldots v_n \rangle\ p')(\rho_2 \circ \rho_1) \\
= & \quad \langle \text{definition of } p \rangle \\
& p(\rho_2 \circ \rho_1)
\end{aligned}
$$

as required.

Consider the case where $p = \texttt{create}\ o\colon \mathrm{ClT}\ p'$. Then

$$
\begin{aligned}
& p\rho_1\rho_2 \\
= & \quad \langle \text{expand } p \rangle \\
& (\texttt{create}\ o\colon \mathrm{ClT}\ p')\rho_1\rho_2 \\
= & \quad \langle \text{renaming procedure, choosing } \delta_1 \text{ appropriately } \rangle \\
& (\texttt{create}\ o\delta_1\rho_1\colon \mathrm{ClT}\ p'\delta_1\rho_1)\rho_2 \\
= & \quad \langle \text{renaming procedure, choosing } \delta_2 \text{ appropriately} \rangle \\
& \texttt{create}\ o\delta_1\rho_1\delta_2\rho_2\colon \mathrm{ClT}\ p'\delta_1\rho_1\delta_2\rho_2
\end{aligned}
$$

and

$$
\begin{aligned}
& p(\rho_2 \circ \rho_1) \\
= & \quad \langle \text{expand } p \rangle \\
& (\texttt{create}\ o\colon \mathrm{ClT}\ p')(\rho_2 \circ \rho_1) \\
= & \quad \langle \text{renaming procedure, choosing } \delta \text{ appropriately} \rangle \\
& \texttt{create}\ o\delta(\rho_2 \circ \rho_1)\colon \mathrm{ClT}\ p'\delta(\rho_2 \circ \rho_1)
\end{aligned}
$$

Our goal will be to show that

$$
\texttt{create}\ o\delta(\rho_2 \circ \rho_1)\colon \mathrm{ClT}\ p'\delta(\rho_2 \circ \rho_1) \equiv_\alpha \texttt{create}\ o\delta_1\rho_1\delta_2\rho_2\colon \mathrm{ClT}\ p'\delta_1\rho_1\delta_2\rho_2
$$

We will use the change of bound names rule for `create`, CHNG-CRT, with a renaming, ρ^α, which will map $o\delta(\rho_2 \circ \rho_1)$ into $o\delta_1\rho_1\delta_2\rho_2$ and leave free names

alone[1]. Let

$$\rho^\alpha = \rho_2 \circ \delta_2 \circ \rho_1 \circ \delta_1 \circ \delta \circ {\rho_1}^{-1} \circ {\rho_2}^{-1}$$

We first show that ρ^α leaves free names untouched. Consider an instance of a free name, n, occurring in `create` $o\delta(\rho_2 \circ \rho_1)$: ClT $p'\delta(\rho_2 \circ \rho_1)$. Due to the fact that new names are chosen when the renaming procedure encounters binders, free names in p' are not affected by the shove δ. Moreover, any shoves generated when the renaming $(\rho_2 \circ \rho_1)$ is applied to $p'\delta$, will not affect free names. Thus $n = n'(\rho_2 \circ \rho_1)$ for some name n' which is free in p'. We show that $n \notin \text{Change}(\rho^\alpha)$.

$$
\begin{array}{ll}
 & n\rho^\alpha \\
= & \quad \langle \text{definition of } n' \rangle \\
 & n'(\rho_2 \circ \rho_1)\rho^\alpha \\
= & \quad \langle \text{composition} \rangle \\
 & n'(\rho^\alpha \circ \rho_2 \circ \rho_1) \\
= & \quad \langle \text{expand } \rho^\alpha \rangle \\
 & n'(\rho_2 \circ \delta_2 \circ \rho_1 \circ \delta_1 \circ \delta \circ {\rho_1}^{-1} \circ {\rho_2}^{-1} \circ \rho_2 \circ \rho_1) \\
= & \quad \langle ({\rho_1}^{-1} \circ {\rho_2}^{-1} \circ \rho_2 \circ \rho_1) = 1 \rangle \\
 & n'(\rho_2 \circ \delta_2 \circ \rho_1 \circ \delta_1 \circ \delta) \\
= & \quad \langle \text{composition} \rangle \\
 & n'\delta\delta_1\rho_1\delta_2\rho_2 \\
= & \quad \langle \text{choice of } \delta \rangle \\
 & n'\delta_1\rho_1\delta_2\rho_2 \\
= & \quad \langle \text{choice of } \delta_1 \rangle
\end{array}
$$

[1] In this case, it is clearly easier to construct ρ^α manually as the transposition

$$(o\delta(\rho_2 \circ \rho_1)\ o\delta_1\rho_1\delta_2\rho_2)$$

We choose to use a more general approach in order to better illustrate the ρ^α-construction.

$$
\begin{aligned}
& n'\rho_1\delta_2\rho_2 \\
= \quad & \langle \text{choice of } \delta_2 \rangle \\
& n'\rho_1\rho_2 \\
= \quad & \langle \text{composition} \rangle \\
& n'(\rho_2 \circ \rho_1) \\
= \quad & \langle \text{definition of } n' \rangle \\
& n
\end{aligned}
$$

The last few steps of this argument are due to the fact that that δ, δ_1 and δ_2 are shoves generated by the renaming procedure and so won't affect the free names of p' and $p'\rho_1$. So,

$$\mathrm{FN}(\mathtt{create}\ o\delta(\rho_2 \circ \rho_1){:}\,\mathrm{ClT}\ p'\delta(\rho_2 \circ \rho_1)) \pitchfork \mathrm{Change}(\rho^\alpha)$$

Hence, ρ^α can be legitimately used with the CHNG-CRT rule.

Now

$$
\begin{aligned}
& p'\delta(\rho_2 \circ \rho_1)\rho^\alpha \\
\equiv_\alpha \quad & \langle \text{induction hypothesis twice} \rangle \\
& p'(\rho^\alpha \circ \rho_2 \circ \rho_1 \circ \delta) \\
= \quad & \langle \text{expand } \rho^\alpha \rangle \\
& p'(\rho_2 \circ \delta_2 \circ \rho_1 \circ \delta_1 \circ \delta \circ {\rho_1}^{-1} \circ {\rho_2}^{-1} \circ \rho_2 \circ \rho_1 \circ \delta) \\
= \quad & \langle (\delta \circ {\rho_1}^{-1} \circ {\rho_2}^{-1} \circ \rho_2 \circ \rho_1 \circ \delta) = 1 \rangle \\
& p'(\rho_2 \circ \delta_2 \circ \rho_1 \circ \delta_1) \\
\equiv_\alpha \quad & \langle \text{induction hypothesis three times} \rangle \\
& p'\delta_1\rho_1\delta_2\rho_2
\end{aligned}
$$

and, similarly,

$$o\delta(\rho_2 \circ \rho_1)\rho^\alpha = o\delta_1\rho_1\delta_2\rho_2$$

We are finally in a position to use ρ^α.

$\texttt{create}\ o\delta(\rho_2 \circ \rho_1){:}\,\mathrm{ClT}\ p'\delta(\rho_2 \circ \rho_1)$

$\equiv_\alpha$ ⟨CHNG-CRT⟩

$\texttt{create}\ o\delta(\rho_2 \circ \rho_1)\rho^\alpha{:}\,\mathrm{ClT}\ p'\delta(\rho_2 \circ \rho_1)\rho^\alpha$

$\equiv_\alpha$ ⟨from above with CONG-CRT⟩

$\texttt{create}\ o\delta(\rho_2 \circ \rho_1)\rho^\alpha{:}\,\mathrm{ClT}\ p'\delta_1\rho_1\delta_2\rho_2$

$=$ ⟨from above, $o\delta(\rho_2 \circ \rho_1)\rho^\alpha = o\delta_1\rho_1\delta_2\rho_2$⟩

$\texttt{create}\ o\delta_1\rho_1\delta_2\rho_2{:}\,\mathrm{ClT}\ p'\delta_1\rho_1\delta_2\rho_2$

as required. ■

Lemma B.1.3 *For two pieces of code, p_1 and p_2, and a renaming, ρ,*

3.5.3 p75

$$p_1 \equiv_\alpha p_2 \Rightarrow p_1\rho \equiv_\alpha p_2\rho$$

Proof: We use induction on the structure of the proof tree of $p_1 \equiv_\alpha p_2$. We provide two cases which illustrate the approach in general.

Consider the case where the last rule in the proof is CONG-RCV:

$$\frac{p_1' \equiv_\alpha p_2'}{c?(r_1{:}\,T_1, \ldots r_n{:}\,T_n)\ p_1' \equiv_\alpha c?(r_1{:}\,T_1, \ldots r_n{:}\,T_n)\ p_2'}$$

Now

$(c?(r_1{:}\,T_1, \ldots r_n{:}\,T_n)\ p_1')\rho$

$=$ ⟨renaming procedure, choosing δ_1 appropriately⟩

$c\rho?(r_1\delta_1\rho{:}\,T_1, \ldots r_n\delta_1\rho{:}\,T_n)\ p_1'\delta_1\rho$

and

$(c?(r_1{:}\,T_1, \ldots r_n{:}\,T_n)\ p_2')\rho$

$=$ ⟨renaming procedure, choosing δ_2 appropriately⟩

$c\rho?(r_1\delta_2\rho{:}\,T_1, \ldots r_n\delta_2\rho{:}\,T_n)\ p_2'\delta_2\rho$

We use a ρ^α-construction to show that

$$c\rho?(r_1\delta_1\rho\colon T_1, \ldots r_n\delta_1\rho\colon T_n)\ p'_1\delta_1\rho \equiv_\alpha c\rho?(r_1\delta_2\rho\colon T_1, \ldots r_n\delta_2\rho\colon T_n)\ p'_2\delta_2\rho$$

Let

$$\rho^\alpha = \rho \circ \delta_1 \circ \delta_2 \circ \rho^{-1}$$

Then, using an argument as in the proof of Lemma B.1.2, we can show that

$$\mathrm{FN}(c\rho?(r_1\delta_2\rho\colon T_1, \ldots r_n\delta_2\rho\colon T_n)\ p'_2\delta_2\rho) \pitchfork \mathrm{Change}(\rho^\alpha)$$

which will allow us to use ρ^α in CHNG-RCV.

So,

$$\begin{array}{ll} & p'_2\delta_2\rho\rho^\alpha \\ \equiv_\alpha & \langle \text{Lemma B.1.2} \rangle \\ & p'_2(\rho \circ \delta_1 \circ \delta_2 \circ \rho^{-1} \circ \rho \circ \delta_2) \\ = & \langle (\delta_2 \circ \rho^{-1} \circ \rho \circ \delta_2) = 1 \rangle \\ & p'_2(\rho \circ \delta_1) \\ \equiv_\alpha & \langle \text{induction hypothesis} \rangle \\ & p'_1(\rho \circ \delta_1) \\ \equiv_\alpha & \langle \text{Lemma B.1.2} \rangle \\ & p'_1\delta_1\rho \end{array}$$

and similarly

$$r_i\delta_2\rho\rho^\alpha = r_i\delta_1\rho$$

for each i. Finally, we use ρ^α,

$$\begin{array}{ll} & c\rho?(r_1\delta_2\rho\colon T_1, \ldots r_n\delta_2\rho\colon T_n)\ p'_2\delta_2\rho \\ \equiv_\alpha & \langle \text{CHNG-RCV} \rangle \\ & c\rho?(r_1\delta_2\rho\rho^\alpha\colon T_1, \ldots r_n\delta_2\rho\rho^\alpha\colon T_n)\ p'_2\delta_2\rho\rho^\alpha \\ \equiv_\alpha & \langle \text{from above, with CONG-RCV} \rangle \end{array}$$

$$
\begin{array}{ll}
& c\rho?(r_1\delta_2\rho\rho^\alpha{:}\,T_1, \ldots r_n\delta_2\rho\rho^\alpha{:}\,T_n)\ p'_1\delta_1\rho \\
= & \langle r_i\delta_2\rho\rho^\alpha = r_i\delta_1\rho \text{ for each } i\rangle \\
& c\rho?(r_1\delta_1\rho{:}\,T_1, \ldots r_n\delta_1\rho{:}\,T_n)\ p'_1\delta_1\rho
\end{array}
$$

as required.

Say that the proof tree is an instance of CHNG-ACC:

$$c?r\ p \equiv_\alpha c?r\rho'\ p\rho'$$

where $\mathrm{FN}(c?r\ p) \pitchfork \mathrm{Change}(\rho')$. Now

$$
\begin{array}{ll}
& (c?r\ p)\rho \\
= & \langle \text{renaming procedure, choosing } \delta_1 \text{ appropriately}\rangle \\
& c\rho?r\delta_1\rho\ p\delta_1\rho
\end{array}
$$

and

$$
\begin{array}{ll}
& (c?r\rho'\ p\rho')\rho \\
= & \langle \text{renaming procedure, choosing } \delta_2 \text{ appropriately}\rangle \\
& c\rho?r\rho'\delta_2\rho\ p\rho'\delta_2\rho
\end{array}
$$

Use a ρ^α-construction with

$$\rho^\alpha = \rho \circ \delta_1 \circ {\rho'}^{-1} \circ \delta_2 \circ \rho^{-1}$$

to show

$$c\rho?r\delta_1\rho\ p\delta_1\rho \equiv_\alpha c\rho?r\rho'\delta_2\rho\ p\rho'\delta_2\rho$$

as required. ■

The following technical lemma is needed for the proofs of lemmas B.2.1 and B.3.9.

Lemma B.1.4 *Say* $\text{Change}(\rho) \pitchfork \text{FN}(p)$. *Then* $p\rho \equiv_\alpha p$.

Proof: We use an induction on the structure of p. The only non-obvious cases are the ones involving binders. These are illustrated by:

$$
\begin{array}{ll}
 & (\texttt{new}\ c{:}\,T\ p')\rho \\
= & \langle \text{renaming procedure, choosing } \delta \text{ appropriately} \rangle \\
 & \texttt{new}\ c\delta\rho{:}\,T\ p'\delta\rho \\
\equiv_\alpha & \langle \text{Lemma B.1.2 and CONG-NEW} \rangle \\
 & \texttt{new}\ c\delta\rho{:}\,T\ p'(\rho \circ \delta) \\
= & \langle \text{composition} \rangle \\
 & \texttt{new}\ c(\rho \circ \delta){:}\,T\ p'(\rho \circ \delta) \\
\equiv_\alpha & \langle \text{CHNG-NEW} \rangle \\
 & \texttt{new}\ c{:}\,T\ p'
\end{array}
$$

The last step is since

$$\text{Change}(\rho \circ \delta) \subseteq (\text{Change}(\rho) \cup \text{Change}(\delta)) \pitchfork \text{FN}(\texttt{new}\ c{:}\,T\ p')$$

■

B.2 Results about Agents

Most results about agents stated in the body of the thesis follow directly from the corresponding results for code, so we consider their proofs unnecessary. Instead, this section proves some results about agents which are needed later in this appendix.

Lemma B.2.1 *Say* $\mathrm{FN}(g) \subseteq A$ *and* $\rho_1 \simeq_A \rho_2$. *Then*

$$g\rho_1 \equiv_\alpha g\rho_2$$

Proof: Use an induction on the structure of g. The nil and parallel cases are obvious. The case for primitive agents follows from Lemma B.1.4. ■

Corollary B.2.2 *Say* ρ_1 *and* ρ_2 *agree on* $\mathrm{FN}(g)$. *Then* $g\rho_1 \equiv_\alpha g\rho_2$.

The following result shows that α-equivalence and structural equivalence are mutually non-interfering. Lemma B.2.4 proves that an instance of the α-/structural equivalence relation can be split into an instance of α-equivalence followed by an instance of structural equivalence (or vice-versa). We will need this property for the proofs of lemmas B.4.7–B.4.10.

Lemma B.2.3

$$\begin{array}{lllllll} & g_1 & \equiv_\alpha & g & \stackrel{\triangleright}{\equiv} & g_2 & \textit{some } g \\ \Leftrightarrow & g_1 & \stackrel{\triangleright}{\equiv} & g' & \equiv_\alpha & g_2 & \textit{some } g' \end{array}$$

Proof: ($\Rightarrow$) We can use the proof tree of $g_1 \equiv_\alpha g$ to give a function $f_{\equiv_\alpha}$ which takes any agent with the same set of agents as g_1 to an agent with the same set of agents as g in an obvious way. It is easy to show that for any agent g_0 in its domain, we have $g_0 \equiv_\alpha f_{\equiv_\alpha}(g_0)$.

Similarly, we can use the proof tree of $g \stackrel{\triangleright}{\equiv} g_2$ to give a function which takes any agent with the same structure as g to an agent with the structure of g_2. It is easy to show that for any agent g_0 in its domain, we have $g_0 \stackrel{\triangleright}{\equiv} f_{\stackrel{\triangleright}{\equiv}}(g_0)$.

It can be seen that $f_{\equiv_\alpha}(f_{\stackrel{\triangleright}{\equiv}}(g_1)) = g_2$. Use $g' = f_{\stackrel{\triangleright}{\equiv}}(g_1)$ as the required intermediate agent. The case for ($\Leftarrow$) is similar. ■

Lemma B.2.4 *For two agents g_1 and g_2 such that $g_1 \overset{\triangleright}{\equiv}_\alpha g_2$, there exists agents g and g' such that*

$$\begin{array}{ccccc} g_1 & \equiv_\alpha & g & \overset{\triangleright}{\equiv} & g_2 \\ g_1 & \overset{\triangleright}{\equiv} & g' & \equiv_\alpha & g_2 \end{array}$$

Proof: We use induction on the proof of $g_1 \overset{\triangleright}{\equiv}_\alpha g_2$. The only interesting case is where the last step of the proof is EQV-TRN:

$$\frac{g_1 \overset{\triangleright}{\equiv}_\alpha g_2 \qquad g_2 \overset{\triangleright}{\equiv}_\alpha g_3}{g_1 \overset{\triangleright}{\equiv}_\alpha g_3}$$

By the induction hypothesis, there must exist agents g_1', g_1'', g_2' and g_2'' such that

$$\begin{array}{ccccccccc} g_1 & \equiv_\alpha & g_1' & \overset{\triangleright}{\equiv} & g_2 & \equiv_\alpha & g_2' & \overset{\triangleright}{\equiv} & g_3 \\ g_1 & \overset{\triangleright}{\equiv} & g_1'' & \equiv_\alpha & g_2 & \overset{\triangleright}{\equiv} & g_2'' & \equiv_\alpha & g_3 \end{array}$$

Now, by agent-$\overset{\triangleright}{\equiv}$-EQV-TRN, we have that $g_1 \equiv_\alpha g_1' \overset{\triangleright}{\equiv} g_2''$. We can use Lemma B.2.3 to give us $g_1 \overset{\triangleright}{\equiv} g \equiv_\alpha g_2''$ for some g. We can then use agent-$\equiv_\alpha$-EQV-TRN to give us that $g_1 \overset{\triangleright}{\equiv} g \equiv_\alpha g_3$ as required.

Similarly, by agent-$\equiv_\alpha$-EQV-TRN we have $g_1 \overset{\triangleright}{\equiv} g_1'' \equiv_\alpha g_2'$. Lemma B.2.3 gives us $g_1 \equiv_\alpha g' \overset{\triangleright}{\equiv} g_2'$ for some g'. Use agent-$\overset{\triangleright}{\equiv}$-EQV-TRN to give $g_1 \overset{\triangleright}{\equiv} g' \equiv_\alpha g_3$ as required. ■

B.3 Results about Configurations

This section proves a number of results about configurations, most of which were stated in the body of the thesis. The section is divided into three subsections. Section B.3.1 gives properties of configurations under renaming. Section B.3.2 gives properties of the α-equivalence system for configurations. Lastly, Section B.3.3 gives properties of the structural equivalence system for configurations.

B.3.1 Configuration Renaming

This subsection gives the proofs of lemmas 4.2.2–4.2.4.

4.2.2 p101

Lemma B.3.1 *For a configuration K and renaming ρ,*

$$\mathrm{FN}(K\rho) = \mathrm{FN}(K)\rho$$

Proof: This is similar to the proof of Lemma 3.4.5 on page 72. We use induction on construction of K. The only interesting case occurs when $K = K'\{^{\Phi}_{\Delta}$ and this can be tackled like the case given for that lemma. ■

4.2.3 p101

Lemma B.3.2 *For a configuration K and a renaming ρ,*

$$\mathrm{STATE}(K\rho) = \mathrm{STATE}(K)\rho$$

Proof: We use induction on the structure of K. If K is either Nil or $o[p]$ then we are done, as the state will be empty.

Say that $K = (K_1|K_2)$, then

$$
\begin{array}{ll}
 & \mathrm{STATE}((K_1|K_2)\rho) \\
= & \quad \langle \text{renaming procedure} \rangle \\
 & \mathrm{STATE}(K_1\rho|K_2\rho) \\
= & \quad \langle \text{definition of } \mathrm{STATE}(\cdot) \rangle \\
 & \mathrm{STATE}(K_1\rho) \cup \mathrm{STATE}(K_2\rho) \\
= & \quad \langle \text{induction hypothesis} \rangle \\
 & \mathrm{STATE}(K_1)\rho \cup \mathrm{STATE}(K_2)\rho \\
= & \quad \langle \rho \text{ is bijective} \rangle \\
 & (\mathrm{STATE}(K_1) \cup \mathrm{STATE}(K_2))\rho \\
= & \quad \langle \text{definition of } \mathrm{STATE}(\cdot) \rangle \\
 & \mathrm{STATE}(K_1|K_2)\rho
\end{array}
$$

Say that $K = K'\{^{\Phi}_{\Delta}$, then

$$\begin{array}{ll} & \text{STATE}((K'\{^{\Phi}_{\Delta})\rho) \\ = & \langle\text{renaming procedure, choosing } \delta \text{ appropriately}\rangle \\ & \text{STATE}(K'\delta\rho\{^{\Phi\delta\rho}_{\Delta\delta\rho}) \\ = & \langle\text{definition of STATE}(\cdot)\rangle \\ & (\text{STATE}(K'\delta\rho) \cup \text{Dom}(\Delta\delta\rho)) \setminus \text{Dom}(\Phi\delta\rho) \\ = & \langle\text{induction hypothesis and set theory}\rangle \\ & (\text{STATE}(K'\delta)\rho \cup \text{Dom}(\Delta\delta)\rho) \setminus \text{Dom}(\Phi\delta)\rho \\ = & \langle\text{induction hypothesis and set theory}\rangle \\ & (\text{STATE}(K')\delta\rho \cup \text{Dom}(\Delta)\delta\rho) \setminus \text{Dom}(\Phi)\delta\rho \\ = & \langle\rho \text{ and } \delta \text{ are bijective}\rangle \\ & ((\text{STATE}(K') \cup \text{Dom}(\Delta)) \setminus \text{Dom}(\Phi))\delta\rho \\ = & \langle\text{choice of } \delta\rangle \\ & ((\text{STATE}(K') \cup \text{Dom}(\Delta)) \setminus \text{Dom}(\Phi))\rho \\ = & \langle\text{definition of STATE}(\cdot)\rangle \\ & \text{STATE}(K'\{^{\Phi}_{\Delta})\rho \end{array}$$

■

4.2.4 p102

Lemma B.3.3 *For a configuration K and a renaming ρ,*

$$\text{WF}(K\rho) \Leftrightarrow \text{WF}(K)$$

Proof: We use induction on the construction of K. If K is either Nil or $o[p]$, then both K and $K\rho$ will be well-formed.

Say that $K = (K_1|K_2)$. Then

$$\begin{array}{ll} & \text{WF}((K_1|K_2)\rho) \\ \Leftrightarrow & \langle\text{renaming procedure}\rangle \\ & \text{WF}(K_1\rho|K_2\rho) \end{array}$$

$\Leftrightarrow$ $\langle$definition of WF$(\cdot)\rangle$

$\mathrm{WF}(K_1\rho) \wedge \mathrm{WF}(K_1\rho) \wedge (\mathrm{STATE}(K_1\rho) \pitchfork \mathrm{STATE}(K_2\rho))$

$\Leftrightarrow$ $\langle$induction hypothesis twice and Lemma B.3.2 twice$\rangle$

$\mathrm{WF}(K_1) \wedge \mathrm{WF}(K_1) \wedge (\mathrm{STATE}(K_1)\rho \pitchfork \mathrm{STATE}(K_2)\rho)$

$\Leftrightarrow$ $\langle\rho$ is bijective$\rangle$

$\mathrm{WF}(K_1) \wedge \mathrm{WF}(K_1) \wedge (\mathrm{STATE}(K_1) \pitchfork \mathrm{STATE}(K_2))$

$\Leftrightarrow$ $\langle$definition of WF$(\cdot)\rangle$

$\mathrm{WF}(K_1|K_2)$

Say that $K = K'\{^{\Phi}_{\Delta}$. Then

$\mathrm{WF}((K'\{^{\Phi}_{\Delta})\rho)$

$\Leftrightarrow$ $\langle$renaming procedure, choosing δ appropriately$\rangle$

$\mathrm{WF}(K'\delta\rho\{^{\Phi\delta\rho}_{\Delta\delta\rho})$

$\Leftrightarrow$ $\langle$definition of WF$(\cdot)\rangle$

$\mathrm{WF}(K'\delta\rho)$

$\wedge(\mathrm{STATE}(K'\delta\rho) \pitchfork \mathrm{Dom}(\Delta\delta\rho))$

$\wedge(\mathrm{OBJECTS}(\Phi\delta\rho) \subseteq (\mathrm{STATE}(K'\delta\rho) \cup \mathrm{Dom}(\Delta\delta\rho)))$

$\Leftrightarrow$ $\langle$induction hypothesis twice, Lemma B.3.2 twice and set theory$\rangle$

$\mathrm{WF}(K')$

$\wedge(\mathrm{STATE}(K')\delta\rho \pitchfork \mathrm{Dom}(\Delta)\delta\rho)$

$\wedge(\mathrm{OBJECTS}(\Phi)\delta\rho \subseteq (\mathrm{STATE}(K')\delta\rho \cup \mathrm{Dom}(\Delta)\delta\rho))$

$\Leftrightarrow$ $\langle\rho$ and δ are bijective$\rangle$

$\mathrm{WF}(K')$

$\wedge(\mathrm{STATE}(K') \pitchfork \mathrm{Dom}(\Delta))$

$\wedge(\mathrm{OBJECTS}(\Phi)\delta\rho \subseteq (\mathrm{STATE}(K') \cup \mathrm{Dom}(\Delta))\delta\rho)$

$\Leftrightarrow$ $\langle\rho$ and δ are bijective$\rangle$

$\mathrm{WF}(K')$

$\wedge(\mathrm{STATE}(K') \pitchfork \mathrm{Dom}(\Delta))$

$\wedge\text{OBJECTS}(\Phi) \subseteq (\text{STATE}(K') \cup \text{Dom}(\Delta))$

$\Leftrightarrow$ $\langle$definition of $\text{WF}(\cdot)\rangle$

$\text{WF}(K'\{^{\Phi}_{\Delta})$

■

B.3.2 α-Equivalence for Configurations

This subsection gives the proofs of lemmas 4.2.5–4.2.9.

4.2.5 p102

Lemma B.3.4 *For two configurations K_1 and K_2,*

$$K_1 \equiv_\alpha K_2 \Rightarrow \text{FN}(K_1) = \text{FN}(K_2)$$

Proof: Use an induction on the proof tree of $K_1 \equiv_\alpha K_2$. The approach is similar to Lemma B.1.1. ■

4.2.6 p102

Lemma B.3.5 *For two configurations K_1 and K_2,*

$$K_1 \equiv_\alpha K_2 \Rightarrow \text{STATE}(K_1) = \text{STATE}(K_2)$$

Proof: We use induction on the structure of the proof tree of $K_1 \equiv_\alpha K_2$. The only interesting case is where the proof tree is an instance of the CHNG-SCP:

$$K\{^{\Phi}_{\Delta} \equiv_\alpha K\rho\{^{\Phi\rho}_{\Delta\rho}$$

Then

$\text{STATE}(K\rho\{^{\Delta\rho}_{\Phi\rho})$

$=$ $\langle$definition of $\text{STATE}(\cdot)\rangle$

$(\text{STATE}(K\rho) \cup \text{Dom}(\Delta\rho)) \setminus \text{Dom}(\Phi\rho)$

$=$ $\langle$Lemma B.3.2 and set theory$\rangle$

$$
\begin{array}{ll}
& (\mathrm{STATE}(K)\rho \cup \mathrm{Dom}(\Delta)\rho) \setminus \mathrm{Dom}(\Phi)\rho \\
= & \quad \langle \rho \text{ is bijective} \rangle \\
& (\mathrm{STATE}(K) \cup \mathrm{Dom}(\Delta))\rho \setminus \mathrm{Dom}(\Phi)\rho \\
= & \quad \langle \rho \text{ is bijective} \rangle \\
& ((\mathrm{STATE}(K) \cup \mathrm{Dom}(\Delta)) \setminus \mathrm{Dom}(\Phi))\rho \\
= & \quad \langle \text{choice of } \rho \text{ and Lemma 4.2.1} \rangle \\
& (\mathrm{STATE}(K) \cup \mathrm{Dom}(\Delta)) \setminus \mathrm{Dom}(\Phi) \\
= & \quad \langle \text{definition of STATE}(\cdot) \rangle \\
& \mathrm{STATE}(K\{^{\Phi}_{\Delta})
\end{array}
$$

The second last step depends on the fact that $\mathrm{FN}(K\{^{\Phi}_{\Delta}) \pitchfork \mathrm{Change}(\rho)$ ■

4.2.7 p104

Lemma B.3.6 *For two configurations K_1 and K_2,*

$$K_1 \equiv_\alpha K_2 \Rightarrow \mathrm{WF}(K_1) \Leftrightarrow \mathrm{WF}(K_2)$$

Proof: We use induction on the structure of the proof tree of $K_1 \equiv_\alpha K_2$. Again, the only interesting case is where the tree is an instance of CHNG-SCP:

$$K\{^{\Phi}_{\Delta} \equiv_\alpha K\rho\{^{\Phi\rho}_{\Delta\rho}$$

Then,

$$
\begin{array}{ll}
& \mathrm{WF}(K\rho\{^{\Phi\rho}_{\Delta\rho}) \\
\Leftrightarrow & \quad \langle \text{definition of WF}(\cdot) \rangle \\
& \mathrm{WF}(K\rho) \\
& \wedge(\mathrm{STATE}(K\rho) \pitchfork \mathrm{Dom}(\Delta\rho)) \\
& \wedge(\mathrm{OBJECTS}(\Phi\rho) \subseteq (\mathrm{STATE}(K\rho) \cup \mathrm{Dom}(\Delta\rho))) \\
\Leftrightarrow & \quad \langle \text{induction hypothesis, Lemma B.3.2 and set theory} \rangle \\
& \mathrm{WF}(K) \\
& \wedge(\mathrm{STATE}(K)\rho \pitchfork \mathrm{Dom}(\Delta)\rho)
\end{array}
$$

$\wedge(\text{OBJECTS}(\Phi)\rho \subseteq (\text{STATE}(K)\rho \cup \text{Dom}(\Delta)\rho))$

$\Leftrightarrow \quad \langle \rho \text{ is bijective} \rangle$

$\text{WF}(K)$

$\wedge(\text{STATE}(K) \pitchfork \text{Dom}(\Delta))$

$\wedge(\text{OBJECTS}(\Phi)\rho \subseteq (\text{STATE}(K) \cup \text{Dom}(\Delta))\rho)$

$\Leftrightarrow \quad \langle \rho \text{ is bijective} \rangle$

$\text{WF}(K)$

$\wedge(\text{STATE}(K) \pitchfork \text{Dom}(\Delta))$

$\wedge(\text{OBJECTS}(\Phi) \subseteq (\text{STATE}(K) \cup \text{Dom}(\Delta)))$

$\Leftrightarrow \quad \langle \text{definition of WF}(\cdot) \rangle$

$\text{WF}(K\{^{\Phi}_{\Delta})$

■

4.2.8 p104

Lemma B.3.7 *For a configuration K and two renamings ρ_1 and ρ_2,*

$$K\rho_1\rho_2 \equiv_\alpha K(\rho_2 \circ \rho_1)$$

Proof: We use an induction on the structure of the configuration K. If $K = \text{Nil}$, then the result is obvious. If $K = o[p]$, then the result follows in a straightforward fashion from Lemma B.1.2.

Say that $K = (K_1|K_2)$. Then

$(K_1|K_2)\rho_1\rho_2$

$= \quad \langle \text{renaming procedure} \rangle$

$(K_1\rho_1|K_2\rho_1)\rho_2$

$= \quad \langle \text{renaming procedure} \rangle$

$(K_1\rho_1\rho_2|K_2\rho_1\rho_2)$

$\equiv_\alpha \quad \langle \text{induction hypothesis twice and CONG-PAR} \rangle$

$(K_1(\rho_2 \circ \rho_1)|K_2(\rho_2 \circ \rho_1))$

$= \quad \langle\text{renaming procedure}\rangle$

$\quad (K_1|K_2)(\rho_2 \circ \rho_1)$

as required in this case.

Consider the case where $K = K'\{^{\Phi}_{\Delta}$. Then

$\quad (K'\{^{\Phi}_{\Delta})\rho_1\rho_2$

$= \quad \langle\text{renaming procedure, choosing } \delta_1 \text{ appropriately}\rangle$

$\quad (K'\delta_1\rho_1\{^{\Phi\delta_1\rho_1}_{\Delta\delta_1\rho_1})\rho_2$

$= \quad \langle\text{renaming procedure, choosing } \delta_2 \text{ appropriately}\rangle$

$\quad K'\delta_1\rho_1\delta_2\rho_2\{^{\Phi\delta_1\rho_1\delta_2\rho_2}_{\Delta\delta_1\rho_1\delta_2\rho_2}$

and

$\quad (K'\{^{\Phi}_{\Delta})(\rho_2 \circ \rho_1)$

$= \quad \langle\text{renaming procedure, choosing } \delta \text{ appropriately}\rangle$

$\quad K'\delta(\rho_2 \circ \rho_1)\{^{\Phi\delta(\rho_2\circ\rho_1)}_{\Delta\delta(\rho_2\circ\rho_1)}$

Use a ρ^α-construction with

$$\rho^\alpha = \rho_2 \circ \delta_2 \circ \rho_1 \circ \delta_1 \circ \delta \circ {\rho_1}^{-1} \circ {\rho_2}^{-1}$$

Then,

$\quad K'\delta(\rho_2 \circ \rho_1)\rho^\alpha$

$\equiv_\alpha \quad \langle\text{induction hypothesis twice}\rangle$

$\quad K'(\rho^\alpha \circ \rho_2 \circ \rho_1 \circ \delta)$

$= \quad \langle\text{expand } \rho^\alpha\rangle$

$\quad K'(\rho_2 \circ \delta_2 \circ \rho_1 \circ \delta_1 \circ \delta \circ {\rho_1}^{-1} \circ {\rho_2}^{-1} \circ \rho_2 \circ \rho_1 \circ \delta)$

$= \quad \langle(\delta \circ {\rho_1}^{-1} \circ {\rho_2}^{-1} \circ \rho_2 \circ \rho_1 \circ \delta) = 1\rangle$

$\quad K'(\rho_2 \circ \delta_2 \circ \rho_1 \circ \delta_1)$

$\equiv_\alpha \quad \langle\text{induction hypothesis three times}\rangle$

$K'\delta_1\rho_1\delta_2\rho_2$

Similarly

$$\begin{aligned}\Phi\delta(\rho_2 \circ \rho_1)\rho^\alpha &= \Phi\delta_1\rho_1\delta_2\rho_2 \\ \Delta\delta(\rho_2 \circ \rho_1)\rho^\alpha &= \Delta\delta_1\rho_1\delta_2\rho_2\end{aligned}$$

Using an argument exactly as in Lemma B.1.2, we can show that

$$\mathrm{FN}(K'\delta(\rho_2 \circ \rho_1)\{^{\Phi\delta(\rho_2\circ\rho_1)}_{\Delta\delta(\rho_2\circ\rho_1)}) \pitchfork \mathrm{Change}(\rho^\alpha)$$

Thus,

$$\begin{aligned}
& K'\delta(\rho_2 \circ \rho_1)\{^{\Phi\delta(\rho_2\circ\rho_1)}_{\Delta\delta(\rho_2\circ\rho_1)} \\
\equiv_\alpha \quad & \langle \text{CHNG-SCP} \rangle \\
& K'\delta(\rho_2 \circ \rho_1)\rho^\alpha\{^{\Phi\delta(\rho_2\circ\rho_1)\rho^\alpha}_{\Delta\delta(\rho_2\circ\rho_1)\rho^\alpha} \\
\equiv_\alpha \quad & \langle \text{from above, using CONG-SCP} \rangle \\
& K'\delta_1\rho_1\delta_2\rho_2\{^{\Phi\delta(\rho_2\circ\rho_1)\rho^\alpha}_{\Delta\delta(\rho_2\circ\rho_1)\rho^\alpha} \\
= \quad & \langle \text{from above, the dictionaries are equal} \rangle \\
& K'\delta_1\rho_1\delta_2\rho_2\{^{\Phi\delta_1\rho_1\delta_2\rho_2}_{\Delta\delta_1\rho_1\delta_2\rho_2}
\end{aligned}$$

as required. ■

4.2.9
p104

Lemma B.3.8 *For two configurations K_1 and K_2, and a renaming ρ*

$$K_1 \equiv_\alpha K_2 \Rightarrow K_1\rho \equiv_\alpha K_1\rho$$

Proof: We use an induction on the proof tree of $K_1 \equiv_\alpha K_2$. There is only one interesting case, where the proof tree is an instance of CHNG-SCP:

$$K\{^\Phi_\Delta \equiv_\alpha K\rho'\{^{\Phi\rho'}_{\Delta\rho'}$$

Now

$$(K\{^{\Phi}_{\Delta})\rho = K\delta_1\rho\{^{\Phi\delta_1\rho}_{\Delta\delta_1\rho}$$

and

$$(K\rho'\{^{\Phi\rho'}_{\Delta\rho'})\rho = K\rho'\delta_2\rho\{^{\Phi\rho'\delta_2\rho}_{\Delta\rho'\delta_2\rho}$$

Use a ρ^{α}-construction with

$$\rho^{\alpha} = \rho \circ \delta_1 \circ \rho'^{-1} \circ \delta_2 \circ \rho^{-1}$$

An argument similar to previous proofs, using ρ^{α}, will show that

$$K\delta_1\rho\{^{\Phi\delta_1\rho}_{\Delta\delta_1\rho} \equiv_{\alpha} K\rho'\delta_2\rho\{^{\Phi\rho'\delta_2\rho}_{\Delta\rho'\delta_2\rho}$$

as required. ■

The following lemmas and corollaries justify many of the reasoning steps used in the rest of this appendix.

Lemma B.3.9 *Say* Change(ρ) $\pitchfork$ FN(K). *Then* $K\rho \equiv_{\alpha} K$.

Proof: We use an induction on the structure of K. The cases for nil and parallel configurations are obvious. The case for a primitive configuration follows from Lemma B.1.4. We give the case for when $K = K'\{^{\Phi}_{\Delta}$.

$(K'\{^{\Phi}_{\Delta})\rho$

$=$ ⟨renaming procedure, choosing δ appropriately⟩

$K'\delta\rho\{^{\Phi\delta\rho}_{\Delta\delta\rho}$

$\equiv_{\alpha}$ ⟨Lemma B.3.7 and CONG-SCP⟩

$K'(\rho \circ \delta)\{^{\Phi\delta\rho}_{\Delta\delta\rho}$

$=$ ⟨dictionaries are equal⟩

$K'(\rho \circ \delta)\{^{\Phi(\rho\circ\delta)}_{\Delta(\rho\circ\delta)}$

$\equiv_{\alpha}$ ⟨CHNG-SCP⟩

$K'\{^{\Phi}_{\Delta}$

The last step is since

$$\mathrm{Change}(\rho \circ \delta) \subseteq (\mathrm{Change}(\rho) \cup \mathrm{Change}(\delta)) \pitchfork \mathrm{FN}(K'\{^{\Phi}_{\Delta})$$

■

Corollary B.3.10 *Say ρ_1 and ρ_2 agree on* $\mathrm{FN}(K)$. *Then $K\rho_1 \equiv_\alpha K\rho_2$.*

Corollary B.3.11 *Say* $\mathrm{FN}(K) \subseteq A \cup B$ *and* $\rho_1|_A + \rho_2|_B \simeq \rho$. *Then*

$$K(\rho_1|_A + \rho_2|_B) \equiv_\alpha K\rho$$

In the following lemma, we use the fact that agent configurations are syntactically valid configurations.

Lemma B.3.12 *Say $g_1\{^{\Phi_1}_{\Delta_1} \equiv_\alpha g_2\{^{\Phi_2}_{\Delta_2}$. Then, for some renaming ρ such that* $\mathrm{FN}(g_2\{^{\Phi_2}_{\Delta_2}) \pitchfork \mathrm{Change}(\rho)$, *we have*

$$\Phi_2\rho = \Phi_1 \qquad \Delta_2\rho = \Delta_1 \qquad g_2\rho \equiv_\alpha g_1$$

Proof: We use an induction on the proof tree of $g_1\{^{\Phi_1}_{\Delta_1} \equiv_\alpha g_2\{^{\Phi_2}_{\Delta_2}$. When the proof tree is an instance of the EQV-RFL rule, the identity renaming works.

Say that the last step of the proof tree is an instance of EQV-SYM:

$$\frac{g_2\{^{\Phi_2}_{\Delta_2} \equiv_\alpha g_1\{^{\Phi_1}_{\Delta_1}}{g_1\{^{\Phi_1}_{\Delta_1} \equiv_\alpha g_2\{^{\Phi_2}_{\Delta_2}}$$

By the induction hypothesis, there exists some ρ such that $\mathrm{FN}(g_1\{^{\Phi_1}_{\Delta_1}) \pitchfork \mathrm{Change}(\rho)$ and

$$\Phi_1\rho = \Phi_2 \qquad \Delta_1\rho = \Delta_2 \qquad g_1\rho \equiv_\alpha g_2$$

By Lemma B.3.4, we know $\mathrm{FN}(g_1\{^{\Phi_1}_{\Delta_1}) = \mathrm{FN}(g_2\{^{\Phi_2}_{\Delta_2})$ and, by Lemma 3.4.1, we know $\mathrm{Change}(\rho) = \mathrm{Change}(\rho^{-1})$ so

$$\mathrm{FN}(g_2\{^{\Phi_2}_{\Delta_2}) \pitchfork \mathrm{Change}(\rho^{-1})$$

Also

$$\begin{aligned} \Phi_2\rho^{-1} &= (\Phi_1\rho)\rho^{-1} &= \Phi_1 \\ \Delta_2\rho^{-1} &= (\Delta_1\rho)\rho^{-1} &= \Delta_1 \end{aligned}$$

and

$$g_2\rho^{-1} \equiv_\alpha (g_1\rho)\rho^{-1} \equiv_\alpha g_1(\rho^{-1} \circ \rho) = g_1$$

Therefore we can use ρ^{-1} for this case.

Say the last step of the proof tree is an instance of EQV-TRN. Then, since $\equiv_\alpha$ preserves structure, the tree must end as follows:

$$\frac{g_1\{^{\Phi_1}_{\Delta_1} \equiv_\alpha g\{^{\Phi}_{\Delta} \qquad g\{^{\Phi}_{\Delta} \equiv_\alpha g_2\{^{\Phi_2}_{\Delta_2}}{g_1\{^{\Phi_1}_{\Delta_1} \equiv_\alpha g_2\{^{\Phi_2}_{\Delta_2}}$$

By the induction hypothesis, there must be renamings ρ_1 and ρ_2 such that

$$\begin{array}{lllllllll} \Phi\rho_1 &=& \Phi_1 & \Delta\rho_1 &=& \Delta_1 & g\rho_1 &\equiv_\alpha& g_1 \\ \Phi_2\rho_2 &=& \Phi & \Delta_2\rho_2 &=& \Delta & g_2\rho_2 &\equiv_\alpha& g \end{array}$$

where $\mathrm{FN}(g\{^{\Phi}_{\Delta}) \pitchfork \mathrm{Change}(\rho_1)$ and $\mathrm{FN}(g_2\{^{\Phi_2}_{\Delta_2}) \pitchfork \mathrm{Change}(\rho_2)$. Now, by Lemma B.3.4, $\mathrm{FN}(g_2\{^{\Phi_2}_{\Delta_2}) = \mathrm{FN}(g\{^{\Phi}_{\Delta})$ so

$$\mathrm{FN}(g_2\{^{\Phi_2}_{\Delta_2}) \pitchfork (\mathrm{Change}(\rho_1) \cup \mathrm{Change}(\rho_2))$$

Thus

$$\mathrm{FN}(g_2\{^{\Phi_2}_{\Delta_2}) \pitchfork \mathrm{Change}(\rho_1 \circ \rho_2)$$

Also,

$$\begin{array}{lllllll} \Phi_2(\rho_1 \circ \rho_2) &=& (\Phi_2\rho_2)\rho_1 &=& \Phi\rho_1 &=& \Phi_1 \\ \Delta_2(\rho_1 \circ \rho_2) &=& (\Delta_2\rho_2)\rho_1 &=& \Delta\rho_1 &=& \Delta_1 \\ g_2(\rho_1 \circ \rho_2) &\equiv_\alpha& (g_2\rho_2)\rho_1 &\equiv_\alpha& g\rho_1 &\equiv_\alpha& g_1 \end{array}$$

so $\rho_1 \circ \rho_2$ can be our required renaming.

Say the last step of the proof tree is an instance of CONG-SCP. Then the tree must end as follows:

$$\frac{g_1 \equiv_\alpha g_2}{g_1\{^{\Phi_1}_{\Delta_1} \equiv_\alpha g_2\{^{\Phi_2}_{\Delta_2}}$$

where $\Phi_1 = \Phi_2$ and $\Delta_1 = \Delta_2$. The identity renaming will work in this case.

Consider the case where the proof tree is an instance of CHNG-SCP:

$$g_1\{^{\Phi_1}_{\Delta_1} \equiv_\alpha g_1\rho\{^{\Phi_1\rho}_{\Delta_1\rho}$$

We know that $\mathrm{FN}(g_1\{^{\Phi_1}_{\Delta_1}) \pitchfork \mathrm{Change}(\rho)$, so by reasoning similar to the case for symmetry,

$$\mathrm{FN}(g_2\rho\{^{\Phi_2\rho}_{\Delta_2\rho}) \pitchfork \mathrm{Change}(\rho^{-1})$$

and

$$\begin{array}{rcl} \Phi_1\rho\rho^{-1} & = & \Phi_1 \\ \Delta_1\rho\rho^{-1} & = & \Delta_1 \\ g_1\rho\rho^{-1} & \equiv_\alpha & g_1 \end{array}$$

Thus, ρ^{-1} works for this case. No other cases fit, so we are done. ■

B.3.3 Structural Equivalence for Configurations

This subsection gives the proofs for lemmas 4.2.10–4.2.13.

4.2.10 p106

Lemma B.3.13 *For two configurations K_1 and K_2,*

$$K_1 \overset{\blacktriangleright}{\equiv} K_2 \Rightarrow \mathrm{FN}(K_1) = \mathrm{FN}(K_2)$$

Proof: We use an induction on the proof tree of $K_1 \overset{\blacktriangleright}{\equiv} K_2$. The two interesting cases are FLATTEN and EXTRUDE.

Say the proof tree is an instance of FLATTEN:

$$K\{^{\Phi_1}_{\Delta_1}\{^{\Phi_2}_{\Delta_2} \overset{\blacktriangleright}{\equiv} K\{^{\Phi_1\cup\Phi_2}_{\Delta_1\cup\Delta_2}$$

Well,

$$
\begin{array}{cl}
 & \mathrm{FN}(K\{^{\Phi_1}_{\Delta_1}\{^{\Phi_2}_{\Delta_2}) \\
= & \quad \langle \text{definition of } \mathrm{FN}(\cdot) \rangle \\
 & (\mathrm{FN}(K\{^{\Phi_1}_{\Delta_1}) \cup \mathrm{FN}(\Delta_2)) \smallsetminus \mathrm{Dom}(\Phi_2) \\
= & \quad \langle \text{definition of } \mathrm{FN}(\cdot) \rangle \\
 & (((\mathrm{FN}(K) \cup \mathrm{FN}(\Delta_1)) \smallsetminus \mathrm{Dom}(\Phi_1)) \cup \mathrm{FN}(\Delta_2)) \smallsetminus \mathrm{Dom}(\Phi_2) \\
= & \quad \langle \mathrm{Dom}(\Phi_1) \pitchfork \mathrm{FN}(\Delta_2) \rangle \\
 & ((\mathrm{FN}(K) \cup \mathrm{FN}(\Delta_1) \cup \mathrm{FN}(\Delta_2)) \smallsetminus \mathrm{Dom}(\Phi_1)) \smallsetminus \mathrm{Dom}(\Phi_2) \\
= & \quad \langle \text{set theory and definitions} \rangle \\
 & (\mathrm{FN}(K) \cup \mathrm{FN}(\Delta_1 \cup \Delta_2)) \smallsetminus \mathrm{Dom}(\Phi_1 \cup \Phi_2) \\
= & \quad \langle \text{definition of } \mathrm{FN}(\cdot) \rangle \\
 & \mathrm{FN}(K\{^{\Delta_1 \cup \Delta_2}_{\Phi_1 \cup \Phi_2})
\end{array}
$$

where the third step is justified by a side condition of the FLATTEN rule.

Say that the proof tree is an instance of EXTRUDE:

$$K_1|(K_2\{^{\Phi}_{\Delta}) \stackrel{\blacktriangleright}{\equiv} (K_1|K_2)\{^{\Phi}_{\Delta}$$

Then,

$$
\begin{array}{cl}
 & \mathrm{FN}(K_1|(K_2\{^{\Phi}_{\Delta})) \\
= & \quad \langle \text{definition of } \mathrm{FN}(\cdot) \rangle \\
 & \mathrm{FN}(K_1) \cup \mathrm{FN}(K_2\{^{\Phi}_{\Delta}) \\
= & \quad \langle \text{definition of } \mathrm{FN}(\cdot) \rangle \\
 & \mathrm{FN}(K_1) \cup ((\mathrm{FN}(K_2) \cup \mathrm{FN}(\Delta)) \smallsetminus \mathrm{Dom}(\Phi)) \\
= & \quad \langle \mathrm{FN}(K_1) \pitchfork \mathrm{Dom}(\Phi) \rangle \\
 & (\mathrm{FN}(K_1) \cup \mathrm{FN}(K_2) \cup \mathrm{FN}(\Delta)) \smallsetminus \mathrm{Dom}(\Phi) \\
= & \quad \langle \text{definition of } \mathrm{FN}(\cdot) \rangle \\
 & (\mathrm{FN}(K_1|K_2) \cup \mathrm{FN}(\Delta)) \smallsetminus \mathrm{Dom}(\Phi)
\end{array}
$$

$$
\begin{aligned}
= \quad & \langle \text{definition of FN}(\cdot) \rangle \\
& \text{FN}((K_1|K_2)\{^{\Phi}_{\Delta})
\end{aligned}
$$

where the third step is justified by the side condition of the EXTRUDE rule.

■

4.2.11 p106

Lemma B.3.14 *For two configurations K_1 and K_2,*

$$K_1 \overset{\blacktriangleright}{\equiv} K_2 \Rightarrow \text{STATE}(K_1) = \text{STATE}(K_2)$$

Proof: We use an induction on the proof tree of $K_1 \overset{\blacktriangleright}{\equiv} K_2$. Again, we only examine the two interesting cases.

Say that the proof tree is an instance of FLATTEN:

$$K\{^{\Phi_1}_{\Delta_1}\{^{\Phi_2}_{\Delta_2} \overset{\blacktriangleright}{\equiv} K\{^{\Phi_1 \cup \Phi_2}_{\Delta_1 \cup \Delta_2}$$

Well,

$$
\begin{aligned}
& \text{STATE}(K\{^{\Phi_1}_{\Delta_1}\{^{\Phi_2}_{\Delta_2}) \\
= \quad & \langle \text{definition of STATE}(\cdot) \rangle \\
& (\text{STATE}(K\{^{\Phi_1}_{\Delta_1}) \cup \text{Dom}(\Delta_2)) \setminus \text{Dom}(\Phi_2) \\
= \quad & \langle \text{definition of STATE}(\cdot) \rangle \\
& (((\text{STATE}(K) \cup \text{Dom}(\Delta_1)) \setminus \text{Dom}(\Phi_1)) \cup \text{Dom}(\Delta_2)) \setminus \text{Dom}(\Phi_2) \\
= \quad & \langle \text{Dom}(\Delta_2) \subseteq \text{FN}(\Delta_2) \text{ and } \text{FN}(\Delta_2) \pitchfork \text{Dom}(\Phi_1) \rangle \\
& ((\text{STATE}(K) \cup \text{Dom}(\Delta_1) \cup \text{Dom}(\Delta_2)) \setminus \text{Dom}(\Phi_1)) \setminus \text{Dom}(\Phi_2) \\
= \quad & \langle \text{set theory} \rangle \\
& (\text{STATE}(K) \cup \text{Dom}(\Delta_1 \cup \Delta_2)) \setminus \text{Dom}(\Phi_1 \cup \Phi_2) \\
= \quad & \langle \text{definition of STATE}(\cdot) \rangle \\
& \text{STATE}(K\{^{\Delta_1 \cup \Delta_2}_{\Phi_1 \cup \Phi_2})
\end{aligned}
$$

Say that the proof tree is an instance of EXTRUDE:

$$K_1|(K_2\{^{\Phi}_{\Delta}) \overset{\blacktriangleright}{\equiv} (K_1|K_2)\{^{\Phi}_{\Delta}$$

Then,

$$
\begin{array}{ll}
 & \mathrm{STATE}(K_1|(K_2\{^{\Phi}_{\Delta})) \\
= & \quad \langle \text{definition of STATE}(\cdot) \rangle \\
 & \mathrm{STATE}(K_1) \cup \mathrm{STATE}(K_2\{^{\Phi}_{\Delta}) \\
= & \quad \langle \text{definition of STATE}(\cdot) \rangle \\
 & \mathrm{STATE}(K_1) \cup ((\mathrm{STATE}(K_2) \cup \mathrm{Dom}(\Delta)) \smallsetminus \mathrm{Dom}(\Phi)) \\
= & \quad \langle \mathrm{STATE}(K_1) \subseteq \mathrm{FN}(K_1) \text{ and } \mathrm{FN}(K_1) \pitchfork \mathrm{Dom}(\Phi) \rangle \\
 & (\mathrm{STATE}(K_1) \cup \mathrm{STATE}(K_2) \cup \mathrm{Dom}(\Delta)) \smallsetminus \mathrm{Dom}(\Phi) \\
= & \quad \langle \text{definition of STATE}(\cdot) \rangle \\
 & (\mathrm{STATE}(K_1|K_2) \cup \mathrm{Dom}(\Delta)) \smallsetminus \mathrm{Dom}(\Phi) \\
= & \quad \langle \text{definition of STATE}(\cdot) \rangle \\
 & \mathrm{STATE}((K_1|K_2)\{^{\Phi}_{\Delta})
\end{array}
$$

where the third step is justified by the side condition of the EXTRUDE rule.

■

4.2.12
p106

Lemma B.3.15 *For two configurations K_1 and K_2,*

$$K_1 \stackrel{\blacktriangleright}{\equiv} K_2 \Rightarrow \mathrm{WF}(K_1) \Leftrightarrow \mathrm{WF}(K_2)$$

Proof: We use induction on the proof tree of $K_1 \stackrel{\blacktriangleright}{\equiv} K_2$. The two interesting cases are FLATTEN and EXTRUDE.

Say that the proof tree is an instance of FLATTEN:

$$K\{^{\Phi_1}_{\Delta_1}\{^{\Phi_2}_{\Delta_2} \stackrel{\blacktriangleright}{\equiv} K\{^{\Phi_1 \cup \Phi_2}_{\Delta_1 \cup \Delta_2}$$

Then

$$
\begin{array}{ll}
 & \mathrm{WF}(K\{^{\Phi_1}_{\Delta_1}\{^{\Phi_2}_{\Delta_2}) \\
\Leftrightarrow & \quad \langle \text{definition of WF}(\cdot) \rangle \\
 & \mathrm{WF}(K\{^{\Phi_1}_{\Delta_1})
\end{array}
$$

$$
\begin{array}{ll}
 & \wedge(\mathrm{STATE}(K\{^{\Phi_1}_{\Delta_1}) \pitchfork \mathrm{Dom}(\Delta_2)) \\
 & \wedge(\mathrm{OBJECTS}(\Phi_2) \subseteq (\mathrm{STATE}(K\{^{\Phi_1}_{\Delta_1}) \cup \mathrm{Dom}(\Delta_2))) \\
\Leftrightarrow & \langle \text{definition of } \mathrm{WF}(\cdot) \rangle \\
 & \mathrm{WF}(K) \\
 & \wedge(\mathrm{STATE}(K) \pitchfork \mathrm{Dom}(\Delta_1)) \\
 & \wedge(\mathrm{OBJECTS}(\Phi_1) \subseteq (\mathrm{STATE}(K) \cup \mathrm{Dom}(\Delta_1))) \\
 & \wedge(\mathrm{STATE}(K\{^{\Phi_1}_{\Delta_1}) \pitchfork \mathrm{Dom}(\Delta_2)) \\
 & \wedge(\mathrm{OBJECTS}(\Phi_2) \subseteq (\mathrm{STATE}(K\{^{\Phi_1}_{\Delta_1}) \cup \mathrm{Dom}(\Delta_2))) \\
\Leftrightarrow & \langle \text{definition of } \mathrm{STATE}(\cdot) \rangle \\
 & \mathrm{WF}(K) \\
 & \wedge(\mathrm{STATE}(K) \pitchfork \mathrm{Dom}(\Delta_1)) \\
 & \wedge(\mathrm{OBJECTS}(\Phi_1) \subseteq (\mathrm{STATE}(K) \cup \mathrm{Dom}(\Delta_1))) \\
 & \wedge(((\mathrm{STATE}(K) \cup \mathrm{Dom}(\Delta_1)) \setminus \mathrm{Dom}(\Phi_1)) \pitchfork \mathrm{Dom}(\Delta_2)) \\
 & \wedge(\mathrm{OBJECTS}(\Phi_2) \subseteq \\
 & \qquad\qquad (((\mathrm{STATE}(K) \cup \mathrm{Dom}(\Delta_1)) \setminus \mathrm{Dom}(\Phi_1)) \cup \mathrm{Dom}(\Delta_2))) \\
\Leftrightarrow & \langle \mathrm{Dom}(\Phi_1) \pitchfork \mathrm{Dom}(\Delta_2) \text{ and } \mathrm{OBJECTS}(\Phi_2) \pitchfork \mathrm{Dom}(\Phi_1) \rangle \\
 & \mathrm{WF}(K) \\
 & \wedge(\mathrm{STATE}(K) \pitchfork \mathrm{Dom}(\Delta_1)) \\
 & \wedge((\mathrm{STATE}(K) \cup \mathrm{Dom}(\Delta_1)) \pitchfork \mathrm{Dom}(\Delta_2)) \\
 & \wedge(\mathrm{OBJECTS}(\Phi_1) \subseteq (\mathrm{STATE}(K) \cup \mathrm{Dom}(\Delta_1))) \\
 & \wedge(\mathrm{OBJECTS}(\Phi_2) \subseteq (\mathrm{STATE}(K) \cup \mathrm{Dom}(\Delta_1) \cup \mathrm{Dom}(\Delta_2))) \\
\Leftrightarrow & \langle \mathrm{Dom}(\Delta_1) \pitchfork \mathrm{Dom}(\Delta_2) \text{ and } \mathrm{OBJECTS}(\Phi_1) \pitchfork \mathrm{Dom}(\Delta_2) \rangle \\
 & \mathrm{WF}(K) \\
 & \wedge(\mathrm{STATE}(K) \pitchfork \mathrm{Dom}(\Delta_1)) \\
 & \wedge(\mathrm{STATE}(K) \pitchfork \mathrm{Dom}(\Delta_2)) \\
 & \wedge(\mathrm{OBJECTS}(\Phi_1) \subseteq (\mathrm{STATE}(K) \cup \mathrm{Dom}(\Delta_1) \cup \mathrm{Dom}(\Delta_2))) \\
 & \wedge(\mathrm{OBJECTS}(\Phi_2) \subseteq (\mathrm{STATE}(K) \cup \mathrm{Dom}(\Delta_1) \cup \mathrm{Dom}(\Delta_2)))
\end{array}
$$

$\Leftrightarrow \quad \langle\text{set theory}\rangle$

$\text{WF}(K)$

$\wedge(\text{STATE}(K) \pitchfork (\text{Dom}(\Delta_1) \cup \text{Dom}(\Delta_2)))$

$\wedge((\text{OBJECTS}(\Phi_1) \cup \text{OBJECTS}(\Phi_2)) \subseteq$

$\qquad (\text{STATE}(K) \cup \text{Dom}(\Delta_1) \cup \text{Dom}(\Delta_2)))$

$\Leftrightarrow \quad \langle\text{set theory}\rangle$

$\text{WF}(K)$

$\wedge(\text{STATE}(K) \pitchfork \text{Dom}(\Delta_1 \cup \Delta_2))$

$\wedge(\text{OBJECTS}(\Phi_1 \cup \Phi_2) \subseteq (\text{STATE}(K) \cup \text{Dom}(\Delta_1 \cup \Delta_2)))$

$\Leftrightarrow \quad \langle\text{definition of WF}(\cdot)\rangle$

$\text{WF}(K\{^{\Phi_1 \cup \Phi_2}_{\Delta_1 \cup \Delta_2})$

Most of the reasoning steps depend on the side conditions of the FLATTEN rule.

Say that the proof tree is an instance of EXTRUDE:

$$K_1|(K_2\{^{\Phi}_{\Delta}) \overset{\blacktriangleright}{\equiv} (K_1|K_2)\{^{\Phi}_{\Delta}$$

Then,

$\text{WF}(K_1|K_2\{^{\Phi}_{\Delta})$

$\Leftrightarrow \quad \langle\text{definition of WF}(\cdot)\rangle$

$\text{WF}(K_1)$

$\wedge\text{WF}(K_2\{^{\Phi}_{\Delta})$

$\wedge(\text{STATE}(K_1) \pitchfork \text{STATE}(K_2\{^{\Phi}_{\Delta}))$

$\Leftrightarrow \quad \langle\text{definition of WF}(\cdot) \text{ and STATE}(\cdot)\rangle$

$\text{WF}(K_1)$

$\wedge\text{WF}(K_2)$

$\wedge(\text{STATE}(K_2) \pitchfork \text{Dom}(\Delta))$

$\wedge(\text{OBJECTS}(\Phi) \subseteq (\text{STATE}(K_2) \cup \text{Dom}(\Delta)))$

$\wedge(\text{STATE}(K_1) \pitchfork ((\text{STATE}(K_2) \cup \text{Dom}(\Delta)) \setminus \text{Dom}(\Phi)))$

$\Leftrightarrow \quad \langle \text{STATE}(K_1) \subseteq \text{FN}(K_1),\ \text{FN}(K_1) \pitchfork \text{Dom}(\Phi)$

$\quad \text{and OBJECTS}(\Phi) \subseteq \text{Dom}(\Phi)\rangle$

$\text{WF}(K_1)$

$\wedge\text{WF}(K_2)$

$\wedge(\text{STATE}(K_2) \pitchfork \text{Dom}(\Delta))$

$\wedge(\text{OBJECTS}(\Phi) \subseteq (\text{STATE}(K_1) \cup \text{STATE}(K_2) \cup \text{Dom}(\Delta)))$

$\wedge(\text{STATE}(K_1) \pitchfork (\text{STATE}(K_2) \cup \text{Dom}(\Delta)))$

$\Leftrightarrow \quad \langle \text{definition of STATE}(\cdot) \text{ and set theory}\rangle$

$\text{WF}(K_1)$

$\wedge\text{WF}(K_2)$

$\wedge(\text{STATE}(K_2) \pitchfork \text{Dom}(\Delta))$

$\wedge(\text{OBJECTS}(\Phi) \subseteq (\text{STATE}(K_1|K_2) \cup \text{Dom}(\Delta)))$

$\wedge(\text{STATE}(K_1) \pitchfork \text{STATE}(K_2))$

$\wedge(\text{STATE}(K_1) \pitchfork \text{Dom}(\Delta))$

$\Leftrightarrow \quad \langle \text{definition of WF}(\cdot) \text{ and set theory}\rangle$

$\text{WF}(K_1|K_2)$

$\wedge((\text{STATE}(K_1) \cup \text{STATE}(K_2)) \pitchfork \text{Dom}(\Delta))$

$\wedge(\text{OBJECTS}(\Phi) \subseteq (\text{STATE}(K_1|K_2) \cup \text{Dom}(\Delta)))$

$\Leftrightarrow \quad \langle \text{definition of STATE}(\cdot)\rangle$

$\text{WF}(K_1|K_2)$

$\wedge(\text{STATE}(K_1|K_2) \pitchfork \text{Dom}(\Delta))$

$\wedge(\text{OBJECTS}(\Phi) \subseteq (\text{STATE}(K_1|K_2) \cup \text{Dom}(\Delta)))$

$\Leftrightarrow \quad \langle \text{definition of WF}(\cdot)\rangle$

$\text{WF}((K_1|K_2)\{^{\Phi}_{\Delta})$

Most of the reasoning steps depend on the side conditions of the EXTRUDE rule. ■

4.2.13
p106

Lemma B.3.16 *For two configurations, K_1 and K_2, and a renaming, ρ,*

$$K_1 \stackrel{\blacktriangleright}{\equiv} K_2 \Rightarrow K_1\rho \stackrel{\blacktriangleright}{\equiv} K_1\rho$$

Proof: We use induction on the proof tree of $K_1 \stackrel{\blacktriangleright}{\equiv} K_2$. Again, the two interesting cases are FLATTEN and EXTRUDE.

Say the proof tree is an instance of FLATTEN:

$$K\{^{\Phi_1}_{\Delta_1}\{^{\Phi_2}_{\Delta_2} \stackrel{\blacktriangleright}{\equiv} K\{^{\Phi_1 \cup \Phi_2}_{\Delta_1 \cup \Delta_2}$$

Then

$$\begin{array}{ll} & (K\{^{\Phi_1}_{\Delta_1}\{^{\Phi_2}_{\Delta_2})\rho \\ = & \langle \text{renaming procedure, choosing } \delta_2 \text{ appropriately} \rangle \\ & (K\{^{\Phi_1}_{\Delta_1})\delta_2\rho\{^{\Phi_2\delta_2\rho}_{\Delta_2\delta_2\rho} \\ = & \langle \text{renaming procedure, no shove necessary due to choice of } \delta_2 \rangle \\ & (K\delta_2\{^{\Phi_1\delta_2}_{\Delta_1\delta_2})\rho\{^{\Phi_2\delta_2\rho}_{\Delta_2\delta_2\rho} \\ = & \langle \text{renaming procedure, choosing } \delta_1 \text{ appropriately} \rangle \\ & K\delta_2\delta_1\rho\{^{\Phi_1\delta_2\delta_1\rho}_{\Delta_1\delta_2\delta_1\rho}\{^{\Phi_2\delta_2\rho}_{\Delta_2\delta_2\rho} \\ \equiv_\alpha & \langle \text{by change of bound names and congruence; } \delta \text{ given below} \rangle \\ & K\delta_2\delta_1\rho\delta\{^{\Phi_1\delta_2\delta_1\rho\delta}_{\Delta_1\delta_2\delta_1\rho\delta}\{^{\Phi_2\delta_2\rho}_{\Delta_2\delta_2\rho} \end{array}$$

where δ shoves $\text{Dom}(\Phi_1\delta_2\delta_1\rho) \cap (\text{Dom}(\Phi_2\delta_2\rho) \cup \text{FN}(\Delta_2\delta_2\rho))$. We introduce δ to flatten the above configuration. It is designed to guarantee the side conditions of the FLATTEN rule, giving the following.

$$K\delta_2\delta_1\rho\delta\{^{\Phi_1\delta_2\delta_1\rho\delta}_{\Delta_1\delta_2\delta_1\rho\delta}\{^{\Phi_2\delta_2\rho}_{\Delta_2\delta_2\rho} \stackrel{\blacktriangleright}{\equiv} K\delta_2\delta_1\rho\delta\{^{\Phi_1\delta_2\delta_1\rho\delta \cup \Phi_2\delta_2\rho}_{\Delta_1\delta_2\delta_1\rho\delta \cup \Delta_2\delta_2\rho}$$

Two of the side conditions follow immediately from the choice of δ:

$$\begin{array}{rcl} \text{Dom}(\Phi_1\delta_2\delta_1\rho\delta) & \pitchfork & \text{Dom}(\Phi_2\delta_2\rho) \\ \text{Dom}(\Phi_1\delta_2\delta_1\rho\delta) & \pitchfork & \text{FN}(\Delta_2\delta_2\rho) \end{array}$$

The third side condition requires some reasoning. Let $F = \mathrm{FN}(K\{^{\Phi_1}_{\Delta_1}\{^{\Phi_2}_{\Delta_2})$. Here, we write type and state dictionaries in place of their domains to reduce visual complexity.

$\Delta_1 \pitchfork \Delta_2$

$\Rightarrow$ $\langle$set theory$\rangle$

$(\Delta_1 \cap (F \cup \Phi_2)) \pitchfork \Delta_2$

$\Rightarrow$ $\langle\delta_2$ is a bijection$\rangle$

$(\Delta_1 \cap (F \cup \Phi_2))\delta_2 \pitchfork \Delta_2\delta_2$

$\Rightarrow$ $\langle$ by set theory $(\Delta_1 \cap (F \cup \Phi_2))\delta_2 \subseteq F\delta_2 \cup \Phi_2\delta_2$

and $F\delta_2 \cup \Phi_2\delta_2 \pitchfork \Phi_1\delta_2$

so $(\Delta_1 \cap (F \cup \Phi_2))\delta_2 \pitchfork \Phi_1\delta_2$

and therefore $(\Delta_1 \cap (F \cup \Phi_2)\delta_2 \pitchfork \mathrm{Change}(\delta_1)$ $\rangle$

$(\Delta_1 \cap (F \cup \Phi_2))\delta_2\delta_1 \pitchfork \Delta_2\delta_2$

$\Rightarrow$ $\langle\rho$ is a bijection$\rangle$

$(\Delta_1 \cap (F \cup \Phi_2))\delta_2\delta_1\rho \pitchfork \Delta_2\delta_2\rho$

$\Rightarrow$ $\langle$ since $(\Delta_1 \cap (F \cup \Phi_2))\delta_2\delta_1\rho \pitchfork \Phi_1\delta_2\delta_1\rho$

we have $(\Delta_1 \cap (F \cup \Phi_2))\delta_2\delta_1\rho \pitchfork \mathrm{Change}(\delta)$ $\rangle$

$(\Delta_1 \cap (F \cup \Phi_2))\delta_2\delta_1\rho\delta \pitchfork \Delta_2\delta_2\rho$

$\Rightarrow$ $\langle$ by choice of δ, observing $\mathrm{Dom}(\Delta_2) \subseteq \mathrm{FN}(\Delta_2)$ and using logic $\rangle$

$(\Delta_1 \cap (F \cup \Phi_2))\delta_2\delta_1\rho\delta \pitchfork \Delta_2\delta_2\rho$

$\wedge\ \Phi_1\delta_2\delta_1\rho\delta \pitchfork \Delta_2\delta_2\rho$

$\Rightarrow$ $\langle$set theory$\rangle$

$(\Delta_1 \cap (F \cup \Phi_2))\delta_2\delta_1\rho\delta \pitchfork \Delta_2\delta_2\rho$

$\wedge\ (\Delta_1 \cap \Phi_1)\delta_2\delta_1\rho\delta \pitchfork \Delta_2\delta_2\rho$

$\Rightarrow$ $\langle$set theory$\rangle$

$(\Delta_1 \cap (F \cup \Phi_2 \cup \Phi_1))\delta_2\delta_1\rho\delta \pitchfork \Delta_2\delta_2\rho$

$\Rightarrow$ $\langle\Delta_1 \subseteq (F \cup \Phi_2 \cup \Phi_1)\rangle$

$\Delta_1\delta_2\delta_1\rho\delta \pitchfork \Delta_2\delta_2\rho$

Thus the third side condition holds and by transitivity we have

$$(K\{^{\Phi_1}_{\Delta_1}\{^{\Phi_2}_{\Delta_2})\rho \overset{\blacktriangleright}{\equiv} K\delta_2\delta_1\rho\delta\{^{\Phi_1\delta_2\delta_1\rho\delta\cup\Phi_2\delta_2\rho}_{\Delta_1\delta_2\delta_1\rho\delta\cup\Delta_2\delta_2\rho}$$

Now considering the right-hand side, we have

$$\begin{array}{ll} & (K\{^{\Phi_1\cup\Phi_2}_{\Delta_1\cup\Delta_2})\rho \\ = & \langle \text{renaming procedure, choosing } \delta' \text{ appropriately} \rangle \\ & K\delta'\rho\{^{(\Phi_1\cup\Phi_2)\delta'\rho}_{(\Delta_1\cup\Delta_2)\delta'\rho} \end{array}$$

We will now use a ρ^α-construction to show that

$$K\delta_2\delta_1\rho\delta\{^{\Phi_1\delta_2\delta_1\rho\delta\cup\Phi_2\delta_2\rho}_{\Delta_1\delta_2\delta_1\rho\delta\cup\Delta_2\delta_2\rho} \equiv_\alpha K\delta'\rho\{^{(\Phi_1\cup\Phi_2)\delta'\rho}_{(\Delta_1\cup\Delta_2)\delta'\rho}$$

and transitivity will give the desired result.

We define ρ^α as an isolated sum:

$$\rho^\alpha = 1|_{F\rho} + (\delta \circ \rho \circ \delta_1 \circ \delta_2 \circ \delta' \circ \rho^{-1})\big|_{\Phi_1\delta'\rho} + (\rho \circ \delta_2 \circ \delta' \circ \rho^{-1})\big|_{\Phi_2\delta'\rho}$$

To show that the isolated sum is well-formed we show that the isolated sets we use have the correct disjointness properties. First,

$$\begin{array}{ll} & \text{FN}(K\delta_2\delta_1\rho\delta\{^{\Phi_1\delta_2\delta_1\rho\delta\cup\Phi_2\delta_2\rho}_{\Delta_1\delta_2\delta_1\rho\delta\cup\Delta_2\delta_2\rho}) \\ = & \langle \text{by Lemma B.3.13} \rangle \\ & \text{FN}((K\{^{\Phi_1}_{\Delta_1}\{^{\Phi_2}_{\Delta_2})\rho) \\ = & \langle \text{by Lemma B.3.1} \rangle \\ & \text{FN}((K\{^{\Phi_1}_{\Delta_1}\{^{\Phi_2}_{\Delta_2}))\rho \\ = & \langle \text{definition of } F \rangle \\ & F\rho \\ = & \langle \text{ by Lemma B.3.13, since } K\{^{\Phi_1}_{\Delta_1}\{^{\Phi_2}_{\Delta_2} \overset{\blacktriangleright}{\equiv} K\{^{\Phi_1\cup\Phi_2}_{\Delta_1\cup\Delta_2} \ \rangle \\ & \text{FN}((K\{^{\Phi_1\cup\Phi_2}_{\Delta_1\cup\Delta_2}))\rho \end{array}$$

$$
\begin{aligned}
= &\quad \langle \text{by Lemma B.3.1} \rangle \\
&\mathrm{FN}((K\{^{\Phi_1 \cup \Phi_2}_{\Delta_1 \cup \Delta_2})\rho) \\
= &\quad \langle \text{the configurations are equal} \rangle \\
&\mathrm{FN}(K\delta'\rho\{^{(\Phi_1 \cup \Phi_2)\delta'\rho}_{(\Delta_1 \cup \Delta_2)\delta'\rho})
\end{aligned}
$$

The well-formedness conditions for the domain sets are $F\rho \pitchfork \Phi_1\delta'\rho$, $F\rho \pitchfork \Phi_2\delta'\rho$ and $\Phi_1\delta'\rho \pitchfork \Phi_2\delta'\rho$ and the well-formedness conditions for the co-domain sets are $F\rho \pitchfork \Phi_1\delta_2\delta_1\rho\delta$, $F\rho \pitchfork \Phi_2\delta_2\rho$ and $\Phi_1\delta_2\delta_1\rho\delta \pitchfork \Phi_2\delta_2\rho$. The proofs of these conditions are similar to the proof of the side condition for the FLATTEN rule we gave above, so we omit them.

We can use ρ^α in the CHNG-SCP rule since it agrees with 1 on $F\rho$ and, hence, leaves free names alone. This gives:

$$
K\delta'\rho\{^{(\Phi_1 \cup \Phi_2)\delta'\rho}_{(\Delta_1 \cup \Delta_2)\delta'\rho} \equiv_\alpha K\delta'\rho\rho^\alpha\{^{(\Phi_1 \cup \Phi_2)\delta'\rho\rho^\alpha}_{(\Delta_1 \cup \Delta_2)\delta'\rho\rho^\alpha}
$$

Naturally $(\Phi_1 \cup \Phi_2)\delta'\rho\rho^\alpha = (\Phi_1\delta'\rho \cup \Phi_2\delta'\rho)\rho^\alpha = \Phi_1\delta_2\delta_1\rho\delta \cup \Phi_2\delta_2\rho$.

Using various properties of isolated sum, we have

$$
\begin{aligned}
&\rho^\alpha \circ \rho \circ \delta' \\
= &\quad \langle \text{definition of } \rho^\alpha \rangle \\
&(1|_{F\rho} \\
&\;+(\delta \circ \rho \circ \delta_1 \circ \delta_2 \circ \delta' \circ \rho^{-1})|_{\Phi_1\delta'\rho} \\
&\;+(\rho \circ \delta_2 \circ \delta' \circ \rho^{-1})|_{\Phi_2\delta'\rho} \qquad) \circ \rho \circ \delta' \\
\simeq &\quad \langle \text{Lemma A.2.6} \rangle \\
&(\rho \circ \delta')|_{F\rho\rho^{-1}\delta'} \\
&+(\delta \circ \rho \circ \delta_1 \circ \delta_2 \circ \delta' \circ \rho^{-1} \circ \rho \circ \delta')|_{\Phi_1\delta'\rho\rho^{-1}\delta'} \\
&+(\rho \circ \delta_2 \circ \delta' \circ \rho^{-1} \circ \rho \circ \delta')|_{\Phi_2\delta'\rho\rho^{-1}\delta'} \\
= &\quad \langle F\rho\rho^{-1}\delta' = F,\ \Phi_1\delta'\rho\rho^{-1}\delta' = \Phi_1 \text{ and } \Phi_2\delta'\rho\rho^{-1}\delta' = \Phi_2 \rangle \\
&(\rho \circ \delta')|_F \\
&+(\delta \circ \rho \circ \delta_1 \circ \delta_2 \circ \delta' \circ \rho^{-1} \circ \rho \circ \delta')|_{\Phi_1}
\end{aligned}
$$

$+(\rho \circ \delta_2 \circ \delta' \circ \rho^{-1} \circ \rho \circ \delta')|_{\Phi_2}$

$\simeq$ $\langle$Lemma A.2.3$\rangle$

$\rho|_F$

$+(\delta \circ \rho \circ \delta_1 \circ \delta_2)|_{\Phi_1}$

$+(\rho \circ \delta_2)|_{\Phi_2}$

We derive two statements of agreement from this. Firstly

$\rho^\alpha \circ \rho \circ \delta'$

$\simeq$ $\langle$from above$\rangle$

$\rho|_F + (\delta \circ \rho \circ \delta_1 \circ \delta_2)|_{\Phi_1} + (\rho \circ \delta_2)|_{\Phi_2}$

$\simeq$ $\langle$Lemma A.2.3$\rangle$

$(\delta \circ \rho \circ \delta_1 \circ \delta_2)|_F + (\delta \circ \rho \circ \delta_1 \circ \delta_2)|_{\Phi_1} + (\delta \circ \rho \circ \delta_1 \circ \delta_2)|_{\Phi_2}$

$\simeq$ $\langle$ Lemma A.2.1 $\rangle$

$(\delta \circ \rho \circ \delta_1 \circ \delta_2)$

and secondly

$\rho^\alpha \circ \rho \circ \delta'$

$\simeq$ $\langle$from above$\rangle$

$\rho|_F + (\delta \circ \rho \circ \delta_1 \circ \delta_2)|_{\Phi_1} + (\rho \circ \delta_2)|_{\Phi_2}$

$\simeq$ $\langle$Lemma A.2.3$\rangle$

$(\rho \circ \delta_2)|_F + (\delta \circ \rho \circ \delta_1 \circ \delta_2)|_{\Phi_1} + (\rho \circ \delta_2)|_{\Phi_2}$

$\simeq$ $\langle$reorder, desugar the ternary sum and use Lemma A.2.1$\rangle$

$(\rho \circ \delta_2)|_{\Phi_2 \cup F} + (\delta \circ \rho \circ \delta_1 \circ \delta_2)|_{\Phi_1}$

Since $\mathrm{FN}(K) \subseteq F \cup \Phi_1 \cup \Phi_2$ we can use Corollary B.3.10 to give

$$K(\rho^\alpha \circ \rho \circ \delta') \equiv_\alpha K(\delta \circ \rho \circ \delta_1 \circ \delta_2)$$

Similarly, since $\mathrm{FN}(\Delta_1) \subseteq F \cup \Phi_1 \cup \Phi_2$, we have

$$\Delta_1(\rho^\alpha \circ \rho \circ \delta') = \Delta_1(\delta \circ \rho \circ \delta_1 \circ \delta_2)$$

Since $\mathrm{FN}(\Delta_2) \subseteq F \cup \Phi_2$ we have

$$\Delta_2(\rho^\alpha \circ \rho \circ \delta') = \Delta_2(\rho \circ \delta_2)$$

We can summarise the facts so far as

$$\begin{array}{rcl} K\delta'\rho\rho^\alpha & \equiv_\alpha & K\delta_2\delta_1\rho\delta \\ (\Phi_1 \cup \Phi_2)\delta'\rho\rho^\alpha & = & \Phi_1\delta_2\delta_1\rho\delta \cup \Phi_2\delta_2\rho \\ (\Delta_1 \cup \Delta_2)\delta'\rho\rho^\alpha & = & \Delta_1\delta_2\delta_1\rho\delta \cup \Delta_2\delta_2\rho \end{array}$$

Using CONG-SCP, we have

$$K\delta'\rho\rho^\alpha\{^{(\Phi_1\cup\Phi_2)\delta'\rho\rho^\alpha}_{(\Delta_1\cup\Delta_2)\delta'\rho\rho^\alpha} \equiv_\alpha K\delta_2\delta_1\rho\delta\{^{\Phi_1\delta_2\delta_1\rho\delta\cup\Phi_2\delta_2\rho}_{\Delta_1\delta_2\delta_1\rho\delta\cup\Delta_2\delta_2\rho}$$

The result follows by transitivity.

Say the proof tree is an instance of EXTRUDE:

$$K_1|(K_2\{^\Phi_\Delta) \stackrel{\blacktriangleright}{\equiv} (K_1|K_2)\{^\Phi_\Delta$$

Now

$$\begin{array}{ll} & (K_1|(K_2\{^\Phi_\Delta))\rho \\ = & \langle\text{renaming procedure}\rangle \\ & K_1\rho|(K_2\{^\Phi_\Delta)\rho \\ = & \langle\text{renaming procedure, choosing } \delta_1 \text{ appropriately}\rangle \\ & K_1\rho|(K_2\delta_1\rho\{^{\Phi\delta_1\rho}_{\Delta\delta_1\rho}) \\ \equiv_\alpha & \langle\text{CHNG-SCP and CONG-PAR; } \delta \text{ given below}\rangle \\ & K_1\rho|(K_2\delta_1\rho\delta\{^{\Phi\delta_1\rho\delta}_{\Delta\delta_1\rho\delta}) \\ \stackrel{\blacktriangleright}{\equiv} & \langle\text{EXTRUDE, by choice of } \delta\rangle \\ & (K_1\rho|K_2\delta_1\rho\delta)\{^{\Phi\delta_1\rho\delta}_{\Delta\delta_1\rho\delta} \end{array}$$

where δ shoves $\mathrm{FN}(K_1\rho) \cap \mathrm{Dom}(\Phi\delta_1\rho)$. Also

$$
\begin{array}{ll}
& ((K_1|K_2)\{^{\Phi}_{\Delta})\rho \\
= & \langle \text{renaming procedure, choosing } \delta_2 \text{ appropriately} \rangle \\
& (K_1|K_2)\delta_2\rho\{^{\Phi\delta_2\rho}_{\Delta\delta_2\rho} \\
= & \langle \text{renaming procedure} \rangle \\
& (K_1\delta_2|K_2\delta_2)\rho\{^{\Phi\delta_2\rho}_{\Delta\delta_2\rho} \\
= & \langle \text{Change}(\delta_2) \pitchfork \text{FN}(K_1) \rangle \\
& (K_1|K_2\delta_2)\rho\{^{\Phi\delta_2\rho}_{\Delta\delta_2\rho} \\
= & \langle \text{renaming procedure} \rangle \\
& (K_1\rho|K_2\delta_2\rho)\{^{\Phi\delta_2\rho}_{\Delta\delta_2\rho}
\end{array}
$$

The third step is since δ_2 shoves a subset of Dom(Φ).

A typical ρ^α-construction with

$$\rho^\alpha = (\delta \circ \rho \circ \delta_1 \circ \delta_2 \circ \rho^{-1})$$

gives us

$$(K_1\rho|K_2\delta_2\rho)\{^{\Phi\delta_2\rho}_{\Delta\delta_2\rho} \equiv_\alpha (K_1\rho|K_2\delta_1\rho\delta)\{^{\Phi\delta_1\rho\delta}_{\Delta\delta_1\rho\delta}$$

from which the required result is easily obtained. ■

B.3.4 Labelled Transition System

In this section, we provide the proofs of lemmas 4.2.14 and 4.2.15.

4.2.14 p112

Lemma B.3.17 *Say K_1 and K_2 are configurations such that*

$$\Gamma, \Phi \blacktriangleright K \xrightarrow[l_2]{l_1} K'$$

for any labels l_1 and l_2. Then

$$\text{STATE}(K) = \text{STATE}(K')$$

Proof: We use an induction on the proof tree of $\Gamma, \Phi \blacktriangleright K \xrightarrow[l_2]{l_1} K'$. There are only two interesting cases.

Say that the proof tree is an instance of the create object rule:

$$\Gamma, \Phi \blacktriangleright o[\texttt{create}\ o'\colon \mathrm{ClT}\ p] \longrightarrow o[p]\{^{\{o':\mathrm{ClT}\}}_{\{o':=\emptyset\}}$$

This adds an entry to the state dictionary of the right-hand configuration. However, since it also binds it with an entry in the type dictionary, the state doesn't change across the transition.

Say that the last step of the proof is an instance of the scope restriction rule:

$$\frac{\Gamma, \Phi \cup \Phi' \blacktriangleright K \xrightarrow[l_2]{l_1} K'}{\Gamma, \Phi \blacktriangleright K\{^{\Phi'}_{\Delta} \xrightarrow[l_2]{l_1} K'\{^{\Phi'}_{\Delta}}$$

The induction hypothesis gives us $\mathrm{STATE}(K) = \mathrm{STATE}(K')$. Then,

$$\begin{array}{ll} & \mathrm{STATE}(K\{^{\Phi'}_{\Delta}) \\ = & \quad \langle \text{definition of } \mathrm{STATE}(\cdot) \rangle \\ & (\mathrm{STATE}(K) \cup \mathrm{Dom}(\Delta)) \setminus \mathrm{OBJECTS}(\Phi') \\ = & \quad \langle \text{induction hypothesis} \rangle \\ & (\mathrm{STATE}(K') \cup \mathrm{Dom}(\Delta)) \setminus \mathrm{OBJECTS}(\Phi') \\ = & \quad \langle \text{definition of } \mathrm{STATE}(\cdot) \rangle \\ & \mathrm{STATE}(K'\{^{\Phi'}_{\Delta}) \end{array}$$

as required in this case. ■

Lemma B.3.18 *Say K_1 and K_2 are configurations such that*

4.2.15
p112

$$\Gamma, \Phi \blacktriangleright K \xrightarrow[l_2]{l_1} K'$$

for any labels l_1 and l_2. Then

$$\mathrm{WF}(K) \Rightarrow \mathrm{WF}(K')$$

Proof: We use induction on the proof tree of $\Gamma, \Phi \blacktriangleright K \xrightarrow[l_2]{l_1} K'$. Most of the cases are obvious, so we only give three cases.

Say that the last step of the proof tree is an instance of the scope restriction rule:

$$\frac{\Gamma, \Phi \cup \Phi' \blacktriangleright K \xrightarrow[l_2]{l_1} K'}{\Gamma, \Phi \blacktriangleright K\{^{\Phi'}_{\Delta} \xrightarrow[l_2]{l_1} K'\{^{\Phi'}_{\Delta}}$$

Then the induction hypothesis gives us $\mathrm{WF}(K) \Rightarrow \mathrm{WF}(K')$.

$$\begin{array}{ll}
& \mathrm{WF}(K\{^{\Phi'}_{\Delta}) \\
\Rightarrow & \langle \text{definition of } \mathrm{WF}(\cdot) \rangle \\
& \mathrm{WF}(K) \\
& \wedge(\mathrm{STATE}(K) \pitchfork \mathrm{Dom}(\Delta)) \\
& \wedge(\mathrm{OBJECTS}(\Phi') \subseteq (\mathrm{STATE}(K) \cup \mathrm{Dom}(\Delta))) \\
\Rightarrow & \langle \text{induction hypothesis and Lemma B.3.17} \rangle \\
& \mathrm{WF}(K') \\
& \wedge(\mathrm{STATE}(K') \pitchfork \mathrm{Dom}(\Delta)) \\
& \wedge(\mathrm{OBJECTS}(\Phi') \subseteq (\mathrm{STATE}(K') \cup \mathrm{Dom}(\Delta))) \\
\Rightarrow & \langle \text{definition of } \mathrm{WF}(\cdot) \rangle \\
& \mathrm{WF}(K'\{^{\Phi'}_{\Delta})
\end{array}$$

as required in this case.

Say that the proof tree is an instance of the create object rule:

$$\Gamma, \Phi \blacktriangleright o[\texttt{create}\ o' \colon \mathrm{ClT}\ p] \longrightarrow o[p]\{^{\{o' : \mathrm{ClT}\}}_{\{o' := \emptyset\}}$$

Then

$$\mathrm{WF}(o[\texttt{create}\ o' \colon \mathrm{ClT}\ p]) = \mathrm{true}$$

and

$$
\begin{aligned}
& \mathrm{WF}(o[p]\{^{\{o':\mathrm{ClT}\}}_{\{o':=\emptyset\}}) \\
\Leftrightarrow \quad & \langle \text{definition of WF}(\cdot) \rangle \\
& \mathrm{WF}(o[p]) \\
& \wedge(\mathrm{STATE}(o[p]) \pitchfork \mathrm{Dom}(\{o' := \emptyset\})) \\
& \wedge(\mathrm{OBJECTS}(\{o' \colon \mathrm{ClT}\}) \subseteq (\mathrm{STATE}(o[p]) \cup \mathrm{Dom}(\{o' := \emptyset\}))) \\
\Leftrightarrow \quad & \langle \text{definitions of the various functions} \rangle \\
& (\mathrm{true} \wedge \mathrm{true} \wedge \mathrm{true}) \\
\Leftrightarrow \quad & \langle \text{logic} \rangle \\
& \mathrm{true}
\end{aligned}
$$

Say that the last step of the proof tree is an instance of the communication rule:

$$
\frac{\Gamma, \Phi \blacktriangleright K_1 \xrightarrow{c!\langle v_1,\ldots v_n\rangle} K_1' \qquad \Gamma, \Phi \blacktriangleright K_2 \xrightarrow{c?\langle v_1,\ldots v_n\rangle} K_2'}{\Gamma, \Phi \blacktriangleright K_1|K_2 \longrightarrow K_1'|K_2'}
$$

So we have

$$
\begin{aligned}
& \mathrm{WF}(K_1|K_2) \\
\Leftrightarrow \quad & \langle \text{definition of WF}(\cdot) \rangle \\
& \mathrm{WF}(K_1) \\
& \wedge\mathrm{WF}(K_2) \\
& \wedge\mathrm{STATE}(K_1) \pitchfork \mathrm{STATE}(K_2) \\
\Leftrightarrow \quad & \langle \text{induction hypothesis and Lemma B.3.17} \rangle \\
& \mathrm{WF}(K_1') \\
& \wedge\mathrm{WF}(K_2') \\
& \wedge\mathrm{STATE}(K_1') \pitchfork \mathrm{STATE}(K_2') \\
\Leftrightarrow \quad & \langle \text{definition of WF}(\cdot) \rangle \\
& \mathrm{WF}(K_1'|K_2')
\end{aligned}
$$

■

B.4 Flatten and the Relationship

In this section we give the proofs of results 4.3.1–4.3.5. These results prove properties of the Flatten algorithm and establish the relationship between the configuration-based dynamic system and the agent-based dynamic system.

4.3.1 p119

Theorem B.4.1 *For a configuration K,*

$$\mathrm{WF}(K) \Rightarrow K \overset{\blacktriangleright}{\equiv} \mathrm{Flatten}(K)$$

Proof: We use an induction on the structure of K. The two base cases are easy:

$$\begin{array}{rclclcl} K & = & \mathrm{Nil} & \overset{\blacktriangleright}{\equiv} & \mathrm{Nil}\{^{\emptyset}_{\emptyset} & = & \mathrm{Flatten}(K) \\ K & = & o[p] & \overset{\blacktriangleright}{\equiv} & o[p]\{^{\emptyset}_{\emptyset} & = & \mathrm{Flatten}(K) \end{array}$$

Say that K has the form $K'\{^{\Phi}_{\Delta}$ and that $\mathrm{Flatten}(K') = g'\{^{\Phi'}_{\Delta'}$. Then

$$\mathrm{Flatten}(K) = g'\delta\{^{\Phi'\delta\cup\Phi}_{\Delta'\delta\cup\Delta}$$

where δ shoves $(\mathrm{Dom}(\Phi) \cup \mathrm{FN}(\Delta)) \cap \mathrm{Dom}(\Phi')$. Since $\mathrm{WF}(K)$ we know that $\mathrm{WF}(K')$ by the definition of $\mathrm{WF}(\cdot)$. Thus, the induction hypothesis applies to K'. Therefore

$$\begin{array}{ll} & K' \\ \overset{\blacktriangleright}{\equiv} & \langle \text{induction hypothesis} \rangle \\ & \mathrm{Flatten}(K') \\ = & \langle \text{assumption} \rangle \\ & g'\{^{\Phi'}_{\Delta'} \\ \equiv_\alpha & \langle \text{CHNG-SCP; } \mathrm{FN}(g'\{^{\Phi'}_{\Delta'}) \pitchfork \mathrm{Change}(\delta) \rangle \\ & g'\delta\{^{\Phi'\delta}_{\Delta'\delta} \end{array}$$

By choice of δ, we have the following two properties:

$$\begin{array}{rcl} \mathrm{Dom}(\Phi'\delta) & \pitchfork & \mathrm{Dom}(\Phi) \\ \mathrm{Dom}(\Phi'\delta) & \pitchfork & \mathrm{FN}(\Delta) \end{array}$$

Since we know $\mathrm{Dom}(\Delta) \subseteq \mathrm{FN}(\Delta)$, the second of these gives us $\mathrm{Dom}(\Phi'\delta) \pitchfork \mathrm{Dom}(\Delta)$. Now

$$
\begin{array}{ll}
 & \mathrm{Dom}(\Delta) \\
\pitchfork & \quad \langle \mathrm{WF}(K) \rangle \\
 & \mathrm{STATE}(K') \\
= & \quad \langle \text{Lemma B.3.14} \rangle \\
 & \mathrm{STATE}(g'\delta\{^{\Phi'\delta}_{\Delta'\delta}) \\
= & \quad \langle \text{definition of } \mathrm{STATE}(\cdot) \rangle \\
 & (\mathrm{STATE}(g'\delta) \cup \mathrm{Dom}(\Delta'\delta)) \smallsetminus \mathrm{Dom}(\Phi'\delta) \\
= & \quad \langle \mathrm{STATE}(g'\delta) = \emptyset, \text{ since agents have no state} \rangle \\
 & \mathrm{Dom}(\Delta'\delta) \smallsetminus \mathrm{Dom}(\Phi'\delta)
\end{array}
$$

Since we know $\mathrm{Dom}(\Delta) \pitchfork \mathrm{Dom}(\Phi'\delta)$ we get

$$\mathrm{Dom}(\Delta'\delta) \pitchfork \mathrm{Dom}(\Delta)$$

These properties justify the following reasoning and also reassure us that, in this case, well-formed configurations can be flattened without a clash between state dictionaries.

$$
\begin{array}{ll}
 & K'\{^{\Phi}_{\Delta} \\
\stackrel{\blacktriangleright}{\equiv} & \quad \langle \text{from above, using CONG-SCP} \rangle \\
 & g'\delta\{^{\Phi'\delta}_{\Delta'\delta}\{^{\Phi}_{\Delta} \\
\stackrel{\blacktriangleright}{\equiv} & \quad \langle \text{FLATTEN} \rangle \\
 & g'\delta\{^{(\Phi'\delta)\cup\Phi}_{(\Delta'\delta)\cup\Delta}
\end{array}
$$

Hence

$$K \stackrel{\blacktriangleright}{\equiv} \mathrm{Flatten}(K)$$

as required.

Next, consider the case where K has the form $(K_1|K_2)$ and say that $\text{Flatten}(K_1) = g_1\{^{\Phi_1}_{\Delta_1}$ and $\text{Flatten}(K_2) = g_2\{^{\Phi_2}_{\Delta_2}$. Then

$$\text{Flatten}(K_1|K_2) = (g_1\delta_1|g_2\delta_2)\{^{\Phi_1\delta_1\cup\Phi_2\delta_2}_{\Delta_1\delta_1\cup\Delta_2\delta_2}$$

where δ_2 shoves $\text{FN}(g_1\{^{\Phi_1}_{\Delta_1})\cap\text{Dom}(\Phi_2)$ and δ_1 shoves $(\text{FN}(g_2\delta_2)\cup\text{Dom}(\Phi_2\delta_2)\cup \text{FN}(\Delta_2\delta_2)) \cap \text{Dom}(\Phi_1)$.

Since $\text{WF}(K)$, we know that $\text{WF}(K_1)$ and $\text{WF}(K_2)$ by the definition of $\text{WF}(\cdot)$. Thus, the induction hypothesis applies to K_1 and K_2. Using arguments similar to the above case, we know that

$$\begin{array}{rcl} K_1 & \overset{\blacktriangleright}{\equiv} & g_1\delta_1\{^{\Phi_1\delta_1}_{\Delta_1\delta_1} \\ K_2 & \overset{\blacktriangleright}{\equiv} & g_2\delta_2\{^{\Phi_2\delta_2}_{\Delta_2\delta_2} \end{array}$$

By the choice of δ_1, we know that

$$\begin{array}{rcl} \text{Dom}(\Phi_1\delta_1) & \pitchfork & \text{Dom}(\Phi_2\delta_2) \\ \text{Dom}(\Phi_1\delta_1) & \pitchfork & \text{FN}(\Delta_2\delta_2) \end{array}$$

Using arguments similar to the above case, we have

$$\begin{array}{rcl} \text{STATE}(K_1) & = & \text{Dom}(\Delta_1) \setminus \text{Dom}(\Phi_1) \\ & = & \text{Dom}(\Delta_1\delta_1) \setminus \text{Dom}(\Phi_1\delta_1) \\ \text{STATE}(K_2) & = & \text{Dom}(\Delta_2) \setminus \text{Dom}(\Phi_2) \\ & = & \text{Dom}(\Delta_2\delta_2) \setminus \text{Dom}(\Phi_2\delta_2) \end{array}$$

Since $\text{WF}(K)$, we also know that $\text{STATE}(K_1) \pitchfork \text{STATE}(K_2)$. Thus,

$$\text{Dom}(\Delta_1\delta_1) \setminus \text{Dom}(\Phi_1\delta_1) \pitchfork \text{Dom}(\Delta_2\delta_2) \setminus \text{Dom}(\Phi_2\delta_2)$$

But then

$$\begin{array}{ll} & \text{Dom}(\Delta_1\delta_1) \setminus \text{Dom}(\Phi_1\delta_1) \\ \subseteq & \langle\text{definitions}\rangle \end{array}$$

$$
\begin{array}{ll}
& \mathrm{FN}(g_1\delta_1\{^{\Phi_1\delta_1}_{\Delta_1\delta_1}) \\
= & \quad \langle \text{Lemma B.3.4} \rangle \\
& \mathrm{FN}(g_1\{^{\Phi_1}_{\Delta_1}) \\
\pitchfork & \quad \langle \text{choice of } \delta_2 \rangle \\
& \mathrm{Dom}(\Phi_2\delta_2)
\end{array}
$$

Consequently, $\mathrm{Dom}(\Delta_1\delta_1) \setminus \mathrm{Dom}(\Phi_1\delta_1) \pitchfork \mathrm{Dom}(\Delta_2\delta_2)$. Using $\mathrm{Dom}(\Phi_1\delta_1) \pitchfork \mathrm{Dom}(\Delta_2\delta_2)$ from above gives us

$$\mathrm{Dom}(\Delta_1\delta_1) \pitchfork \mathrm{Dom}(\Delta_2\delta_2)$$

This property reassures us that well-formed configurations can be flattened without a clash between state dictionaries. The properties we have established above justify the following reasoning.

$$
\begin{array}{ll}
& (K_1 | K_2) \\
\overset{\blacktriangleright}{\equiv} & \quad \langle \text{induction hypothesis and CONG-PAR} \rangle \\
& (g_1\{^{\Phi_1}_{\Delta_1} | g_2\{^{\Phi_2}_{\Delta_2}) \\
\equiv_\alpha & \quad \langle \text{CHNG-SCP and CONG-PAR} \rangle \\
& (g_1\{^{\Phi_1}_{\Delta_1} | g_2\delta_2\{^{\Phi_2\delta_2}_{\Delta_2\delta_2}) \\
\overset{\blacktriangleright}{\equiv} & \quad \langle \text{EXTRUDE; } \mathrm{FN}(g_1\{^{\Phi_1}_{\Delta_1}) \pitchfork \mathrm{Dom}(\Phi_2\delta_2) \rangle \\
& (g_1\{^{\Phi_1}_{\Delta_1} | g_2\delta_2)\{^{\Phi_2\delta_2}_{\Delta_2\delta_2} \\
\equiv_\alpha & \quad \langle \text{CHNG-SCP, CONG-PAR and CONG-SCP} \rangle \\
& (g_1\delta_1\{^{\Phi_1\delta_1}_{\Delta_1\delta_1} | g_2\delta_2)\{^{\Phi_2\delta_2}_{\Delta_2\delta_2} \\
\overset{\blacktriangleright}{\equiv} & \quad \langle \text{PAR-ABL and CONG-SCP} \rangle \\
& (g_2\delta_2 | (g_1\delta_1\{^{\Phi_1\delta_1}_{\Delta_1\delta_1}))\{^{\Phi_2\delta_2}_{\Delta_2\delta_2} \\
\overset{\blacktriangleright}{\equiv} & \quad \langle \text{EXTRUDE and CONG-SCP; } \mathrm{FN}(g_2\delta_2) \pitchfork \mathrm{Dom}(\Phi_1\delta_1) \rangle \\
& (g_2\delta_2 | g_1\delta_1)\{^{\Phi_1\delta_1}_{\Delta_1\delta_1}\{^{\Phi_2\delta_2}_{\Delta_2\delta_2} \\
\overset{\blacktriangleright}{\equiv} & \quad \langle \text{FLATTEN} \rangle \\
& (g_2\delta_2 | g_1\delta_1)\{^{\Phi_1\delta_1 \cup \Phi_2\delta_2}_{\Delta_1\delta_1 \cup \Delta_2\delta_2}
\end{array}
$$

$\overset{\blacktriangleright}{\equiv}$ ⟨PAR-ABL and CONG-SCP⟩

$(g_1\delta_1|g_2\delta_2)\{^{\Phi_1\delta_1\cup\Phi_2\delta_2}_{\Delta_1\delta_1\cup\Delta_2\delta_2}$

Thus

$$K \overset{\blacktriangleright}{\equiv} \text{Flatten}(K)$$

as required. ■

We will need the following two technical lemmas[2].

Lemma B.4.2 *Say that* $\text{Flatten}(K) \equiv_\alpha g_1\{^{\Phi_1}_{\Delta_1}$. *Then*

$$\text{Flatten}(K\{^{\Phi}_{\Delta}) \equiv_\alpha g_1\rho\{^{\Phi_1\rho\cup\Phi}_{\Delta_1\rho\cup\Delta}$$

for some/for any renaming ρ *satisfying* $\text{Dom}(\Phi) \pitchfork \text{Dom}(\Phi_1\rho)$ *and* $\text{FN}(\Delta) \pitchfork \text{Dom}(\Phi_1\rho)$.

Proof: Say

$$\begin{aligned}\text{Flatten}(K) &= g'\{^{\Phi'}_{\Delta'}\\ \text{Flatten}(K\{^{\Phi}_{\Delta}) &= g'\delta\{^{\Phi'\delta\cup\Phi}_{\Delta'\delta\cup\Delta}\end{aligned}$$

Now $g'\{^{\Phi'}_{\Delta'}\equiv_\alpha g_1\{^{\Phi_1}_{\Delta_1}$ so, by Lemma B.3.12, there exists some renaming ρ' such that

$$\Phi_1\rho' = \Phi' \qquad \Delta_1\rho' = \Delta' \qquad g_1\rho' \equiv_\alpha g'$$

For the "for some" case, let $\rho = \delta\circ\rho'$. Then

$g_1\rho\{^{\Phi_1\rho\cup\Phi}_{\Delta_1\rho\cup\Delta}$

$=$ ⟨definition of ρ⟩

$g_1(\delta\circ\rho')\{^{\Phi_1(\delta\circ\rho')\cup\Phi}_{\Delta_1(\delta\circ\rho')\cup\Delta}$

[2] It is interesting to observe that Lemmas B.4.2 and B.4.3 seem to have the $\exists/\forall$ property discussed by Gabbay and Pitts [GP99] for formulas involving the choice of fresh names. They introduce the quantifier "И" to capture this notion.

$\equiv_\alpha$ $\langle$Lemma 4.1.8 and CONG-SCP$\rangle$

$g_1\rho'\delta\{^{\Phi_1\rho'\delta\cup\Phi}_{\Delta_1\rho'\delta\cup\Delta}$

$\equiv_\alpha$ $\langle$properties of ρ', Lemma 4.1.9 and CONG-SCP$\rangle$

$g'\delta\{^{\Phi'\delta\cup\Phi}_{\Delta'\delta\cup\Delta}$

For the "for any" case, say that ρ has the required properties. Let $F = \mathrm{FN}(\mathrm{Flatten}(K\{^{\Phi}_{\Delta}))$. We use a ρ^α-construction. Let

$$\rho^\alpha = 1|_F + (\delta \circ \rho' \circ \rho^{-1})\big|_{\Phi_1\rho} + 1|_\Phi$$

We can use ρ^α to show that

$$g_1\rho\{^{\Phi_1\rho\cup\Phi}_{\Delta_1\rho\cup\Delta} \equiv_\alpha g'\delta\{^{\Phi'\delta\cup\Phi}_{\Delta'\delta\cup\Delta}$$

as required. ■

Lemma B.4.3 *Say that*

$$\begin{aligned} \mathrm{Flatten}(K_1) &\equiv_\alpha g_1\{^{\Phi_1}_{\Delta_1} \\ \mathrm{Flatten}(K_2) &\equiv_\alpha g_2\{^{\Phi_2}_{\Delta_2} \end{aligned}$$

Then

$$\mathrm{Flatten}(K_1|K_2) \equiv_\alpha (g_1\rho_1|g_2\rho_2)\{^{\Phi_1\rho_1\cup\Phi_2\rho_2}_{\Delta_1\rho_1\cup\Delta_2\rho_2}$$

for some/for any ρ_1 and ρ_2 satisfying

$$\begin{aligned} \mathrm{FN}(g_1\{^{\Phi_1}_{\Delta_1}) &\pitchfork \mathrm{Dom}(\Phi_2\rho_2) \\ \mathrm{Dom}(\Phi_1\rho_1) &\pitchfork \mathrm{FN}(g_2\rho_2) \\ \mathrm{Dom}(\Phi_1\rho_1) &\pitchfork \mathrm{Dom}(\Phi_2\rho_2) \\ \mathrm{Dom}(\Phi_1\rho_1) &\pitchfork \mathrm{FN}(\Delta_2\rho_2) \end{aligned}$$

Proof: Say

$$\begin{aligned} \mathrm{Flatten}(K_1) &= g'_1\{^{\Phi'_1}_{\Delta'_1} \\ \mathrm{Flatten}(K_2) &= g'_2\{^{\Phi'_2}_{\Delta'_2} \\ \mathrm{Flatten}(K_1|K_2) &= (g'_1\delta_1|g'_2\delta_2)\{^{\Phi'_1\delta_1\cup\Phi'_2\delta_2}_{\Delta'_1\delta_1\cup\Delta'_2\delta_2} \end{aligned}$$

Now, $g'_1\{^{\Phi'_1}_{\Delta'_1} \equiv_\alpha g_1\{^{\Phi_1}_{\Delta_1}$ and $g'_2\{^{\Phi'_2}_{\Delta'_2} \equiv_\alpha g_2\{^{\Phi_2}_{\Delta_2}$ so, by Lemma B.3.12, there are renamings ρ'_1 and ρ'_2 such that

$$\begin{array}{rclcrcl} \Phi_1\rho'_1 & = & \Phi'_1 & \quad & \Phi_2\rho'_2 & = & \Phi'_2 \\ \Delta_1\rho'_1 & = & \Delta'_1 & & \Delta_2\rho'_2 & = & \Delta'_2 \\ g_1\rho'_1 & \equiv_\alpha & g'_1 & & g_2\rho'_2 & \equiv_\alpha & g'_2 \end{array}$$

For the "for some" case, let $\rho_1 = \delta_1 \circ \rho'_1$ and $\rho_2 = \delta_2 \circ \rho'_2$. Then

$$\begin{array}{ll} & (g_1\rho_1|g_2\rho_2)\{^{\Phi_1\rho_1\cup\Phi_2\rho_2}_{\Delta_1\rho_1\cup\Delta_2\rho_2} \\ = & \langle \text{definition of } \rho_1 \text{ and } \rho_2 \rangle \\ & (g_1(\delta_1 \circ \rho'_1)|g_2(\delta_2 \circ \rho'_2))\{^{\Phi_1(\delta_1\circ\rho_1)\cup\Phi_2(\delta_2\circ\rho_2)}_{\Delta_1(\delta_1\circ\rho_1)\cup\Delta_2(\delta_2\circ\rho_2)} \\ \equiv_\alpha & \langle \text{Lemma 4.1.8, CONG-PAR and CONG-SCP} \rangle \\ & (g_1\rho'_1\delta_1|g_2\rho'_2\delta_2)\{^{\Phi_1\rho_1\delta_1\cup\Phi_2\rho_2\delta_2}_{\Delta_1\rho_1\delta_1\cup\Delta_2\rho_2)\delta_2} \\ \equiv_\alpha & \langle \text{Lemma 4.1.9, CONG-PAR and CONG-SCP} \rangle \\ & (g'_1\delta_1|g'_2\delta_2)\{^{\Phi'_1\delta_1\cup\Phi'_2\delta_2}_{\Delta'_1\delta_1\cup\Delta'_2\delta_2} \end{array}$$

For the "for any" case, say that ρ_1 and ρ_2 have the required properties. Let $F = \text{FN}(\text{Flatten}(K_1|K_2))$. We use a ρ^α-construction. Let

$$\rho^\alpha = 1|_F + (\delta_1 \circ \rho'_1 \circ {\rho_1}^{-1})\big|_{\Phi_1\rho_1} + (\delta_2 \circ \rho'_2 \circ {\rho_2}^{-1})\big|_{\Phi_2\rho_2}$$

We can use ρ^α to show that

$$(g_1\rho_1|g_2\rho_2)\{^{\Phi_1\rho_1\cup\Phi_2\rho_2}_{\Delta_1\rho_1\cup\Delta_2\rho_2} \equiv_\alpha (g'_1\delta_1|g'_2\delta_2)\{^{\Phi'_1\delta_1\cup\Phi'_2\delta_2}_{\Delta'_1\delta_1\cup\Delta'_2\delta_2}$$

as required. ■

4.3.2
p119

Lemma B.4.4 *For a configuration K and a renaming ρ,*

$$\text{Flatten}(K\rho) \equiv_\alpha \text{Flatten}(K)\rho$$

Proof: We use an induction on the structure of K. This is obvious for the case where $K = \text{Nil}$.

Consider the case where $K = o[p]$. Then

$$\begin{array}{ll} & \text{Flatten}(o[p]\rho) \\ = & \quad \langle \text{renaming procedure} \rangle \\ & \text{Flatten}((o\rho)[p\rho]) \\ = & \quad \langle \text{Flatten algorithm} \rangle \\ & (o\rho)[p\rho]\{^{\emptyset}_{\emptyset} \\ = & \quad \langle \text{renaming procedure} \rangle \\ & o[p]\rho\{^{\emptyset}_{\emptyset} \\ = & \quad \langle \text{renaming procedure} \rangle \\ & (o[p]\{^{\emptyset}_{\emptyset})\rho \\ = & \quad \langle \text{Flatten algorithm} \rangle \\ & \text{Flatten}(o[p])\rho \end{array}$$

as required in this case.

Consider the case where $K = K'\{^{\Phi}_{\Delta}$. Say $\text{Flatten}(K') = g'\{^{\Phi'}_{\Delta'}$. Then

$$\text{Flatten}(K) = g'\delta\{^{\Phi'\delta \cup \Phi}_{\Delta'\delta \cup \Delta}$$

so

$$\begin{array}{ll} & \text{Flatten}(K)\rho \\ = & \quad \langle \text{by above} \rangle \\ & (g'\delta\{^{\Phi'\delta \cup \Phi}_{\Delta'\delta \cup \Delta})\rho \\ = & \quad \langle \text{renaming procedure, choosing } \delta_1 \text{ appropriately} \rangle \\ & g'\delta\delta_1\rho\{^{(\Phi'\delta \cup \Phi)\delta_1\rho}_{(\Delta'\delta \cup \Delta)\delta_1\rho} \end{array}$$

Now

$$\begin{array}{ll} & (K'\{^{\Phi}_{\Delta})\rho \\ = & \quad \langle \text{renaming procedure, choosing } \delta_2 \text{ appropriately} \rangle \\ & K'\delta_2\rho\{^{\Phi\delta_2\rho}_{\Delta\delta_2\rho} \end{array}$$

and

$$\begin{array}{ll} & \text{Flatten}(K'\delta_2\rho) \\ \equiv_\alpha & \langle \text{induction hypothesis twice} \rangle \\ & \text{Flatten}(K')\delta_2\rho \\ \equiv_\alpha & \langle \text{Lemma B.3.7} \rangle \\ & \text{Flatten}(K')(\rho \circ \delta_2) \\ = & \langle \text{assumption} \rangle \\ & (g'\{^{\Phi'}_{\Delta'})(\rho \circ \delta_2) \\ = & \langle \text{renaming procedure, choosing } \delta_3 \text{ appropriately} \rangle \\ & g'\delta_3(\rho \circ \delta_2)\{^{\Phi'\delta_3(\rho\circ\delta_2)}_{\Delta'\delta_3(\rho\circ\delta_2)} \end{array}$$

Therefore, by Lemma B.4.2, we know that there is some renaming ρ' such that

$$\text{Flatten}(K'\delta_2\rho\{^{\Phi\delta_2\rho}_{\Delta\delta_2\rho}) \equiv_\alpha g'\delta_3(\rho \circ \delta_2)\rho'\{^{\Phi'\delta_3(\rho\circ\delta_2)\rho' \cup \Phi\delta_2\rho}_{\Delta'\delta_3(\rho\circ\delta_2)\rho' \cup \Delta\delta_2\rho}$$

We use a ρ^α-construction. Let $F = \text{FN}(K'\{^{\Phi}_{\Delta})$ and

$$\begin{array}{rcl} \rho^\alpha & = & 1|_{F\rho} \\ & & +(\rho \circ \delta_1 \circ \delta \circ \delta_3 \circ \delta_2 \circ \rho^{-1} \circ \rho'^{-1})\big|_{\Phi'\delta_3(\rho\circ\delta_2)\rho'} \\ & & +(\rho \circ \delta_1 \circ \delta_2 \circ \rho^{-1})|_{\Phi\delta_2\rho} \end{array}$$

As before, we can use ρ^α to show that

$$g'\delta\delta_1\rho\{^{(\Phi'\delta\cup\Phi)\delta_1\rho}_{(\Delta'\delta\cup\Delta)\delta_1\rho} \equiv_\alpha g'\delta_3(\rho \circ \delta_2)\rho'\{^{\Phi'\delta_3(\rho\circ\delta_2)\rho' \cup \Phi\delta_2\rho}_{\Delta'\delta_3(\rho\circ\delta_2)\rho' \cup \Delta\delta_2\rho}$$

from which the required result is easily obtained.

Consider the case where $K = (K_1|K_2)$ and

$$\begin{array}{rcl} \text{Flatten}(K_1) & = & g_1\{^{\Phi_1}_{\Delta_1} \\ \text{Flatten}(K_2) & = & g_2\{^{\Phi_2}_{\Delta_2} \end{array}$$

Then

$$\text{Flatten}(K_1|K_2) = (g_1\delta_1|g_2\delta_2)\{^{\Phi_1\delta_1 \cup \Phi_2\delta_2}_{\Delta_1\delta_1 \cup \Delta_2\delta_2}$$

so

$$\begin{aligned} \text{Flatten}(K_1|K_2)\rho &= ((g_1\delta_1|g_2\delta_2)\{^{\Phi_1\delta_1\cup\Phi_2\delta_2}_{\Delta_1\delta_1\cup\Delta_2\delta_2})\rho \\ &= (g_1\delta_1|g_2\delta_2)\delta\rho\{^{(\Phi_1\delta_1\cup\Phi_2\delta_2)\delta\rho}_{(\Delta_1\delta_1\cup\Delta_2\delta_2)\delta\rho} \end{aligned}$$

Now

$$(K_1|K_2)\rho = K_1\rho|K_2\rho$$

So

$$\begin{aligned} & \text{Flatten}(K_1\rho) \\ \equiv_\alpha \quad & \langle \text{induction hypothesis} \rangle \\ & (g_1\{^{\Phi_1}_{\Delta_1})\rho \\ = \quad & \langle \text{renaming procedure, choosing } \delta_1' \text{ appropriately} \rangle \\ & g_1\delta_1'\rho\{^{\Phi_1\delta_1'\rho}_{\Delta_1\delta_1'\rho} \end{aligned}$$

and

$$\begin{aligned} & \text{Flatten}(K_2\rho) \\ \equiv_\alpha \quad & \langle \text{induction hypothesis} \rangle \\ & (g_2\{^{\Phi_2}_{\Delta_2})\rho \\ = \quad & \langle \text{renaming procedure, choosing } \delta_2' \text{ appropriately} \rangle \\ & g_2\delta_2'\rho\{^{\Phi_2\delta_2'\rho}_{\Delta_2\delta_2'\rho} \end{aligned}$$

Therefore, by Lemma B.4.3, we can chose two renamings ρ_1 and ρ_2 with appropriate properties such that

$$\text{Flatten}((K_1|K_2)\rho) \equiv_\alpha (g_1\delta_1'\rho\rho_1|g_2\delta_2'\rho\rho_2)\{^{\Phi_1\delta_1'\rho\rho_1\cup\Phi_2\delta_2'\rho\rho_2}_{\Delta_1\delta_1'\rho\rho_1\cup\Delta_2\delta_2'\rho\rho_2}$$

We use a ρ^α-construction. Let $F = \text{FN}(K_1|K_2)$ and

$$\begin{aligned} \rho^\alpha \quad = \quad & 1|_{F\rho} \\ & +(\rho\circ\delta\circ\delta_1\circ\delta_1'\circ\rho^{-1}\circ{\rho_1}^{-1})|_{\Phi_1\delta_1'\rho\rho_1} \\ & +(\rho\circ\delta\circ\delta_2\circ\delta_2'\circ\rho^{-1}\circ{\rho_2}^{-1})|_{\Phi_2\delta_2'\rho\rho_2} \end{aligned}$$

We use ρ^α in the typical way to show

$$(g_1\delta_1|g_2\delta_2)\delta\rho\{^{(\Phi_1\delta_1\cup\Phi_2\delta_2)\delta\rho}_{(\Delta_1\delta_1\cup\Delta_2\delta_2)\delta\rho}\equiv_\alpha (g_1\delta'_1\rho\rho_1|g_2\delta'_2\rho\rho_2)\{^{\Phi_1\delta'_1\rho\rho_1\cup\Phi_2\delta'_2\rho\rho_2}_{\Delta_1\delta'_1\rho\rho_1\cup\Delta_2\delta'_2\rho\rho_2}$$

from which the required result is easily obtained. ■

4.3.3 p119

Theorem B.4.5 *Say that K_1 and K_2 are configurations such that $K_1 \equiv_\alpha K_2$, and that*

$$\begin{aligned} \text{Flatten}(K_1) &= g_1\{^{\Phi_1}_{\Delta_1} \\ \text{Flatten}(K_2) &= g_2\{^{\Phi_2}_{\Delta_2} \end{aligned}$$

Then, for some renaming ρ such that $\text{FN}(g_2\{^{\Phi_2}_{\Delta_2}) \pitchfork \text{Change}(\rho)$ we have

$$\Phi_2\rho = \Phi_1 \qquad \Delta_2\rho = \Delta_1 \qquad g_2\rho \equiv_\alpha g_1$$

Proof: We use an induction on the proof tree of $K_1 \equiv_\alpha K_2$. The case for EQV-RFL is obvious.

Consider the case where the last step of the proof tree is an instance of EQV-SYM:

$$\frac{K_2 \equiv_\alpha K_1}{K_1 \equiv_\alpha K_2}$$

By the induction hypothesis there is some renaming, ρ, such that

$$\Phi_1\rho = \Phi_2 \qquad \Delta_1\rho = \Delta_2 \qquad g_1\rho \equiv_\alpha g_2$$

and

$$\text{FN}(g_1\{^{\Phi_1}_{\Delta_1}) \pitchfork \text{Change}(\rho)$$

Well, $\text{FN}(g_1\{^{\Phi_1}_{\Delta_1}) = \text{FN}(g_2\{^{\Phi_2}_{\Delta_2})$ using results B.3.4, B.3.13 and B.4.1. Also $\text{Change}(\rho) = \text{Change}(\rho^{-1})$, so

$$\text{FN}(g_2\{^{\Phi_2}_{\Delta_2}) \pitchfork \text{Change}(\rho^{-1})$$

Now

$$\begin{array}{rclclcl} \Phi_1\rho^{-1} & = & (\Phi_2\rho)\rho^{-1} & = & \Phi_2(\rho^{-1}\circ\rho) & = & \Phi_2 \\ \Delta_1\rho^{-1} & = & (\Delta_2\rho)\rho^{-1} & = & \Delta_2(\rho^{-1}\circ\rho) & = & \Delta_2 \end{array}$$

and, using lemmas 4.1.9 and 4.1.8,

$$g_1\rho^{-1} \equiv_\alpha (g_2\rho)\rho^{-1} \equiv_\alpha g_2(\rho^{-1}\circ\rho) = g_2$$

Therefore ρ^{-1} can be our required renaming.

Say the last step of the proof tree is an instance of EQV-TRN:

$$\frac{K_1 \equiv_\alpha K \qquad K \equiv_\alpha K_2}{K_1 \equiv_\alpha K_2}$$

and say that $\text{Flatten}(K) = g\{^{\Phi}_{\Delta}$. Then, by the induction hypothesis, there are renamings ρ_1 and ρ_2 such that

$$\begin{array}{rclcrclcrcl} \Phi\rho_1 & = & \Phi_1 & \quad & \Delta\rho_1 & = & \Delta_1 & \quad & g\rho_1 & \equiv_\alpha & g_1 \\ \Phi_2\rho_2 & = & \Phi & & \Delta_2\rho_2 & = & \Delta & & g_2\rho_2 & \equiv_\alpha & g \end{array}$$

and

$$\begin{array}{rcl} \text{FN}(g\{^{\Phi}_{\Delta}) & \pitchfork & \text{Change}(\rho_1) \\ \text{FN}(g_2\{^{\Phi_2}_{\Delta_2}) & \pitchfork & \text{Change}(\rho_2) \end{array}$$

Again, by results B.3.4, B.3.13 and B.4.1 we have, $\text{FN}(g_2\{^{\Phi_2}_{\Delta_2}) = \text{FN}(g\{^{\Phi}_{\Delta})$ so

$$\text{FN}(g_2\{^{\Phi_2}_{\Delta_2}) \pitchfork \text{Change}(\rho_1) \cup \text{Change}(\rho_2)$$

Hence

$$\text{FN}(g_2\{^{\Phi_2}_{\Delta_2}) \pitchfork \text{Change}(\rho_1 \circ \rho_2)$$

Now

$$\begin{array}{rclclcl} \Phi_2(\rho_1\circ\rho_2) & = & (\Phi_2\rho_2)\rho_1 & = & \Phi\rho_1 & = & \Phi_1 \\ \Delta_2(\rho_1\circ\rho_2) & = & (\Delta_2\rho_2)\rho_1 & = & \Delta\rho_1 & = & \Delta_1 \\ g_2(\rho_1\circ\rho_2) & \equiv_\alpha & (g_2\rho_2)\rho_1 & \equiv_\alpha & g\rho_1 & \equiv_\alpha & g_1 \end{array}$$

Therefore $\rho_1 \circ \rho_2$ can be our required renaming.

Consider the case where the last step of the tree proof is an instance of CONG-PAR:

$$\frac{K_1' \equiv_\alpha K_2' \qquad K_1'' \equiv_\alpha K_2''}{(K_1'|K_1'') \equiv_\alpha (K_2'|K_2'')}$$

Say

$$\begin{array}{rcl}
\text{Flatten}(K_1') & = & g_1'\{^{\Phi_1'}_{\Delta_1'} \\
\text{Flatten}(K_2') & = & g_2'\{^{\Phi_2'}_{\Delta_2'} \\
\text{Flatten}(K_1'') & = & g_1''\{^{\Phi_1''}_{\Delta_1''} \\
\text{Flatten}(K_2'') & = & g_2''\{^{\Phi_2''}_{\Delta_2''} \\
\text{Flatten}(K_1'|K_1'') & = & (g_1'\delta_1|g_1''\delta_1')\{^{\Phi_1'\delta_1 \cup \Phi_1''\delta_1'}_{\Delta_1'\delta_1 \cup \Delta_1''\delta_1'} \\
\text{Flatten}(K_2'|K_2'') & = & (g_2'\delta_2|g_2''\delta_2')\{^{\Phi_2'\delta_2 \cup \Phi_2''\delta_2'}_{\Delta_2'\delta_2 \cup \Delta_2''\delta_2'}
\end{array}$$

By the induction hypothesis, there are renamings ρ and ρ' such that

$$\begin{array}{rclcrclcrcl}
\Phi_2'\rho & = & \Phi_1' & \quad & \Delta_2'\rho & = & \Delta_1' & \quad & g_2'\rho & \equiv_\alpha & g_1' \\
\Phi_2''\rho' & = & \Phi_1'' & \quad & \Delta_2''\rho' & = & \Delta_1'' & \quad & g_2''\rho' & \equiv_\alpha & g_1''
\end{array}$$

Let $F = \text{FN}(\text{Flatten}(K_2'|K_2''))$ and let

$$\rho^\alpha = 1|_F + (\delta_1 \circ \rho \circ \delta_2)|_{\Phi_2'\delta_2} + (\delta_1' \circ \rho' \circ \delta_2')|_{\Phi_2''\delta_2'}$$

Then $F \pitchfork \text{Change}(\rho^\alpha)$ and we can show

$$\begin{array}{rcl}
(g_2'\delta_2|g_2''\delta_2')\rho^\alpha & \equiv_\alpha & g_1'\delta_1|g_1''\delta_1' \\
(\Phi_2'\delta_2 \cup \Phi_2''\delta_2')\rho^\alpha & = & \Phi_1'\delta_1 \cup \Phi_1''\delta_1' \\
(\Delta_2'\delta_2 \cup \Delta_2''\delta_2')\rho^\alpha & = & \Delta_1'\delta_1 \cup \Delta_1''\delta_1'
\end{array}$$

which is sufficient for the result in this case.

Say the last step of the proof tree is an instance of CONG-SCP:

$$\frac{K_1' \equiv_\alpha K_2'}{K_1'\{^\Phi_\Delta \equiv_\alpha K_2'\{^\Phi_\Delta}$$

Say

$$\begin{aligned}
\text{Flatten}(K_1') &= g_1\{^{\Phi_1}_{\Delta_1} \\
\text{Flatten}(K_2') &= g_2\{^{\Phi_2}_{\Delta_2} \\
\text{Flatten}(K_1'\{^{\Phi}_{\Delta}) &= g_1\delta_1\{^{\Phi_1\delta_1\cup\Phi}_{\Delta_1\delta_1\cup\Delta} \\
\text{Flatten}(K_2'\{^{\Phi}_{\Delta}) &= g_2\delta_2\{^{\Phi_2\delta_2\cup\Phi}_{\Delta_2\delta_2\cup\Delta}
\end{aligned}$$

By the induction hypothesis, there is some renaming ρ such that

$$\Phi_2\rho = \Phi_1 \qquad \Delta_2\rho = \Delta_1 \qquad g_2\rho \equiv_\alpha g_1$$

Let $F = \text{FN}(\text{Flatten}(K_2'\{^{\Phi}_{\Delta}))$ and let

$$\rho^\alpha = 1|_F + (\delta_1 \circ \rho \circ \delta_2)|_{\Phi_2\delta_2} + 1|_\Phi$$

Then $F \pitchfork \text{Change}(\rho^\alpha)$ and

$$(\Phi_2\delta_2 \cup \Phi)\rho^\alpha = \Phi_1\delta_1 \cup \Phi \qquad (\Delta_2\delta_2 \cup \Delta)\rho^\alpha = \Delta_1\delta_1 \cup \Delta \qquad g_2\delta_2\rho^\alpha \equiv_\alpha g_1\delta_1$$

as required in this case.

Say the last step of the proof tree is an instance of ALPHA-COD:

$$\frac{p_1 \equiv_\alpha p_2}{o[p_1] \equiv_\alpha o[p_2]}$$

Then

$$\begin{aligned}
\text{Flatten}(o[p_1]) &= o[p_1]\{^{\emptyset}_{\emptyset} \\
\text{Flatten}(o[p_2]) &= o[p_2]\{^{\emptyset}_{\emptyset}
\end{aligned}$$

The identity renaming will work in this case.

Say the tree proof is an instance of CHNG-SCP:

$$K\{^{\Phi}_{\Delta} \equiv_\alpha K\rho'\{^{\Phi\rho'}_{\Delta\rho'}$$

Say that

$$\begin{aligned}
\text{Flatten}(K) &= g'\{^{\Phi'}_{\Delta'} \\
\text{Flatten}(K\{^{\Phi}_{\Delta}) &= g'\delta\{^{\Phi'\delta\cup\Phi}_{\Delta'\delta\cup\Delta}
\end{aligned}$$

By Lemma B.4.4

$$\mathrm{Flatten}(K\rho') \equiv_\alpha (g'\{^{\Phi'}_{\Delta'})\rho' = g'\delta'\rho'\{^{\Phi'\delta'\rho'}_{\Delta'\delta'\rho'}$$

so, by Lemma B.4.2, we can find ρ'' such that

$$\mathrm{Flatten}(K\rho'\{^{\Phi\rho'}_{\Delta\rho'}) \equiv_\alpha g'\delta'\rho'\rho''\{^{\Phi'\delta'\rho'\rho''\cup\Phi\rho'}_{\Delta'\delta'\rho'\rho''\cup\Delta\rho'}$$

Let $F = \mathrm{FN}(\mathrm{Flatten}(K\rho'\{^{\Phi\rho'}_{\Delta\rho'}))$ and let

$$\rho^\alpha = 1|_F + (\delta \circ \delta' \circ \rho'^{-1} \circ \rho''^{-1})\Big|_{\Phi'\delta'\rho'\rho''} + \rho'^{-1}\Big|_{\Phi\rho'}$$

We can use ρ^α to show that

$$\begin{aligned}(\Phi'\delta'\rho'\rho'' \cup \Phi\rho')\rho^\alpha &= \Phi'\delta \cup \Phi \\ (\Delta'\delta'\rho'\rho'' \cup \Delta\rho')\rho^\alpha &= \Delta'\delta \cup \Delta \\ (g'\delta'\rho'\rho'')\rho^\alpha &\equiv_\alpha g'\delta\end{aligned}$$

as required in this case. ■

4.3.4 p119

Theorem B.4.6 *Say K_1 and K_2 are configurations such that $K_1 \overset{\blacktriangleright}{\equiv} K_2$ and that*

$$\begin{aligned}\mathrm{Flatten}(K_1) &= g_1\{^{\Phi_1}_{\Delta_1} \\ \mathrm{Flatten}(K_2) &= g_2\{^{\Phi_2}_{\Delta_2}\end{aligned}$$

Then, for some renaming ρ such that $\mathrm{FN}(g_2\{^{\Phi}_{\Delta}) \pitchfork \mathrm{Change}(\rho)$ we have

$$\Phi_2\rho = \Phi_1 \qquad \Delta_2\rho = \Delta_1 \qquad g_2\rho \overset{\triangleright}{\equiv}_\alpha g_1$$

Proof: We use an induction on the proof of $K_1 \overset{\blacktriangleright}{\equiv} K_2$. As many of the cases can be approached exactly as in Theorem B.4.5, we only consider those cases which differ.

Say that the last step of the proof tree is ALPHA. Then the result follows directly from Theorem B.4.5.

Say that the proof tree is an instance of PAR-ID:

$$K|\text{Nil} \overset{\blacktriangleright}{\equiv} K$$

and that $\text{Flatten}(K) = g\{^{\Phi}_{\Delta}$. Then

$$\text{Flatten}(K|\text{Nil}) = (g|\text{nil})\{^{\Phi}_{\Delta}$$

We can use the identity renaming, since $(g|\text{nil}) \overset{\triangleright}{\equiv}_\alpha g$.

Say that the proof tree is an instance of PAR-ABL:

$$K_1|K_2 \overset{\blacktriangleright}{\equiv} K_2|K_1$$

Say

$$\begin{array}{rcl}
\text{Flatten}(K_1) & = & g_1\{^{\Phi_1}_{\Delta_1} \\
\text{Flatten}(K_2) & = & g_2\{^{\Phi_2}_{\Delta_2} \\
\text{Flatten}(K_1|K_2) & = & (g_1\delta_1|g_2\delta_2)\{^{\Phi_1\delta_1 \cup \Phi_2\delta_2}_{\Delta_1\delta_1 \cup \Delta_2\delta_2} \\
\text{Flatten}(K_2|K_1) & = & (g_2\delta_2'|g_1\delta_1')\{^{\Phi_2\delta_2' \cup \Phi_1\delta_1'}_{\Delta_2\delta_2' \cup \Delta_1\delta_1'}
\end{array}$$

Let $F = \text{FN}(K_2|K_1)$ and let

$$\rho^\alpha = 1|_F + (\delta_2 \circ \delta_2')|_{\Phi_2\delta_2'} + (\delta_1 \circ \delta_1')|_{\Phi_1\delta_1'}$$

Then

$$(g_2\delta_2'|g_1\delta_1')\rho^\alpha \equiv_\alpha (g_2\delta_2|g_1\delta_1) \overset{\triangleright}{\equiv}_\alpha (g_2\delta_2|g_1\delta_1)$$

and

$$\begin{array}{rcl}
(\Phi_2\delta_2' \cup \Phi_1\delta_1')\rho^\alpha & = & \Phi_1\delta_1 \cup \Phi_2\delta_2 \\
(\Delta_2\delta_2' \cup \Delta_1\delta_1')\rho^\alpha & = & \Delta_1\delta_1 \cup \Delta_2\delta_2
\end{array}$$

Say the tree proof is an instance of PAR-ASS:

$$(K_1|K_2)|K_3 \overset{\blacktriangleright}{\equiv} K_1|(K_2|K_3)$$

Say

$$\begin{array}{rcl}
\text{Flatten}(K_1) & = & g_1\{^{\Phi_1}_{\Delta_1} \\
\text{Flatten}(K_2) & = & g_2\{^{\Phi_2}_{\Delta_2} \\
\text{Flatten}(K_3) & = & g_3\{^{\Phi_3}_{\Delta_3} \\
\text{Flatten}(K_1|K_2) & = & (g_1\delta_1|g_2\delta_2)\{^{\Phi_1\delta_1\cup\Phi_2\delta_2}_{\Delta_1\delta_1\cup\Delta_2\delta_2} \\
\text{Flatten}((K_1|K_2)|K_3) & = & ((g_1\delta_1|g_2\delta_2)\delta_{12}|g_3\delta_3)\{^{(\Phi_1\delta_1\cup\Phi_2\delta_2)\delta_{12}\cup\Phi_3\delta_3}_{(\Delta_1\delta_1\cup\Delta_2\delta_2)\delta_{12}\cup\Delta_3\delta_3} \\
\text{Flatten}(K_2|K_3) & = & (g_2\delta_2'|g_3\delta_3')\{^{\Phi_2\delta_2'\cup\Phi_3\delta_3'}_{\Delta_2\delta_2'\cup\Delta_3\delta_3'} \\
\text{Flatten}(K_1|(K_2|K_3)) & = & (g_1\delta_1'|(g_2\delta_2'|g_3\delta_3')\delta_{23}')\{^{\Phi_1\delta_1'\cup(\Phi_2\delta_2'\cup\Phi_3\delta_3')\delta_{23}'}_{\Delta_1\delta_1'\cup(\Delta_2\delta_2'\cup\Delta_3\delta_3')\delta_{23}'}
\end{array}$$

Let $F = \text{FN}(K_1|(K_2|K_3))$ and let

$$\begin{array}{rcl}
\rho^\alpha & = & 1|_F \\
& & +(\delta_{12}\circ\delta_1\circ\delta_1')|_{\Phi_1\delta_1'} \\
& & +(\delta_{12}\circ\delta_2\circ\delta_2'\circ\delta_{23}')|_{\Phi_2\delta_2'\delta_{23}'} \\
& & +(\delta_3\circ\delta_3'\circ\delta_{23}')|_{\Phi_3\delta_3'\delta_{23}'}
\end{array}$$

Then

$$\begin{array}{rcl}
(g_1\delta_1'|(g_2\delta_2'|g_3\delta_3')\delta_{23}')\rho^\alpha & \overset{\triangleright}{\equiv}_\alpha & (g_1\delta_1|g_2\delta_2)\delta_{12}|g_3\delta_3 \\
(\Phi_1\delta_1'\cup(\Phi_2\delta_2'\cup\Phi_3\delta_3')\delta_{23}')\rho^\alpha & = & (\Phi_1\delta_1\cup\Phi_2\delta_2)\delta_{12}\cup\Phi_3\delta_3 \\
(\Delta_1\delta_1'\cup(\Delta_2\delta_2'\cup\Delta_3\delta_3')\delta_{23}')\rho^\alpha & = & (\Delta_1\delta_1\cup\Delta_2\delta_2)\delta_{12}\cup\Delta_3\delta_3
\end{array}$$

as required in this case.

If the proof tree is an instance of EMPTY, then the identity renaming will work, since $\text{Flatten}(K\{^\emptyset_\emptyset) = \text{Flatten}(K)$.

Say that the proof tree is an instance of FLATTEN:

$$K\{^{\Phi_1}_{\Delta_1}\{^{\Phi_2}_{\Delta_2} \overset{\blacktriangleright}{\equiv} K\{^{\Phi_1\cup\Phi_2}_{\Delta_1\cup\Delta_2}$$

Say that

$$\begin{array}{rcl}
\text{Flatten}(K) & = & g\{^\Phi_\Delta \\
\text{Flatten}(K\{^{\Phi_1}_{\Delta_1}) & = & g\delta_1\{^{\Phi\delta_1\cup\Phi_1}_{\Delta\delta_1\cup\Delta_1} \\
\text{Flatten}(K\{^{\Phi_1}_{\Delta_1}\{^{\Phi_2}_{\Delta_2}) & = & g\delta_1\delta_2\{^{(\Phi\delta_1\cup\Phi_1)\delta_2\cup\Phi_2}_{(\Delta\delta_1\cup\Delta_1)\delta_2\cup\Delta_2} \\
\text{Flatten}(K\{^{\Phi_1\cup\Phi_2}_{\Delta_1\cup\Delta_2}) & = & g\delta'\{^{\Phi\delta'\cup\Phi_1\cup\Phi_2}_{\Delta\delta'\cup\Delta_1\cup\Delta_2}
\end{array}$$

Let $F = \mathrm{FN}(K\{^{\Phi_1 \cup \Phi_2}_{\Delta_1 \cup \Delta_2})$ and let

$$\rho^\alpha = 1|_F + (\delta_2 \circ \delta_1 \circ \delta')|_{\Phi\delta'} + \delta_2|_{\Phi_1} + 1|_{\Phi_2}$$

Then

$$\begin{aligned} g\delta'\rho^\alpha &\overset{\triangleright}{\equiv}_\alpha g\delta_1\delta_2 \\ (\Phi\delta' \cup \Phi_1 \cup \Phi_2)\rho^\alpha &= (\Phi\delta_1 \cup \Phi_1)\delta_2 \cup \Phi_2 \\ (\Delta\delta' \cup \Delta_1 \cup \Delta_2)\rho^\alpha &= (\Delta\delta_1 \cup \Delta_1)\delta_2 \cup \Delta_2 \end{aligned}$$

as required in this case.

Say that the proof tree is an instance of FLATTEN:

$$K_1|(K_2\{^\Phi_\Delta) \overset{\blacktriangleright}{\equiv} (K_1|K_2)\{^\Phi_\Delta$$

Say

$$\begin{aligned} \mathrm{Flatten}(K_1) &= g_1\{^{\Phi_1}_{\Delta_1} \\ \mathrm{Flatten}(K_2) &= g_2\{^{\Phi_2}_{\Delta_2} \\ \mathrm{Flatten}(K_2\{^\Phi_\Delta) &= g_2\delta_2\{^{\Phi_2\delta_2 \cup \Phi}_{\Delta_2\delta_2 \cup \Delta} \\ \mathrm{Flatten}(K_1|(K_2\{^\Phi_\Delta)) &= (g_1\delta_1'|g_2\delta_2\delta_2')\{^{\Phi_1\delta_1' \cup (\Phi_2\delta_2 \cup \Phi)\delta_2'}_{\Delta_1\delta_1' \cup (\Delta_2\delta_2 \cup \Delta)\delta_2'} \\ \mathrm{Flatten}(K_1|K_2) &= g_1\delta_1''|g_2\delta_2''\{^{\Phi_1\delta_1'' \cup \Phi_2\delta_2''}_{\Delta_1\delta_1'' \cup \Delta_2\delta_2''} \\ \mathrm{Flatten}((K_1|K_2)\{^\Phi_\Delta) &= (g_1\delta_1''|g_2\delta_2'')\delta\{^{(\Phi_1\delta_1'' \cup \Phi_2\delta_2'')\delta \cup \Phi}_{(\Delta_1\delta_1'' \cup \Delta_2\delta_2'')\delta \cup \Delta} \end{aligned}$$

Let $F = \mathrm{FN}(K_1|K_2\{^\Phi_\Delta)$ and let

$$\rho^\alpha = 1|_F + (\delta_1' \circ \delta_1'' \circ \delta)|_{\Phi_1\delta_1''\delta} + (\delta_2' \circ \delta_2 \circ \delta_2'' \circ \delta)|_{\Phi_2\delta_2''\delta} + \delta_2'|_\Phi$$

Then

$$\begin{aligned} (g_1\delta_1''|g_2\delta_2'')\delta\rho^\alpha &\overset{\triangleright}{\equiv}_\alpha (g_1\delta_1'|g_2\delta_2\delta_2') \\ ((\Phi_1\delta_1'' \cup \Phi_2\delta_2'')\delta \cup \Phi)\rho^\alpha &= \Phi_1\delta_1' \cup (\Phi_2\delta_2 \cup \Phi)\delta_2' \\ ((\Delta_1\delta_1'' \cup \Delta_2\delta_2'')\delta \cup \Delta)\rho^\alpha &= \Delta_1\delta_1' \cup (\Delta_2\delta_2 \cup \Delta)\delta_2' \end{aligned}$$

as required in this case. ■

Lemmas B.4.7–B.4.10 are necessary for the proof of Theorem B.4.11. They give us an internal view of the shape of a subsystem which can perform

a labelled action. We only provide the proof of Lemma B.4.7 as the others are similar.

Lemma B.4.7 *Say K_1 and K_2 are configurations such that*

$$\Gamma, \Phi \blacktriangleright K_1 \xrightarrow{c!\langle v_1,\ldots v_n\rangle} K_2$$

and

$$\begin{array}{rcl} \text{Flatten}(K_1) & = & g_1\{^{\Phi_1}_{\Delta_1} \\ \text{Flatten}(K_2) & = & g_2\{^{\Phi_2}_{\Delta_2} \end{array}$$

then there is some renaming ρ such that $\text{FN}(g_2\{^{\Phi_2}_{\Delta_2}) \pitchfork \text{Change}(\rho)$ *and*

$$\begin{array}{rcl} \Phi_2\rho & = & \Phi_1 \\ \Delta_2\rho & = & \Delta_1 \\ g_1 & \overset{\triangleright}{\equiv} & o[c!\langle v_1,\ldots v_n\rangle\ p]||g \\ g_2\rho & \overset{\triangleright}{\equiv}_\alpha & o[p]||g \end{array}$$

Proof: We use an induction on the proof tree of

$$\Gamma, \Phi \blacktriangleright K_1 \xrightarrow{c!\langle v_1,\ldots v_n\rangle} K_2$$

Say that the proof tree is an instance of SND:

$$\Gamma, \Phi \blacktriangleright o[c!\langle v_1,\ldots v_n\rangle\ p] \xrightarrow{c!\langle v_1,\ldots v_n\rangle} o[p]$$

Well

$$\begin{array}{rcl} \text{Flatten}(o[c!\langle v_1,\ldots v_n\rangle\ p]) & = & o[c!\langle v_1,\ldots v_n\rangle\ p]\{^{\emptyset}_{\emptyset} \\ \text{Flatten}(o[p]) & = & o[p]\{^{\emptyset}_{\emptyset} \end{array}$$

then

$$\begin{array}{rcl} o[c!\langle v_1,\ldots v_n\rangle\ p] & \overset{\triangleright}{\equiv} & o[c!\langle v_1,\ldots v_n\rangle\ p]|\text{nil} \\ o[p] & \overset{\triangleright}{\equiv}_\alpha & o[p]|\text{nil} \end{array}$$

So, with ρ as the identity renaming, we have the result for this case.

Say that the last step of the proof tree is an instance of EQV-LFT:

$$\frac{K_1 \overset{\blacktriangleright}{\equiv} K_2 \qquad \Gamma, \Phi \blacktriangleright K_1 \xrightarrow[l_2]{l_1} K_1'}{\Gamma, \Phi \blacktriangleright K_2 \xrightarrow[l_2]{l_1} K_1'}$$

Say

$$\begin{array}{rcl} \text{Flatten}(K_1) & = & g_1\{^{\Phi_1}_{\Delta_1} \\ \text{Flatten}(K_1') & = & g_1'\{^{\Phi_1'}_{\Delta_1'} \\ \text{Flatten}(K_2) & = & g_2\{^{\Phi_2}_{\Delta_2} \end{array}$$

Since $K_1 \overset{\blacktriangleright}{\equiv} K_2$ we have $K_2 \overset{\blacktriangleright}{\equiv} K_1$. Thus, by Theorem B.4.6 there exists some renaming ρ such that

$$\Phi_1\rho = \Phi_2 \qquad \Delta_1\rho = \Delta_2 \qquad g_1\rho \overset{\triangleright}{\equiv}_\alpha g_2$$

Also, by the induction hypothesis, there is some renaming ρ' such that

$$\Phi_1'\rho' = \Phi_1 \qquad \Delta_1'\rho' = \Delta_1$$

and

$$\begin{array}{rcl} g_1 & \overset{\triangleright}{\equiv} & o[c!\langle v_1, \ldots v_n\rangle\ p]||g \\ g_1'\rho' & \overset{\triangleright}{\equiv}_\alpha & o[p]||g \end{array}$$

Then

$$\begin{array}{ll} & g_2 \\ \overset{\triangleright}{\equiv}_\alpha & \langle \text{choice of } \rho \rangle \\ & g_1\rho \\ \overset{\triangleright}{\equiv} & \langle \text{Lemma 4.1.6} \rangle \\ & (o[c!\langle v_1, \ldots v_n\rangle\ p]||g)\rho \\ = & \langle \text{renaming procedure; } \{c, v_1, \ldots v_n\} \pitchfork \text{Change}(\rho) \rangle \\ & (o\rho)[c!\langle v_1, \ldots v_n\rangle\ p\rho]||(g\rho) \end{array}$$

At this point, we'll need the following sublemma.

Sublemma 1 *Say*

$$g \overset{\triangleright}{\equiv}_\alpha o[c!\langle v_1, \ldots v_n\rangle \; p]|g_1$$

then

$$g \overset{\triangleright}{\equiv} o[c!\langle v_1, \ldots v_n\rangle \; p']|g_1'$$

where

$$p' \equiv_\alpha p \qquad\qquad g_1' \equiv_\alpha g_1$$

Proof of Sublemma: By Lemma B.2.4 we know

$$g \overset{\triangleright}{\equiv} g' \equiv_\alpha o[c!\langle v_1, \ldots v_n\rangle \; p]|g_1$$

and by the nature of $\equiv_\alpha$ system, we know that

$$g' = o[c!\langle v_1, \ldots v_n\rangle \; p']|g_1'$$

for some p' and g_1' such that

$$p' \equiv_\alpha p \qquad\qquad g_1' \equiv_\alpha g_1$$

as required. □

Thus, by this sublemma

$$g_2 \overset{\triangleright}{\equiv} (o\rho)[c!\langle v_1, \ldots v_n\rangle \; p']|g'$$

where

$$p' \equiv_\alpha p\rho \qquad\qquad g' \equiv_\alpha g\rho$$

Let $\rho^\alpha = \rho \circ \rho'$. Then

$$\Phi_1'\rho^\alpha = \Phi_2 \qquad\qquad \Delta_1'\rho^\alpha = \Delta_2$$

$$
\begin{array}{ll}
 & g_1'\rho^\alpha \\
\equiv_\alpha & \langle \text{Lemma 4.1.8} \rangle \\
 & g_1'\rho'\rho \\
\overset{\triangleright}{\equiv}_\alpha & \langle \text{choice of } \rho' \text{ and Lemma 4.1.11} \rangle \\
 & (o[p]||g)\rho \\
= & \langle \text{renaming procedure} \rangle \\
 & (o\rho)[p\rho]||(g\rho) \\
\equiv_\alpha & \langle \text{above, ALPHA-COD and CONG-PAR} \rangle \\
 & (o\rho)[p']||g'
\end{array}
$$

as required.

Say that the last step of the proof is an instance of EQV-RHT:

$$
\frac{K_1' \overset{\blacktriangleright}{\equiv} K_2 \qquad \Gamma, \Phi \blacktriangleright K_1 \xrightarrow[l_2]{l_1} K_1'}{\Gamma, \Phi \blacktriangleright K_1 \xrightarrow[l_2]{l_1} K_2}
$$

Say

$$
\begin{array}{rcl}
\text{Flatten}(K_1) & = & g_1\{^{\Phi_1}_{\Delta_1} \\
\text{Flatten}(K_1') & = & g_1'\{^{\Phi_1'}_{\Delta_1'} \\
\text{Flatten}(K_2) & = & g_2\{^{\Phi_2}_{\Delta_2}
\end{array}
$$

Since $K_1' \overset{\blacktriangleright}{\equiv} K_2$, Lemma B.4.6 means that there is some renaming ρ such that

$$
\Phi_2\rho = \Phi_1' \qquad \Delta_2\rho = \Delta_1' \qquad g_2\rho \overset{\triangleright}{\equiv}_\alpha g_1'
$$

By the induction hypothesis, applied to the right subtree, there is some renaming ρ' such that

$$
\begin{array}{rcl}
\Phi_1'\rho' & = & \Phi_1 \\
\Delta_1'\rho' & = & \Delta_1 \\
g_1 & \overset{\triangleright}{\equiv} & o[c!\langle v_1, \ldots v_n \rangle \; p]||g \\
g_1'\rho' & \overset{\triangleright}{\equiv}_\alpha & o[p]||g
\end{array}
$$

Let $\rho^\alpha = \rho' \circ \rho$. The result easily follows by using ρ^α.

Say that the last step of the proof tree is an instance of PAR:

$$\frac{\Gamma, \Phi \blacktriangleright K_1 \xrightarrow[l_2]{l_1} K_2}{\Gamma, \Phi \blacktriangleright K_1|K \xrightarrow[l_2]{l_1} K_2|K}$$

Say

$$\begin{aligned}
\text{Flatten}(K) &= g\{^{\Phi}_{\Delta} \\
\text{Flatten}(K_1) &= g_1\{^{\Phi_1}_{\Delta_1} \\
\text{Flatten}(K_2) &= g_2\{^{\Phi_2}_{\Delta_2} \\
\text{Flatten}(K_1|K) &= (g_1\delta_1|g\delta_1')\{^{\Phi_1\delta_1 \cup \Phi\delta_1'}_{\Delta_1\delta_1 \cup \Delta\delta_1'} \\
\text{Flatten}(K_2|K) &= (g_2\delta_2|g\delta_2')\{^{\Phi_2\delta_2 \cup \Phi\delta_2'}_{\Delta_2\delta_2 \cup \Delta\delta_2'}
\end{aligned}$$

By the induction hypothesis, there is some renaming ρ such that

$$\begin{aligned}
\Phi_2\rho &= \Phi_1 \\
\Delta_2\rho &= \Delta_1 \\
g_1 &\overset{\triangleright}{\equiv} o[c!\langle v_1, \dots v_n\rangle\ p]|g' \\
g_2\rho &\overset{\triangleright}{\equiv}_\alpha o[p]|g'
\end{aligned}$$

Let $F = \text{FN}(\text{Flatten}(K_2|K))$ and let

$$\rho^\alpha = 1|_F + (\delta_1 \circ \rho \circ {\delta_2}^{-1})\big|_{\Phi_2\delta_2} + (\delta_1' \circ {\delta_2'}^{-1})\Big|_{\Phi\delta_2'}$$

Then

$$\begin{aligned}
(\Phi_2\delta_2 \cup \Phi\delta_2')\rho^\alpha &= \Phi_1\delta_1 \cup \Phi\delta_1' \\
(\Delta_2\delta_2 \cup \Delta\delta_2')\rho^\alpha &= \Delta_1\delta_1 \cup \Delta\delta_1'
\end{aligned}$$

and

$$\begin{aligned}
& g_1\delta_1|g\delta_1' \\
\overset{\triangleright}{\equiv} \quad & \langle \text{Lemma 4.1.6 and CONG-PAR} \rangle \\
& (o[c!\langle v_1, \dots v_n\rangle\ p]|g')\delta_1|g\delta_1' \\
= \quad & \langle \text{renaming procedure; } \{c, v_1, \dots v_n\} \pitchfork \text{Change}(\rho) \rangle
\end{aligned}$$

$$
\begin{array}{ll}
& ((o\delta_1)[c!\langle v_1, \ldots v_n\rangle\ p\delta_1]|g'\delta_1)|g\delta_1' \\
\stackrel{\triangleright}{\equiv} & \langle \text{PAR-ASS} \rangle \\
& (o\delta_1)[c!\langle v_1, \ldots v_n\rangle\ p\delta_1]|(g'\delta_1|g\delta_1')
\end{array}
$$

Using arguments about isolated sum similar to those in Lemma B.3.16 gives us

$$
\begin{array}{ll}
& (g_2\delta_2|g\delta_2')\rho^\alpha \\
= & \langle \text{renaming procedure} \rangle \\
& g_2\delta_2\rho^\alpha|g\delta_2'\rho^\alpha \\
\stackrel{\triangleright}{\equiv}_\alpha & \langle \text{isolated sum splits} \rangle \\
& g_2\rho\delta_1|g\delta_1' \\
\stackrel{\triangleright}{\equiv}_\alpha & \langle \text{from above with CONG-PAR} \rangle \\
& (o[p]|g')\delta_1|g\delta_1' \\
= & \langle \text{renaming procedure} \rangle \\
& ((o\delta_1)[p\delta_1]|g'\delta_1)|g\delta_1' \\
\stackrel{\triangleright}{\equiv} & \langle \text{PAR-ASS} \rangle \\
& (o\delta_1)[p\delta_1]|(g'\delta_1|g\delta_1')
\end{array}
$$

as required.

Say that the last step of the proof is an instance of SCP:

$$
\frac{\Gamma, \Phi \cup \Phi' \blacktriangleright K_1 \xrightarrow[l_2]{l_1} K_2}{\Gamma, \Phi \blacktriangleright K_1\{^{\Phi'}_{\Delta} \xrightarrow[l_2]{l_1} K_2\{^{\Phi'}_{\Delta}}
$$

Say

$$
\begin{array}{rcl}
\text{Flatten}(K_1) & = & g_1\{^{\Phi_1}_{\Delta_1} \\
\text{Flatten}(K_2) & = & g_2\{^{\Phi_2}_{\Delta_2} \\
\text{Flatten}(K_1\{^{\Phi'}_{\Delta}) & = & g_1\delta_1\{^{\Phi_1\delta_1 \cup \Phi'}_{\Delta_1\delta_1 \cup \Delta} \\
\text{Flatten}(K_2\{^{\Phi'}_{\Delta}) & = & g_2\delta_2\{^{\Phi_2\delta_2 \cup \Phi'}_{\Delta_2\delta_2 \cup \Delta}
\end{array}
$$

By the induction hypothesis, there is some renaming ρ such that

$$\begin{aligned} \Phi_2\rho &= \Phi_1 \\ \Delta_2\rho &= \Delta_1 \\ g_1 &\overset{\triangleright}{\equiv} o[c!\langle v_1, \ldots v_n\rangle\ p]|g \\ g_2\rho &\overset{\triangleright}{\equiv}_\alpha o[p]|g \end{aligned}$$

Let $F = \mathrm{FN}(\mathrm{Flatten}(K_2\{^{\Phi'}_{\emptyset}))$ and let

$$\rho^\alpha = 1|_F + (\delta_1 \circ \rho \circ {\delta_2}^{-1})\big|_{\Phi_2\delta_2} + 1|_{\Phi'}$$

Then

$$\begin{aligned} (\Phi_2\delta_2 \cup \Phi')\rho^\alpha &= \Phi_1\delta_1 \cup \Phi' \\ (\Delta_2\delta_2 \cup \Delta)\rho^\alpha &= \Delta_1\delta_1 \cup \Delta \\ g_1\delta_1 &\overset{\triangleright}{\equiv} (o[c!\langle v_1, \ldots v_n\rangle\ p]|g)\delta_1 \\ &= o\delta_1[c!\langle v_1, \ldots v_n\rangle\ p\delta_1]|g\delta_1 \\ g_2\delta_2\rho^\alpha &\overset{\triangleright}{\equiv}_\alpha (o[p]|g)\delta_1 \\ &\overset{\triangleright}{\equiv}_\alpha o\delta_1[p\delta_1]|g\delta_1 \end{aligned}$$

as required. ■

Lemma B.4.8 *Say K_1 and K_2 are configurations such that*

$$\Gamma, \Phi \blacktriangleright K_1 \xrightarrow{c?\langle v_1, \ldots v_n\rangle} K_2$$

and

$$\begin{aligned} \mathrm{Flatten}(K_1) &= g_1\{^{\Phi_1}_{\Delta_1} \\ \mathrm{Flatten}(K_2) &= g_2\{^{\Phi_2}_{\Delta_2} \end{aligned}$$

then there is some renaming ρ such that $\mathrm{FN}(g_2\{^{\Phi_2}_{\Delta_2}) \pitchfork \mathrm{Change}(\rho)$ *and*

$$\begin{aligned} \Phi_2\rho &= \Phi_1 \\ \Delta_2\rho &= \Delta_1 \\ g_1 &\overset{\triangleright}{\equiv} o[c?(r_1{:}T_1, \ldots r_n{:}T_n)\ p]|g \\ g_2\rho &\overset{\triangleright}{\equiv}_\alpha o[p\{\!|{}^{v_1}/r_1 \ldots {}^{v_n}/r_n|\!\}]|g \end{aligned}$$

Lemma B.4.9 *Say K_1 and K_2 are configurations such that*

$$\Gamma, \Phi \blacktriangleright K_1 \xrightarrow[x:T]{o'.m!\langle v_1,\ldots v_n\rangle} K_2$$

and

$$\begin{array}{rcl} \text{Flatten}(K_1) & = & g_1\{^{\Phi_1}_{\Delta_1} \\ \text{Flatten}(K_2) & = & g_2\{^{\Phi_2}_{\Delta_2} \end{array}$$

then there is some renaming ρ such that $\text{FN}(g_2\{^{\Phi_2}_{\Delta_2}) \pitchfork \text{Change}(\rho)$ *and*

$$\begin{array}{rcl} \Phi_2\rho & = & \Phi_1 \\ \Delta_2\rho & = & \Delta_1 \\ g_1 & \stackrel{\triangleright}{\equiv} & o[o'.m!\langle v_1,\ldots v_n\rangle?(r_1,\ldots r_{n'})\ p]|g \\ g_2\rho & \stackrel{\triangleright}{\equiv}_\alpha & o[x?(r_1\colon T_1,\ldots r_{n'}\colon T_{n'})\ p]|g \end{array}$$

Lemma B.4.10 *Say K_1 and K_2 are configurations such that*

$$\Gamma, \Phi \blacktriangleright K_1 \xrightarrow[x:T]{o'.m?\langle v_1,\ldots v_n\rangle} K_2$$

and

$$\begin{array}{rcl} \text{Flatten}(K_1) & = & g_1\{^{\Phi_1}_{\Delta_1} \\ \text{Flatten}(K_2) & = & g_2\{^{\Phi_2}_{\Delta_2} \end{array}$$

then there is some renaming ρ such that $\text{FN}(g_2\{^{\Phi_2}_{\Delta_2}) \pitchfork \text{Change}(\rho)$ *and*

$$\begin{array}{rcl} \Phi_2\rho & = & \Phi_1 \\ \Delta_2\rho & = & \Delta_1 \\ g_2\rho & \stackrel{\triangleright}{\equiv}_\alpha & o'[p\{\!|^{v_1}/r_1,\ldots {}^{v_n}/r_n, {}^{o'}/\texttt{this}, {}^{x}/\texttt{return}|\!\}]|g_1 \end{array}$$

Our main result of this section means that any internal action taken by a configuration can be simulated by some reaction of its flattened form.

4.3.5
p120

Theorem B.4.11 *Say K_1 and K_2 are configurations such that*

$$\Gamma, \Phi \blacktriangleright K_1 \longrightarrow K_2$$

and

$$\begin{aligned} \mathrm{Flatten}(K_1\{^{\Phi}_{\emptyset}) &= g_1\{^{\Phi_1}_{\Delta_1} \\ \mathrm{Flatten}(K_2\{^{\Phi}_{\emptyset}) &= g_2\{^{\Phi_2}_{\Delta_2} \end{aligned}$$

then there is some renaming ρ such that $\mathrm{FN}(g_2\{^{\Phi_2}_{\Delta_2}) \pitchfork \mathrm{Change}(\rho)$ *and*

$$\Gamma \rhd g_1\{^{\Phi_1}_{\Delta_1} \longrightarrow g_2\rho\{^{\Phi_2\rho}_{\Delta_2\rho}$$

Proof: We use induction on the proof tree of $\Gamma, \Phi \blacktriangleright K_1 \longrightarrow K_2$. EQV-LFT, EQV-RHT, PAR and SCP can all be dealt with as in the proof of Lemma B.4.7. Say the proof tree is an instance of END:

$$\Gamma, \Phi \blacktriangleright o[\mathtt{end}] \longrightarrow \mathrm{Nil}$$

So

$$\begin{aligned} \mathrm{Flatten}(o[\mathtt{end}]\{^{\Phi}_{\emptyset}) &= o[\mathtt{end}]\{^{\Phi}_{\emptyset} \\ \mathrm{Flatten}(\mathrm{Nil}\{^{\Phi}_{\emptyset}) &= \mathrm{nil}\{^{\Phi}_{\emptyset} \end{aligned}$$

Use the identity renaming and

$$\Gamma \rhd o[\mathtt{end}]\{^{\Phi}_{\emptyset} \longrightarrow \mathrm{nil}\{^{\Phi}_{\emptyset}$$

This gives the result for this case. The case for FRK is similar.

Say that the proof tree is an instance of NEW:

$$\Gamma, \Phi \blacktriangleright o[\mathtt{new}\ c \colon \mathrm{ChT}\ p] \longrightarrow o[p]\{^{\{c:\mathrm{ChT}\}}_{\emptyset}$$

Then

$$\begin{aligned} \mathrm{Flatten}(o[\mathtt{new}\ c \colon \mathrm{ChT}\ p]\{^{\Phi}_{\emptyset}) &= o[\mathtt{new}\ c \colon \mathrm{ChT}\ p]\{^{\Phi}_{\emptyset} \\ \mathrm{Flatten}(o[p]\{^{\{c:\mathrm{ChT}\}}_{\emptyset}\{^{\Phi}_{\emptyset}) &= o[p\{\!|c'/c|\!\}]\{^{\{c':\mathrm{ChT}\}\cup\Phi}_{\emptyset} \end{aligned}$$

The second statement above is true because the shove the Flatten algorithm chooses will exchange c with some new name c'. Use the identity renaming and

$$\Gamma \rhd o[\texttt{new}\ c\colon \mathrm{ChT}\ p]\{^{\Phi}_{\emptyset} \longrightarrow o[p\{\!|c'/c|\!\}]\{^{\{c':\mathrm{ChT}\}\cup\Phi}_{\emptyset}$$

to give the result. The case for CRT is similar to this case.

Say that the proof tree is an instance of UPD:

$$\Gamma, \Phi \blacktriangleright o[a!v\ p]\{^{\emptyset}_{\{o:=f\}} \longrightarrow o[p]\{^{\emptyset}_{\{o:=f\dagger[a\mapsto v]\}}$$

Then

$$\begin{array}{rcl} \mathrm{Flatten}(o[a!v\ p]\{^{\emptyset}_{\{o:=f\}}\{^{\Phi}_{\emptyset}) & = & o[a!v\ p]\{^{\Phi}_{\{o:=f\}} \\ \mathrm{Flatten}(o[p]\{^{\emptyset}_{\{o:=f\dagger[a\mapsto v]\}}\{^{\Phi}_{\emptyset}) & = & o[p]\{^{\Phi}_{\{o:=f\dagger[a\mapsto v]\}} \end{array}$$

Use the identity renaming and

$$\Gamma \rhd o[a!v\ p]\{^{\Phi}_{\{o:=f\}} \longrightarrow o[p]\{^{\Phi}_{\{o:=f\dagger[a\mapsto v]\}}$$

to give the result. The case for ACC can be dealt with like the last two cases.

Say the proof tree ends with an instance of COM:

$$\frac{\Gamma, \Phi \blacktriangleright K_1 \xrightarrow{c!\langle v_1,\ldots v_n\rangle} K_2 \qquad \Gamma, \Phi \blacktriangleright K_1' \xrightarrow{c?\langle v_1,\ldots v_n\rangle} K_2'}{\Gamma, \Phi \blacktriangleright K_1|K_1' \longrightarrow K_2|K_2'}$$

Say

$$\begin{array}{rcl} \mathrm{Flatten}(K_1) & = & g_1\{^{\Phi_1}_{\Delta_1} \\ \mathrm{Flatten}(K_2) & = & g_2\{^{\Phi_2}_{\Delta_2} \\ \mathrm{Flatten}(K_1') & = & g_1'\{^{\Phi_1'}_{\Delta_1'} \\ \mathrm{Flatten}(K_2') & = & g_2'\{^{\Phi_2'}_{\Delta_2'} \end{array}$$

By Lemma B.4.7 and Lemma B.4.8, there exist renamings ρ and ρ' such that

$$\begin{array}{rclcrcl} \Phi_2\rho & = & \Phi_1 & \qquad & \Phi_2'\rho' & = & \Phi_1' \\ \Delta_2\rho & = & \Delta_1 & \qquad & \Delta_2'\rho' & = & \Delta_1' \end{array}$$

and

$$\begin{array}{rcl} g_1 & \stackrel{\triangleright}{\equiv} & o[c!\langle v_1, \ldots v_n \rangle\ p] | g \\ g_2\rho & \stackrel{\triangleright}{\equiv}_\alpha & o[p] | g \\ g'_1 & \stackrel{\triangleright}{\equiv} & o'[c?(r_1\colon T_1, \ldots r_n\colon T_n)\ p'] | g' \\ g'_2\rho' & \stackrel{\triangleright}{\equiv}_\alpha & o'[p'\{\!|^{v_1}/r_1, \ldots {}^{v_n}/r_n|\!\}] | g' \end{array}$$

Also

$$\begin{array}{rcl} \text{Flatten}(K_1|K'_1) & = & (g_1\delta_1|g'_1\delta'_1)\{^{\Phi_1\delta_1\cup\Phi'_1\delta'_1}_{\Delta_1\delta_1\cup\Delta'_1\delta'_1} \\ \text{Flatten}(K_2|K'_2) & = & (g_2\delta_2|g'_2\delta'_2)\{^{\Phi_2\delta_2\cup\Phi'_2\delta'_2}_{\Delta_2\delta_2\cup\Delta'_2\delta'_2} \\ \text{Flatten}((K_1|K'_1)\{^{\Phi}_{\emptyset}) & = & (g_1\delta_1|g'_1\delta'_1)\delta''_1\{^{(\Phi_1\delta_1\cup\Phi'_1\delta'_1)\delta''_1\cup\Phi}_{(\Delta_1\delta_1\cup\Delta'_1\delta'_1)\delta''_1} \\ \text{Flatten}((K_2|K'_2)\{^{\Phi}_{\emptyset}) & = & (g_2\delta_2|g'_2\delta'_2)\delta''_2\{^{(\Phi_2\delta_2\cup\Phi'_2\delta'_2)\delta''_2\cup\Phi}_{(\Delta_2\delta_2\cup\Delta'_2\delta'_2)\delta''_2} \end{array}$$

Let $F = \text{FN}(\text{Flatten}((K_2|K'_2)\{^{\Phi}_{\emptyset}))$ and let

$$\begin{array}{rcl} \rho^\alpha & = & 1|_F \\ & & +(\delta''_1 \circ \delta_1 \circ \rho \circ \delta_2 \circ \delta''_2)|_{\Phi_2\delta_2\delta''_2} \\ & & +(\delta''_1 \circ \delta'_1 \circ \rho' \circ \delta'_2 \circ \delta''_2)|_{\Phi'_2\delta'_2\delta''_2} \\ & & +1|_\Phi \end{array}$$

Then

$$\begin{array}{rcl} ((\Phi_2\delta_2 \cup \Phi'_2\delta'_2)\delta''_2 \cup \Phi)\rho^\alpha & = & (\Phi_1\delta_1 \cup \Phi'_1\delta'_1)\delta''_1 \cup \Phi \\ (\Delta_2\delta_2 \cup \Delta'_2\delta'_2)\delta''_2\rho^\alpha & = & (\Delta_1\delta_1 \cup \Delta'_1\delta'_1)\delta''_1 \end{array}$$

Now

$$\begin{array}{ll} & (g_1\delta_1|g'_1\delta'_1)\delta''_1 \\ = & \quad \langle\text{renaming procedure}\rangle \\ & g_1\delta_1\delta''_1|g'_1\delta'_1\delta''_1 \\ \stackrel{\triangleright}{\equiv} & \quad \langle\text{Lemma 4.1.6}\rangle \\ & (o[c!\langle v_1, \ldots v_n\rangle\ p]|g)\delta_1\delta''_1 \\ & \ \ |\ (o'[c?(r_1\colon T_1, \ldots r_n\colon T_n)\ p']|g')\delta'_1\delta''_1 \\ = & \quad \langle\text{renaming procedure, choosing } \delta \text{ and } \delta' \text{ appropriately}\rangle \\ & (o\delta_1\delta''_1)[c!\langle v_1, \ldots v_n\rangle\ (p\delta_1\delta''_1)]\ |\ (g\delta_1\delta''_1) \end{array}$$

$$
\begin{array}{ll}
& \quad \mid (o'\delta_1'\delta_1'')[c?((r_1\delta\delta'){:}\,T_1, \ldots (r_n\delta\delta'){:}\,T_n)\ (p'\delta\delta_1'\delta'\delta_1'')] \mid (g'\delta_1'\delta_1'') \\
\stackrel{\triangleright}{\equiv} & \langle \text{several structural equivalence rules} \rangle \\
& (o\delta_1\delta_1'')[c!\langle v_1, \ldots v_n\rangle\ (p\delta_1\delta_1'')] \\
& \quad \mid (o'\delta_1'\delta_1'')[c?((r_1\delta\delta'){:}\,T_1, \ldots (r_n\delta\delta'){:}\,T_n)\ (p'\delta\delta_1'\delta'\delta_1'')] \\
& \quad \mid ((g\delta_1\delta_1'') \mid (g'\delta_1'\delta_1''))
\end{array}
$$

and

$$
\begin{array}{ll}
& (g_2\delta_2|g_2'\delta_2')\delta_2''\rho^\alpha \\
= & \langle \text{renaming procedure} \rangle \\
& (g_2\delta_2)\delta_2''\rho^\alpha|(g_2'\delta_2')\delta_2''\rho^\alpha \\
\stackrel{\triangleright}{\equiv}_\alpha & \langle \text{choice of } \rho^\alpha \rangle \\
& (g_2\rho)\delta_1\delta_1''|(g_2'\rho')\delta_1'\delta_1'' \\
\stackrel{\triangleright}{\equiv}_\alpha & \langle \text{Lemma 4.1.11} \rangle \\
& (o[p]|g)\delta_1\delta_1'' \mid (o'[p'\{\!|{}^{v_1}/r_1, \ldots {}^{v_n}/r_n|\!\}]|g')\delta_1'\delta_1'' \\
= & \langle \text{renaming procedure} \rangle \\
& (o\delta_1\delta_1'')[p\delta_1\delta_1''] \mid (g\delta_1\delta_1'') \\
& \quad \mid (o'\delta_1'\delta_1'')[(p'\{\!|{}^{v_1}/r_1, \ldots {}^{v_n}/r_n|\!\})\delta_1\delta_1''] \mid (g'\delta_1\delta_1'') \\
\stackrel{\triangleright}{\equiv}_\alpha & \langle \text{renaming procedure, using } \delta \text{ and } \delta' \text{ from above} \rangle \\
& (o\delta_1\delta_1'')[p\delta_1\delta_1''] \mid (g\delta_1\delta_1'') \\
& \quad \mid (o'\delta_1'\delta_1'')[(p'\delta\delta_1\delta'\delta_1'')\{\!|{}^{v_1}/(r_1\delta\delta'), \ldots {}^{v_n}/(r_n\delta\delta')|\!\}] \mid (g'\delta_1\delta_1'') \\
\stackrel{\triangleright}{\equiv} & \langle \text{several structural equivalence rules} \rangle \\
& (o\delta_1\delta_1'')[p\delta_1\delta_1''] \\
& \quad \mid (o'\delta_1'\delta_1'')[(p'\delta\delta_1\delta'\delta_1'')\{\!|{}^{v_1}/(r_1\delta\delta'), \ldots {}^{v_n}/(r_n\delta\delta')|\!\}] \\
& \quad \mid ((g\delta_1\delta_1'') \mid (g'\delta_1\delta_1''))
\end{array}
$$

It is not difficult to show, using the rules of the reaction relation, that

$$\begin{array}{l}\Gamma \;\triangleright\\ \quad (o\delta_1\delta_1'')[c!\langle v_1,\dots v_n\rangle\ (p\delta_1\delta_1'')] \\ \qquad \mid (o'\delta_1'\delta_1'')[c?((r_1\delta\delta')\colon T_1,\dots (r_n\delta\delta')\colon T_n)\ (p'\delta\delta_1'\delta'\delta_1'')] \\ \qquad \mid ((g\delta_1\delta_1'') \mid (g'\delta_1'\delta_1''))\end{array} \left\{\begin{array}{l}(\Phi_1\delta_1 \cup \Phi_1'\delta_1')\delta_1'' \\ \quad \cup\ \Phi \\ (\Delta_1\delta_1 \cup \Delta_1'\delta_1')\delta_1''\end{array}\right.$$

$$\begin{array}{l}\longrightarrow\\ \quad (o\delta_1\delta_1'')[p\delta_1\delta_1''] \\ \qquad \mid (o'\delta_1'\delta_1'')[(p'\delta\delta_1\delta'\delta_1'')\{\!|{}^{v_1}/(r_1\delta\delta'),\dots {}^{v_n}/(r_n\delta\delta')|\!\}] \\ \qquad \mid ((g\delta_1\delta_1'') \mid (g'\delta_1\delta_1''))\end{array} \left\{\begin{array}{l}(\Phi_1\delta_1 \cup \Phi_1'\delta_1')\delta_1'' \\ \quad \cup\ \Phi \\ (\Delta_1\delta_1 \cup \Delta_1'\delta_1')\delta_1''\end{array}\right.$$

which, using EQV-LFT and EQV-RHT, gives our result in this case.

Say the last step of the proof tree is an instance of MTH:

$$\frac{\Gamma,\Phi \blacktriangleright K_1 \xrightarrow[x:\mathrm{ChT}]{o'.m!\langle v_1,\dots v_n\rangle} K_2 \qquad \Gamma,\Phi \blacktriangleright K_1' \xrightarrow[x:\mathrm{ChT}]{o'.m?\langle v_1,\dots v_n\rangle} K_2'}{\Gamma,\Phi \blacktriangleright K_1|K_1' \longrightarrow (K_2|K_2')\{^{\{x:\mathrm{ChT}\}}_{\emptyset}}$$

Say

$$\begin{array}{rcl}\mathrm{Flatten}(K_1) &=& g_1\{^{\Phi_1}_{\Delta_1} \\ \mathrm{Flatten}(K_2) &=& g_2\{^{\Phi_2}_{\Delta_2} \\ \mathrm{Flatten}(K_1') &=& g_1'\{^{\Phi_1'}_{\Delta_1'} \\ \mathrm{Flatten}(K_2') &=& g_2'\{^{\Phi_2'}_{\Delta_2'}\end{array}$$

By Lemma B.4.9 and Lemma B.4.10, there exist renamings ρ and ρ' such that

$$\begin{array}{rclcrcl}\Phi_2\rho &=& \Phi_1 & \qquad & \Phi_2'\rho' &=& \Phi_1' \\ \Delta_2\rho &=& \Delta_1 & & \Delta_2'\rho' &=& \Delta_1'\end{array}$$

and

$$\begin{array}{rcl}g_1 &\overset{\triangleright}{\equiv}& o[o'.m!\langle v_1,\dots v_n\rangle?(r_1,\dots r_{n'})\ p]|g \\ g_2\rho &\overset{\triangleright}{\equiv}_\alpha& o[x?(r_1\colon T_1,\dots r_{n'}\colon T_{n'})\ p]|g \\ g_2'\rho' &\overset{\triangleright}{\equiv}_\alpha& o'[p'\{\!|{}^{v_1}/s_1,\dots {}^{v_n}/s_n, {}^{o'}/\mathtt{this}, {}^{x}/\mathtt{return}|\!\}]|g_1'\end{array}$$

Also

$$\begin{aligned}
\text{Flatten}(K_1|K_1') &= (g_1\delta_1|g_1'\delta_1')\{^{\Phi_1\delta_1\cup\Phi_1'\delta_1'}_{\Delta_1\delta_1\cup\Delta_1'\delta_1'} \\
\text{Flatten}(K_2|K_2') &= (g_2\delta_2|g_2'\delta_2')\{^{\Phi_2\delta_2\cup\Phi_2'\delta_2'}_{\Delta_2\delta_2\cup\Delta_2'\delta_2'} \\
\text{Flatten}((K_2|K_2')\{^{\{x:T\}}_{\emptyset}) &= (g_2\delta_2|g_2'\delta_2')\delta_x\{^{(\Phi_2\delta_2\cup\Phi_2'\delta_2')\delta_x\cup\{x:T\}}_{(\Delta_2\delta_2\cup\Delta_2'\delta_2')\delta_x} \\
\text{Flatten}((K_1|K_1')\{^{\Phi}_{\emptyset}) &= (g_1\delta_1|g_1'\delta_1')\delta_1''\{^{(\Phi_1\delta_1\cup\Phi_1'\delta_1')\delta_1''\cup\Phi}_{(\Delta_1\delta_1\cup\Delta_1'\delta_1')\delta_1''} \\
\text{Flatten}((K_2|K_2')\{^{\{x:T\}}_{\emptyset}\{^{\Phi}_{\emptyset}) &= (g_2\delta_2|g_2'\delta_2')\delta_x\delta_2''\{^{((\Phi_2\delta_2\cup\Phi_2'\delta_2')\delta_x\cup\{x:T\})\delta_2''\cup\Phi}_{(\Delta_2\delta_2\cup\Delta_2'\delta_2')\delta_x\delta_2''}
\end{aligned}$$

Let $F = \text{FN}(\text{Flatten}((K_2|K_2')\{^{\{x:T\}}_{\emptyset}\{^{\Phi}_{\emptyset}))$, let δ_x' shove $\{x\}\cap(F\cup\text{Dom}((\Phi_1\delta_1\cup\Phi_1'\delta_1')\delta_1''\cup\Phi))$ and let

$$\begin{aligned}
\rho^\alpha = \;& 1|_F \\
& +(\delta_1''\circ\delta_1\circ\rho\circ\delta_2\circ\delta_x\circ\delta_2'')|_{\Phi_2\delta_2\delta_x\delta_2''} \\
& +(\delta_1''\circ\delta_1'\circ\rho'\circ\delta_2'\circ\delta_x\circ\delta_2'')|_{\Phi_2'\delta_2'\delta_x\delta_2''} \\
& +(\delta_x'\circ\delta_2'')|_{\{x:T\}\delta_2''} \\
& +1|_\Phi
\end{aligned}$$

Then

$$\begin{aligned}
(((\Phi_2\delta_2\cup\Phi_2'\delta_2')\delta_x\cup\{x\colon T\})\delta_2''\cup\Phi)\rho^\alpha &= (\Phi_1\delta_1\cup\Phi_1'\delta_1')\delta_1''\cup\{x\delta_x'\colon T\}\cup\Phi \\
(\Delta_2\delta_2\cup\Delta_2'\delta_2')\delta_x\delta_2''\rho^\alpha &= (\Delta_1\delta_1\cup\Delta_1'\delta_1')\delta_1''
\end{aligned}$$

Now

$$\begin{aligned}
& (g_1\delta_1|g_1'\delta_1')\delta_1'' \\
= \quad & \langle\text{renaming procedure}\rangle \\
& g_1\delta_1\delta_1'' \mid g_1'\delta_1'\delta_1'' \\
\overset{\triangleright}{\equiv} \quad & \langle\text{Lemma 4.1.11}\rangle \\
& (o[o'.m!\langle v_1,\dots v_n\rangle?(r_1,\dots r_{n'})\ p]|g)\delta_1\delta_1'' \mid g_1'\delta_1'\delta_1'' \\
\overset{\triangleright}{\equiv} \quad & \langle\text{renaming procedure and structural equivalence laws}\rangle \\
& (o\delta_1\delta_1'')[o'.m!\langle v_1,\dots v_n\rangle?((r_1\delta\delta'),\dots(r_{n'}\delta\delta'))\ (p\delta\delta_1\delta'\delta_1'')] \\
& \mid ((g\delta_1\delta_1'') \mid (g_1'\delta_1'\delta_1''))
\end{aligned}$$

To avoid the complexity of showing how ρ^α affects the two parallel agents, we skip several steps in the following calculation.

$$
\begin{array}{ll}
& (g_2\delta_2|g_2'\delta_2')\delta_x\delta_2''\rho^\alpha \\
= & \langle\text{renaming procedure}\rangle \\
& (g_2\delta_2)\delta_x\delta_2''\rho^\alpha|(g_2'\delta_2')\delta_x\delta_2''\rho^\alpha \\
\stackrel{\triangleright}{\equiv}_\alpha & \langle\text{lemmas 4.1.11 and 4.1.8 and equivalence laws}\rangle \\
& (o\delta_1\delta_1'')[(x\delta_x')?((r_1\delta\delta'){:}\,T_1,\ldots(r_{n'}\delta\delta'){:}\,T_{n'})\ (p\delta\delta_1\delta'\delta_1'')] \\
& \quad |\ o'[p'\{\!|^{v_1}/s_1,\ldots{}^{v_n}/s_n,{}^{o'}/\mathtt{this},{}^{(x\delta_x')}/\mathtt{return}|\!\}] \\
& \quad |\ ((g\delta_1\delta_1'')\ |\ (g_1'\delta_1'\delta_1''))
\end{array}
$$

It is not difficult to show, using the rules of the reaction system, that

$$
\begin{array}{ll}
\Gamma\ \triangleright & \\
\begin{array}{l}
(o\delta_1\delta_1'')[o'.m!\langle v_1,\ldots v_n\rangle?((r_1\delta\delta'),\ldots(r_{n'}\delta\delta')) \\
\hfill (p\delta\delta_1\delta'\delta_1'')] \\
\quad |\ ((g\delta_1\delta_1'')\ |\ (g_1'\delta_1'\delta_1''))
\end{array}
& \begin{cases} (\Phi_1\delta_1\cup\Phi_1'\delta_1')\delta_1'' \\ \ \cup\ \Phi \\ (\Delta_1\delta_1\cup\Delta_1'\delta_1')\delta_1'' \end{cases} \\
\longrightarrow & \\
\begin{array}{l}
(o\delta_1\delta_1'')[(x\delta_x')?((r_1\delta\delta'){:}\,T_1,\ldots(r_{n'}\delta\delta'){:}\,T_{n'})\ (p\delta\delta_1\delta'\delta_1'')] \\
\quad |\ o'[p'\{\!|^{v_1}/s_1,\ldots{}^{v_n}/s_n,{}^{o'}/\mathtt{this},{}^{(x\delta_x')}/\mathtt{return}|\!\}] \\
\quad |\ ((g\delta_1\delta_1'')\ |\ (g_1'\delta_1'\delta_1''))
\end{array}
& \begin{cases} (\Phi_1\delta_1\cup\Phi_1'\delta_1')\delta_1'' \\ \ \cup\{x\delta_x'{:}\,T\}\cup\Phi \\ (\Delta_1\delta_1\cup\Delta_1'\delta_1')\delta_1'' \end{cases}
\end{array}
$$

which, using EQV-LFT and EQV-RHT, gives the result in this case. ■

Appendix C

Technical Results for the Type System

This appendix provides the proofs of results relating to Oompa's type theory. It is structured into four sections. Section C.1 deals with Oompa's type trees, Section C.2 considers subtyping, Section C.3 establishes the soundness of the type safety system with respect to the agent-based dynamic system and Section C.4 does the same for the configuration-based system.

C.1 Type Tress

In this section, we give the proofs of Lemma 5.2.2 and Theorem 5.2.3.

5.2.2 p136

Lemma C.1.1 $\mathrm{E}_{\Gamma}^{V}(T)$ *is finite and can be calculated with finite applications of the look-up function* $\mathrm{e}_{\Gamma}(\cdot)$ *for any finite* Γ*,* T *and* V*.*

Proof: Consider the evaluation as a tree, where an application of the expansion function labels the nodes. Consider such a node $\mathrm{E}_{\Gamma}^{V'}(T')$. Let SubTerms[T] give the set of proper subterms of a type T [1]. We use the tuple

[1]Not the same as the function Sub[·] defined on page 323

$(\#V', \#\mathrm{SubTerms}[\mathrm{T}'])$ as a metric, with the following ordering:

$$(t_1, t_2) \leq (t'_1, t'_2) \text{ if } t_1 \geq t'_1 \text{ or } t_1 = t'_1 \text{ and } t_2 \leq t'_2$$

As we move down through the tree, either the first argument grows or it stays the same and the second shrinks. The first argument is bounded above by $\#V$ plus the number of occurrences of type variables in Γ, which is certainly finite. The second is bounded below by 0. ■

In order to prove Theorem C.1.3, we define a recursive function which, for a given n, is like $\mathrm{Tree}_\Gamma(\cdot)$ except that it only performs n recursive unfoldings and definition look-ups. We augment the language L with the symbol "$\perp^0$", and label nodes with this symbol at the point where an nth unfolding or look-up would previously occur. The different cases are as follows:

$$\begin{array}{lcl}
\mathrm{Tree}^0_\Gamma(\mathtt{rec}\ t.T)(\Lambda) & = & \perp^0 \\
\mathrm{Tree}^{n+1}_\Gamma(\mathtt{rec}\ t.T)(\pi) & = & \mathrm{Tree}^n_\Gamma(T\{\!|^{\mathtt{rec}\ t.T}/t|\!\}) \\
\mathrm{Tree}^0_\Gamma(t)(\Lambda) & = & \perp^0 \\
\mathrm{Tree}^{n+1}_\Gamma(t)(\pi) & = & \mathrm{Tree}^n_\Gamma(\mathrm{e}_\Gamma(t))(\pi)
\end{array}$$

The functions $\mathrm{Tree}^n_\Gamma(\cdot)$ can be viewed as *finite approximations* to the function $\mathrm{Tree}_\Gamma(\cdot)$[2].

Lemma C.1.2 $\mathrm{Tree}_\Gamma(T)(\pi) = \mathrm{Tree}^n_\Gamma(T)(\pi)$ *for all* $n > |\pi|$

Proof: We use induction on the length of π. The base cases are when $\pi = \Lambda$. Consider the structure of T. The base cases for the guarded types and the case for signature types are all the same, since $\mathrm{Tree}_\Gamma(T)$ and $\mathrm{Tree}^n_\Gamma(T)$ are the same on these types.

[2]Using finite approximations to prove a result such as Lemma C.1.2 might be considered old fashioned. An alternative approach might use a co-inductive technique instead.

Say $T = t$, some variable. Then, since $n > 0$ we have:

$$\begin{aligned}\mathrm{Tree}_\Gamma(t)(\Lambda) &= \mathrm{Tree}_\Gamma(\mathrm{e}_\Gamma(t))(\Lambda)\\ \mathrm{Tree}^n_\Gamma(t)(\Lambda) &= \mathrm{Tree}^{n-1}_\Gamma(\mathrm{e}_\Gamma(t))(\Lambda)\end{aligned}$$

In this case both will equal to Intf^m, as that will be the outermost type constructor of $\mathrm{e}_\Gamma(t)$.

Say $T = \texttt{rec}\ t.T'$. Then T' is guarded and since $n > 0$ we have:

$$\begin{aligned}\mathrm{Tree}_\Gamma(\texttt{rec}\ t.T')(\Lambda) &= \mathrm{Tree}_\Gamma(T'\{\!|^{\texttt{rec}\ t.T'}/t|\!\})(\Lambda)\\ \mathrm{Tree}^n_\Gamma(\texttt{rec}\ t.T')(\Lambda) &= \mathrm{Tree}^{n-1}_\Gamma(T'\{\!|^{\texttt{rec}\ t.T'}/t|\!\})(\Lambda)\end{aligned}$$

These will be the same, as the two functions are the same on guarded types, giving us the result in this case.

The inductive cases occur when $\pi = i\pi'$. Again we consider the structure of T.

Say $T = t$, some variable. Then if $\mathrm{e}_\Gamma(t)$ has signature types $\mathrm{SgT}_1, \ldots \mathrm{SgT}_k$ where $k \geq i$ we have

$$\mathrm{Tree}_\Gamma(t)(\pi) = \mathrm{Tree}_\Gamma(\mathrm{e}_\Gamma(t))(i\pi') = \mathrm{Tree}_\Gamma(\mathrm{SgT}_i)(\pi')$$

$$\mathrm{Tree}^n_\Gamma(t)(\pi) = \mathrm{Tree}^{n-1}_\Gamma(\mathrm{e}_\Gamma(t))(i\pi') = \mathrm{Tree}^{n-1}_\Gamma(\mathrm{SgT}_i)(\pi')$$

In this case they are equal by the induction hypothesis. In the case where $\mathrm{e}_\Gamma(t)$ has less than i signatures both are undefined.

Say $T = \texttt{rec}\ t.T'$ and T' has immediate subterms $T_1, \ldots T_k$ where $k \geq i$. Then T' is guarded and

$$\begin{aligned}\mathrm{Tree}_\Gamma(\texttt{rec}\ t.T')(\pi) &= \mathrm{Tree}_\Gamma(T'\{\!|^{\texttt{rec}\ t.T'}/t|\!\})(i\pi')\\ &= \mathrm{Tree}_\Gamma(T_i\{\!|^{\texttt{rec}\ t.T'}/t|\!\})(\pi')\end{aligned}$$

$$\begin{aligned}\mathrm{Tree}^n_\Gamma(\texttt{rec}\ t.T')(\pi) &= \mathrm{Tree}^{n-1}_\Gamma(T'\{\!|^{\texttt{rec}\ t.T'}/t|\!\})(i\pi')\\ &= \mathrm{Tree}^{n-1}_\Gamma(T_i\{\!|^{\texttt{rec}\ t.T'}/t|\!\})(\pi')\end{aligned}$$

and these are equal by the induction hypothesis. In the case where T' has less than i immediate subterms both are undefined.

The guarded cases and the case for signature types are all the same. Say T is some guarded type with immediate subterms $T_1, \dots T_k$ where $k \geq i$. Then

$$\mathrm{Tree}_\Gamma(T)(i\pi') = \mathrm{Tree}_\Gamma(T_i)(\pi')$$

$$\mathrm{Tree}^n_\Gamma(T)(i\pi') = \mathrm{Tree}^{n-1}_\Gamma(T_i)(\pi')$$

These are the same by the induction hypothesis. In the case where T has less than i immediate subterms both are undefined. ■

5.2.3 p136

Theorem C.1.3 $\Gamma \models S = \mathrm{E}^\emptyset_\Gamma(S)$

Proof: We have to prove

$$\mathrm{Tree}_\Gamma(S) = \mathrm{Tree}_\Gamma(\mathrm{E}^\emptyset_\Gamma(S))$$

We prove this by showing that for any n and set of variables V,

$$\mathrm{Tree}^n_\Gamma(S) = \mathrm{Tree}^n_\Gamma(\mathrm{E}^V_\Gamma(S))$$

by induction on n. The theorem statement follows from Lemma C.1.2, since for any argument sequence π we can pick $n = |\pi| + 1$ and $V = \emptyset$.

The base case is where $n = 0$. We use another induction on the construction of S. If $S = \texttt{Long}$ or $S = \texttt{Char}$, then $\mathrm{E}^V_\Gamma(S) = S$. The trees are obviously the same in these cases.

If $S = t$, there are two subcases. If $t \in V$ then $\mathrm{E}^V_\Gamma(S) = S$ and again the trees are the same. If $t \notin V$ then $\mathrm{E}^V_\Gamma(t) = \texttt{rec}\ t.\mathrm{E}^{V\cup\{t\}}_\Gamma(\mathrm{e}_\Gamma(t))$. But then, given π, we have that $\mathrm{Tree}^0_\Gamma(t)(\pi)$ and $\mathrm{Tree}^0_\Gamma(\texttt{rec}\ t.\mathrm{E}^{V\cup\{t\}}_\Gamma(\mathrm{e}_\Gamma(t)))(\pi)$ are either both $\bot^0$ if $\pi = \Lambda$ or both undefined. Thus the trees are the same.

If $S = \texttt{rec}\ t.T$, then $\mathrm{E}_\Gamma^V(S) = \texttt{rec}\ t.\mathrm{E}_\Gamma^{V\cup\{t\}}(T)$ and, given π, we have that $\mathrm{Tree}_\Gamma^0(\texttt{rec}\ t.T)(\pi)$ and $\mathrm{Tree}_\Gamma^0(\texttt{rec}\ t.\mathrm{E}_\Gamma^{V\cup\{t\}}(T))(\pi)$ are either both $\perp^0$ if $\pi = \Lambda$ or both undefined. So the trees are the same.

If $S = \texttt{Chan}\langle T_1, \ldots T_n\rangle$, then $\mathrm{E}_\Gamma^V(S) = \texttt{Chan}\langle \mathrm{E}_\Gamma^V(T_1) \ldots \mathrm{E}_\Gamma^V(T_n)\rangle$. The trees are the same since:

$$\begin{array}{rcl}
\mathrm{Tree}_\Gamma^0(S)(\Lambda) & = & \mathrm{Chan}^n \\
\mathrm{Tree}_\Gamma^0(\mathrm{E}_\Gamma^V(S))(\Lambda) & = & \mathrm{Chan}^n \\
\mathrm{Tree}_\Gamma^0(S)(i\pi) & = & \mathrm{Tree}_\Gamma^0(T_i)(\pi) \\
\mathrm{Tree}_\Gamma^0(\mathrm{E}_\Gamma^V(S))(i\pi) & = & \mathrm{Tree}_\Gamma^0(\mathrm{E}_\Gamma^V(T_i))(\pi)
\end{array}$$

The last two are equal by the local induction hypothesis. The other inductive cases for S are like this channel case.

We now consider the cases when $n > 0$. Again, we use another induction on the construction of S. If $S = \texttt{Long}$ or $S = \texttt{Char}$ then $S = \mathrm{E}_\Gamma^V(S)$, so the trees must be the same. Note that we won't be able to use the local induction hypothesis in the cases for $S = t$ and $S = \texttt{rec}\ t.T$. The main induction hypothesis will be sufficient. If $S = t$ and $t \in V$ then since $\mathrm{E}_\Gamma^V(t) = t$, the trees are the same.

If $S = t$ where $t \notin V$, then

$$\mathrm{E}_\Gamma^V(S) = \mathrm{E}_\Gamma^V(t) = \texttt{rec}\ t.\mathrm{E}_\Gamma^{V\cup\{t\}}(\mathrm{e}_\Gamma(t))$$

and

$$\mathrm{Tree}_\Gamma^n(\texttt{rec}\ t.\mathrm{E}_\Gamma^{V\cup\{t\}}(\mathrm{e}_\Gamma(t))) = \mathrm{Tree}_\Gamma^{n-1}(\mathrm{E}_\Gamma^{V\cup\{t\}}(\mathrm{e}_\Gamma(t))\{\!|{}^{\texttt{rec}\ t.\mathrm{E}_\Gamma^{V\cup\{t\}}(\mathrm{e}_\Gamma(t))}/_{t}|\!\})$$

Also

$$\begin{array}{rcl}
\mathrm{Tree}_\Gamma^n(t) & = & \mathrm{Tree}_\Gamma^{n-1}(\mathrm{e}_\Gamma(t)) \\
& = & \mathrm{Tree}_\Gamma^{n-1}(\mathrm{E}_\Gamma^{V\cup\{t\}}(\mathrm{e}_\Gamma(t)))
\end{array}$$

by the induction hypothesis.

Therefore, it will be sufficient to show that the following two trees are equal:

$$\mathrm{Tree}_{\Gamma}^{n-1}(\mathrm{E}_{\Gamma}^{V\cup\{t\}}(\mathrm{e}_{\Gamma}(t))\{\!|^{\texttt{rec}\ t.\mathrm{E}_{\Gamma}^{V\cup\{t\}}(\mathrm{e}_{\Gamma}(t))}/_{t}|\!\})$$
$$\mathrm{Tree}_{\Gamma}^{n-1}(\mathrm{E}_{\Gamma}^{V\cup\{t\}}(\mathrm{e}_{\Gamma}(t)))$$

It is easy to see that the only place they could differ is at the subtrees where $\mathrm{E}_{\Gamma}^{V\cup\{t\}}(\mathrm{e}_{\Gamma}(t))$ has a free occurrence of t. For one such occurrence the two subtrees will have the form:

$$\mathrm{Tree}_{\Gamma}^{n-1-m}(\texttt{rec}\ t.\mathrm{E}_{\Gamma}^{V\cup\{t\}}(\mathrm{e}_{\Gamma}(t)))$$
$$\mathrm{Tree}_{\Gamma}^{n-1-m}(t)$$

for some $m \geq 0$ (m will be the number of `rec` operators the interpretation function will have passed through). We rely on the convention that bound variables are renamed to guarantee that none of the free variables of $\texttt{rec}\ t.\mathrm{E}_{\Gamma}^{V\cup\{t\}}(\mathrm{e}_{\Gamma}(t))$ will be captured when it is substituted into $\mathrm{E}_{\Gamma}^{V\cup\{t\}}(\mathrm{e}_{\Gamma}(t))$. Thus, no substitution generated when the Tree function encounters the `rec` operator will apply to it. This justifies the form of the first subtree above.

By the induction hypothesis applied to the second subtree:

$$\begin{aligned}\mathrm{Tree}_{\Gamma}^{n-1-m}(t) &= \mathrm{Tree}_{\Gamma}^{n-1-m}(\mathrm{E}_{\Gamma}^{V\cup\{t\}}(t)) \\ &= \mathrm{Tree}_{\Gamma}^{n-1-m}(\texttt{rec}\ t.\mathrm{E}_{\Gamma}^{V\cup\{t\}}(\mathrm{e}_{\Gamma}(t)))\end{aligned}$$

Thus, these subtrees are the same. Thus the main trees must be equal.

Say $S = \texttt{rec}\ t.T$ and use the variable convention to make sure that t is chosen different to any other name in the system. Then

$$\begin{aligned}&\mathrm{Tree}_{\Gamma}^{n}(\mathrm{E}_{\Gamma}^{V}(S)) \\ = \quad &\langle \text{by the definition of } \mathrm{E}_{\Gamma}^{V}(\cdot)\rangle \\ &\mathrm{Tree}_{\Gamma}^{n}(\texttt{rec}\ t.\mathrm{E}_{\Gamma}^{V\cup\{t\}}(T)) \\ = \quad &\langle \text{by the definition of } \mathrm{Tree}_{\Gamma}^{n}(\cdot)\rangle \\ &\mathrm{Tree}_{\Gamma}^{n-1}(\mathrm{E}_{\Gamma}^{V\cup\{t\}}(T)\{\!|^{\texttt{rec}\ t.\mathrm{E}_{\Gamma}^{V\cup\{t\}}(T)}/_{t}|\!\})\end{aligned}$$

Also,

$$
\begin{array}{ll}
& \mathrm{Tree}^{n}_{\Gamma}(\texttt{rec}\ t.T) \\
= & \quad \langle \text{by the definition of } \mathrm{Tree}^{n}_{\Gamma}(\cdot) \rangle \\
& \mathrm{Tree}^{n-1}_{\Gamma}(T\{\!|^{\texttt{rec}\ t.T}/t|\!\}) \\
= & \quad \langle \text{main induction hypothesis} \rangle \\
& \mathrm{Tree}^{n-1}_{\Gamma}(\mathrm{E}^{V\cup\{t\}}_{\Gamma}(T\{\!|^{\texttt{rec}\ t.T}/t|\!\}))
\end{array}
$$

Therefore, it will be sufficient to show that the following two trees are equal:

$$
\begin{array}{l}
\mathrm{Tree}^{n-1}_{\Gamma}(\mathrm{E}^{V\cup\{t\}}_{\Gamma}(T)\{\!|^{\texttt{rec}\ t.\mathrm{E}^{V\cup\{t\}}_{\Gamma}(T)}/t|\!\}) \\
\mathrm{Tree}^{n-1}_{\Gamma}(\mathrm{E}^{V\cup\{t\}}_{\Gamma}(T\{\!|^{\texttt{rec}\ t.T}/t|\!\}))
\end{array}
$$

These trees will have the same "upper structure" and they could only differ at the subtrees where T has free variables. There are two cases.

Consider an occurrence of t in T. The subtrees at this point will have the form

$$
\begin{array}{l}
\mathrm{Tree}^{n-1-m}_{\Gamma}(\texttt{rec}\ t.\mathrm{E}^{V\cup\{t\}}_{\Gamma}(T)) \\
\mathrm{Tree}^{n-1-m}_{\Gamma}(\mathrm{E}^{V\cup U\cup\{t\}}_{\Gamma}(\texttt{rec}\ t.T))
\end{array}
$$

for some $m \geq 0$ and set of variables U. Again, we rely on the convention that bound variables are renamed to guarantee that none of the free variables of $\texttt{rec}\ t.T$ are captured when it is substituted into T. Thus, no substitution generated when the Tree function encounters the `rec` operator will apply to it.

The first subtree is equal to

$$\mathrm{Tree}^{n-1-m}_{\Gamma}(\mathrm{E}^{V}_{\Gamma}(\texttt{rec}\ t.T))$$

and by the induction hypothesis twice this is equal to

$$\mathrm{Tree}^{n-1-m}_{\Gamma}(\texttt{rec}\ t.T) = \mathrm{Tree}^{n-1-m}_{\Gamma}(\mathrm{E}^{V\cup U\cup\{t\}}_{\Gamma}(\texttt{rec}\ t.T))$$

Thus in this case, the subtrees are the same.

Next, consider an occurrence of s in T, where $s \neq t$. The subtrees at this point will be

$$\mathrm{Tree}_{\Gamma}^{n-1-m}(\mathrm{E}_{\Gamma}^{V \cup U \cup \{t\}}(s)\rho_1 \ldots \rho_k)$$
$$\mathrm{Tree}_{\Gamma}^{n-1-m}(\mathrm{E}_{\Gamma}^{V \cup U \cup \{t\}}(s)\{\![\texttt{rec}\ t.\mathrm{E}_{\Gamma}^{V \cup \{t\}}(T)/t]\!\}\rho_1 \ldots \rho_k)$$

where $\rho_1 \ldots \rho_k$ are substitutions generated when the Tree function encounters the `rec` operator. As t was chosen different from any other name, the extra substitution in the second subtree can be dropped — it won't apply to anything. Thus, these subtrees are the same, and hence the main trees are the same.

Say $S = \texttt{Intf}\{\mathrm{SgT}_1, \ldots \mathrm{SgT}_m\}$. Then

$$\mathrm{E}_{\Gamma}^{V}(S) = \texttt{Intf}\{\mathrm{E}_{\Gamma}^{V}(\mathrm{SgT}_1), \ldots \mathrm{E}_{\Gamma}^{V}(\mathrm{SgT}_m)\}$$

The trees are the same, since:

$$\begin{aligned}
\mathrm{Tree}_{\Gamma}^{n}(S)(\Lambda) &= \mathrm{Intf}^{m} \\
\mathrm{Tree}_{\Gamma}^{n}(\mathrm{E}_{\Gamma}^{V}(S))(\Lambda) &= \mathrm{Intf}^{m} \\
\mathrm{Tree}_{\Gamma}^{n}(S)(i\pi) &= \mathrm{Tree}_{\Gamma}^{n}(\mathrm{SgT}_i)(\pi) \\
\mathrm{Tree}_{\Gamma}^{n}(\mathrm{E}_{\Gamma}^{V}(S))(i\pi) &= \mathrm{Tree}_{\Gamma}^{n}(\mathrm{E}_{\Gamma}^{V}(\mathrm{SgT}_i))(\pi)
\end{aligned}$$

The last two are equal by the local induction hypothesis. The other inductive cases for S are all like this one, so we're done. ■

C.2 The Subtyping System

This section establishes the termination, completeness and soundness of the subtyping system, results 5.3.1, 5.3.2 and 5.3.3 respectively.

C.2.1 Termination

We need the following technical construction to facilitate some of our proofs. We define a function Sub[·], which gives the set of subterms of a closed type including recursive unfoldings:

$$
\begin{array}{rcl}
\mathrm{Sub}[\mathtt{Chan}\langle T_1, \ldots T_n\rangle] & = & \{\mathtt{Chan}\langle T_1, \ldots T_n\rangle\} \\
 & & \cup\ \mathrm{Sub}[T_1]\ \cup \ldots \mathrm{Sub}[T_n] \\
\mathrm{Sub}[\mathtt{InCh}\langle T_1, \ldots T_n\rangle] & = & \{\mathtt{InCh}\langle T_1, \ldots T_n\rangle\} \\
 & & \cup\ \mathrm{Sub}[T_1]\ \cup \ldots \mathrm{Sub}[T_n] \\
\mathrm{Sub}[\mathtt{OuCh}\langle T_1, \ldots T_n\rangle] & = & \{\mathtt{OuCh}\langle T_1, \ldots T_n\rangle\} \\
 & & \cup\ \mathrm{Sub}[T_1] \cup\ \ldots \mathrm{Sub}[T_n] \\
\mathrm{Sub}[\mathtt{Intf}\{\mathrm{SgT}_1, \ldots \mathrm{SgT}_n\}] & = & \{\mathtt{Intf}\{\mathrm{SgT}_1, \ldots \mathrm{SgT}_n\}\} \\
 & & \cup\ \mathrm{Sub}[\mathrm{SgT}_1]\ \cup \ldots \mathrm{Sub}[\mathrm{SgT}_n] \\
\mathrm{Sub}[m?(S_1, \ldots S_n)!\langle T_1, \ldots T_{n'}\rangle] & = & \{m?(S_1, \ldots S_n)!\langle T_1, \ldots T_{n'}\rangle\} \\
 & & \cup\ \mathrm{Sub}[S_1]\ \cup \ldots \mathrm{Sub}[S_n] \\
 & & \cup\ \mathrm{Sub}[T_1]\ \cup \ldots \mathrm{Sub}[T_{n'}] \\
\mathrm{Sub}[\mathtt{rec}\ t.T] & = & \{\mathtt{rec}\ t.T\} \\
 & & \cup\ \{S\{\!\!\{^{\mathtt{rec}\ t.T}/t\}\!\!\} \mid S \in \mathrm{Sub}[T]\}
\end{array}
$$

We also use $\mathrm{Sub}[S, T]$ to represent $\mathrm{Sub}[S] \cup \mathrm{Sub}[T]$.

Lemma C.2.1 *For a finite closed type T,* $\mathrm{Sub}[T]$ *is finite*

Proof: $\mathrm{Sub}[T]$ can have no more elements than the distinct subterms of T. ■

Theorem C.2.2 ***(Termination)*** *The subtyping algorithm is terminating.* 5.3.1 p140

Proof: Consider the algorithm applied to $\vDash_a S \leq T$ where S and T are closed. Consider a node in the proof tree $\Sigma \vDash_a S' \leq T'$. We claim that this node satisfies the following three properties:

1. S' and T' are in $\mathrm{Sub}[S, T]$.

2. For every assumption $U_1 \leq U_2 \in \Sigma$, both U_1 and U_2 are in $\mathrm{Sub}[S, T]$.

3. No assumption occurs in Σ more than once.

These properties clearly hold for the root node. We show that if the current goal node satisfies them, then so do the premises.

In the cases of the SUB-RFL and SUB-ASS rules, there are no premises. The cases for the channel subtyping rules and the SUB-SIG and SUB-INT rules are all similar, so we describe only one. Consider the SUB-CHIN rule. $S = \mathtt{Chan}\langle S_1, \ldots S_n\rangle \in \mathrm{Sub}[S, T]$ so by the definition of $\mathrm{Sub}[\cdot]$, $S_1, \ldots S_n \in \mathrm{Sub}[S, T]$ and $T = \mathtt{InCh}\langle T_1, \ldots T_n\rangle \in \mathrm{Sub}[S, T]$, so $T_1, \ldots T_n \in \mathrm{Sub}[S, T]$. This means that the first property holds for all the premises. This rule adds no new assumptions, so the second and third property hold.

Consider the SUB-RECRHT rule. S' was in $\mathrm{Sub}[S, T]$ as it is one of the types in the antecedent. $T'\{\!|{}^{\mathtt{rec}\ t.T'}/t|\!\}$ is in $\mathrm{Sub}[S, T]$, since $\mathtt{rec}\ t.T'$ is. Thus the first property holds. The second property comes from the fact that the added assumption is exactly the antecedent, so both of its elements are in $\mathrm{Sub}[S, T]$. The third property comes from the fact that if we added an assumption which was already there, it would have contradicted the algorithm order, which stipulates that the SUB-ASS rule be attempted before the SUB-RECRHT rule. The SUB-RECLFT rule can be dealt with similarly.

We now associate a measure with each node. Let

$$M(\Sigma \vdash_a S \leq T) = (n, m)$$

where n is the number of assumptions in Σ and m is the maximum nesting of brackets in either S or T. We say that $(n, m) > (n', m')$ if $n < n'$ or else $n = n'$ and $m > m'$.

It is easy to see that at any application of the rules, the measure of any of the premises is less than that of the antecedent. The SUB-RFL and SUB-ASS rules do not generate any new premises, the SUB-RECLFT and SUB-RECRHT rules add an assumption and the other rules all reduce the maximum nesting of brackets in their types. The measure of a goal cannot decrease for ever, as the number of assumptions is bounded by $(\#\mathrm{Sub}[S,T])^2$ and the second component must stay greater than zero. Thus the algorithm must terminate. ■

C.2.2 Completeness

In this section, we need another technical construction.

Call a subtype algorithm statement, $\Sigma \models_a S \leq T$, *sound* if

- $\mathrm{Tree}(S) \leq \mathrm{Tree}(T)$
- $\mathrm{Tree}(U_1) \leq \mathrm{Tree}(U_2)$ for each $U_1 \leq U_2 \in \Sigma$.

Lemma C.2.3 *If a statement, $\Sigma \models_a S \leq T$, is sound, then it matches the antecedent of one of the rules of the subtyping algorithm and the premises of the rule are all sound.*

Proof: Let $\Sigma \models_a S \leq T$ be a sound statement. If $S = T$ then the SUB-RFL rule applies and there are no premises. If $S \leq T \in \Sigma$ then the SUB-ASS rule applies and there are no premises.

Suppose $S \leq T \notin \Sigma$. Say neither S nor T are of recursive form. Since the statement is sound, $\mathrm{Tree}(S) \leq \mathrm{Tree}(T)$, so there is some tree simulation $\mathcal{R}$ such that $(\mathrm{Tree}(S), \mathrm{Tree}(T)) \in \mathcal{R}$. For this to be the case, $\mathcal{R}$ must satisfy the conditions of a tree simulation and we consider those conditions as they apply to $\mathrm{Tree}(T)(\Lambda)$.

If $T = \texttt{Long}$, then $\text{Tree}(T)(\Lambda) = \texttt{Long}^0$ and hence $\text{Tree}(S)(\Lambda) = \texttt{Long}^0$, so $S = \texttt{Long}$. Therefore, $S = T$ and we have dealt with this case above.

We pick $T = \texttt{InCh}\langle T_1, \ldots T_n\rangle$ as an example of the other non-recursive cases. They can all be tackled in a similar way. If $T = \texttt{InCh}\langle T_1, \ldots T_n\rangle$, then $\text{Tree}(T)(\Lambda) = \text{InCh}^n$. Hence, by the definition of Oompa tree simulation, either $\text{Tree}(S)(\Lambda) = \text{InCh}^n$ or $\text{Tree}(S)(\Lambda) = \text{Chan}^n$. It follows that S is either $\texttt{InCh}\langle S_1, \ldots S_n\rangle$ or $\texttt{Chan}\langle S_1, \ldots S_n\rangle$, and that $\text{Tree}(S_i) = \text{Tree}(S)(i) \leq \text{Tree}(T)(i) = \text{Tree}(T_i)$. Either the SUB-ININ or SUB-CHIN rules will match $\Gamma \models_a S \leq T$ and the premises will all have the form $\Sigma \models_a S_i \leq T_i$. Σ is the same as for the antecedent, so the premises are sound as required.

Suppose S is of recursive form, say $S = \texttt{rec}\ s.S'$. Then SUB-RECLFT applies. This rule has one premise of the form $\Sigma \cup \{S \leq T\} \models_a S'\{\!|^{\texttt{rec}\ s.S'}/s|\!\} \leq T$. The new assumption, $S \leq T$, has $\text{Tree}(S) \leq \text{Tree}(T)$ since the original judgement was sound. Also, $\text{Tree}(\texttt{rec}\ s.S') = \text{Tree}(S'\{\!|^{\texttt{rec}\ s.S'}/s|\!\})$ and the right hand side of the judgements, T, has not changed, so the trees of the premise types are still the same. Thus the premise is sound. The case when T is of recursive form is similar. ■

Lemma C.2.4 *Say S and T are closed types and* $\text{Tree}(S) \leq \text{Tree}(T)$*. Then* $\models_a S \leq T$*.*

Proof: If $\text{Tree}(T) \leq \text{Tree}(S)$ then $\models_a S \leq T$ is sound. The algorithm cannot return false on a sound statement since, by Lemma C.2.3, any premises of any sound statement are also sound and therefore matched by a rule. Since the algorithm is terminating by Theorem C.2.2, it must return true. ■

5.3.2
p140

Theorem C.2.5 ***(Completeness)*** *If* $\Gamma \models S \leq T$ *then* $\Gamma \vdash S \leq T$*.*

Proof: $\Gamma \models S \leq T$ means $\text{Tree}_\Gamma(S) \leq \text{Tree}_\Gamma(T)$. But by Theorem C.1.3,

$$\text{Tree}(\text{E}^{\emptyset}_{\Gamma}(S)) = \text{Tree}_\Gamma(S) \leq \text{Tree}_\Gamma(T) = \text{Tree}(\text{E}^{\emptyset}_{\Gamma}(T))$$

Both of $\mathrm{E}^{\emptyset}_{\Gamma}(S)$ and $\mathrm{E}^{\emptyset}_{\Gamma}(T)$ are closed, so by the Lemma C.2.4, the algorithm will affirm $\models_a \mathrm{E}^{\emptyset}_{\Gamma}(S) \leq \mathrm{E}^{\emptyset}_{\Gamma}(T)$ which is what $\Gamma \vdash S \leq T$ means. ■

C.2.3 Soundness

The following lemma describes a property usually called *weakening*.

Lemma C.2.6 $\Sigma \models_a S \leq T$ *implies* $\Sigma \cup \Sigma' \models_a S \leq T$.

Proof: This is obvious after an inspection of the rules. ■

Lemma C.2.7

1. *Suppose* $\models_a \texttt{rec}\ s.S \leq T$ *and* $(\texttt{rec}\ s.S \leq T) \notin \Sigma$. *Then*

$$\Sigma \cup \{\texttt{rec}\ s.S \leq T\} \models_a U_1 \leq U_2 \quad \textit{implies} \quad \Sigma \models_a U_1 \leq U_2$$

2. *Suppose* $\models_a S \leq \texttt{rec}\ t.T$ *and* $(S \leq \texttt{rec}\ t.T) \notin \Sigma$. *Then*

$$\Sigma \cup \{S \leq \texttt{rec}\ t.T\} \models_a U_1 \leq U_2 \quad \textit{implies} \quad \Sigma \models_a U_1 \leq U_2$$

Proof: We only prove this for the first case. Use induction on the length of a derivation of $\Sigma \cup \{\texttt{rec}\ s.S \leq T\} \models_a U_1 \leq U_2$.

Say the last rule used was the SUB-ASS rule. There are two cases. If $U_1 \leq U_2 \in \Sigma$, then the rule still applies with the smaller premise Σ. Say $U_1 = \texttt{rec}\ s.S$ and $U_2 = T$. Now, we know $\models_a \texttt{rec}\ s.S \leq T$, so using the Lemma C.2.6 we have $\Sigma \models_a \texttt{rec}\ s.S \leq T$.

In the cases where the last rule is not the SUB-ASS rule, the premises are all of the form $\Sigma' \cup \{\texttt{rec}\ s.S \leq T\} \models_a U'_1 \leq U'_2$. Any assumption added in Σ' not in Σ will definitely not be $\texttt{rec}\ s.S \leq T$, therefore $\texttt{rec}\ s.S \leq T \notin \Sigma'$. Thus the induction hypothesis applies to the premise, giving us $\Sigma' \models_a U'_1 \leq U'_2$. The last rule will still apply, giving us $\Sigma \models_a U_1 \leq U_2$ as required. ■

Lemma C.2.8

1. *If* $\models_a \texttt{rec}\ s.S \leq T$ *then* $\models_a S\{\!|{}^{\texttt{rec}\ s.S}/s|\!\} \leq T$

2. *If* $\models_a S \leq \texttt{rec}\ t.T$ *and* S *is not of recursive form* *then* $\models_a S \leq T\{\!|{}^{\texttt{rec}\ t.T}/t|\!\}$

3. *If* $\models_a \texttt{rec}\ s.S \leq \texttt{rec}\ t.T$ *then* $\models_a S\{\!|{}^{\texttt{rec}\ s.S}/s|\!\} \leq T\{\!|{}^{\texttt{rec}\ t.T}/T|\!\}$

Proof:

1. Consider the last rule used in $\models_a \texttt{rec}\ s.S \leq T$. It is either SUB-RFL or SUB-RECLFT and we consider the SUB-RFL case first, so $T = \texttt{rec}\ s.S$. Also by SUB-RFL, we have

   $$\{S\{\!|{}^{\texttt{rec}\ s.S}/s|\!\} \leq \texttt{rec}\ s.T\} \models_a S\{\!|{}^{\texttt{rec}\ s.S}/s|\!\} \leq S\{\!|{}^{\texttt{rec}\ s.S}/s|\!\}$$

   Using the SUB-RECRHT rule, this gives us

   $$\models_a S\{\!|{}^{\texttt{rec}\ s.S}/s|\!\} \leq \texttt{rec}\ s.S$$

   as required in this case.

   Next we consider the case where the last rule used was SUB-RECLFT. So, the tree had the form:

   $$\frac{\{\texttt{rec}\ s.S \leq T\} \models_a S\{\!|{}^{\texttt{rec}\ s.S}/s|\!\} \leq T}{\models_a \texttt{rec}\ s.S \leq T}$$

   We can use Lemma C.2.7 to remove the assumption from the antecedent, giving us $\models_a S\{\!|{}^{\texttt{rec}\ s.S}/s|\!\} \leq T$ as required in this case.

2. Similarly to part 1, the two cases which can apply are SUB-RFL and the SUB-RECRHT and they can be dealt with in the same way. The extra condition on this case means we can avoid considering the case where the last rule was SUB-RECLFT.

3. We use part 1 and then part 2 to give the result.

■

Lemma C.2.9 *If S and T are closed and guarded types such that $\models_a S \leq T$, then* $\text{Tree}(S) \leq \text{Tree}(T)$.

Proof: Let

$$\mathcal{R} = \{(\text{Tree}(S), \text{Tree}(T)) | \models_a S \leq T \text{ where } S \text{ and } T \text{ are guarded}\}$$

We claim that $\mathcal{R}$ is a tree simulation.

Assume $(\text{Tree}(S), \text{Tree}(T)) \in \mathcal{R}$. We go through the conditions of what it means to be a tree simulation, depending on the construction of T. Say that $T = \texttt{Chan}\langle T_1, \ldots T_n\rangle$. Then $\text{Tree}(T)(\Lambda) = \text{Chan}^n$. The only rules that could establish $\models_a S \leq T$ are either SUB-RFL or SUB-CHCH. The first case is obvious and the second means that $S = \texttt{Chan}\langle S_1, \ldots S_n\rangle$. Thus $\text{Tree}(S)(\Lambda) = \text{Chan}^n$.

For $\mathcal{R}$ to work, we need both

$$(\text{Tree}(S)(i), \text{Tree}(T)(i)) \in \mathcal{R}$$
$$(\text{Tree}(T)(i), \text{Tree}(S)(i)) \in \mathcal{R}$$

Now $\text{Tree}(S)(i) = \text{Tree}(S_i)$ and $\text{Tree}(T)(i) = \text{Tree}(T_i)$. The premises of the rule give us $\models_a S_i \leq T_i$ and $\models_a T_i \leq S_i$. By Lemma C.2.8 we can unfold S_i and T_i into guarded terms S_i' and T_i' such that $\models_a S_i' \leq T_i'$. $\models_a S_i' \leq T_i'$ and $\models_a T_i' \leq S_i'$, so $(\text{Tree}(S_i'), \text{Tree}(T_i')) \in \mathcal{R}$ and $(\text{Tree}(T_i'), \text{Tree}(S_i')) \in \mathcal{R}$. By the definition of the Tree function, unfolding a type does not affect it's tree, so we have $(\text{Tree}(S)(i), \text{Tree}(T)(i)) \in \mathcal{R}$ and $(\text{Tree}(T)(i), \text{Tree}(S)(i)) \in \mathcal{R}$. This gives us this case.

Since S and T are guarded the other cases are all very similar to the one given. ■

5.3.3 p140

Theorem C.2.10 ***(Soundness)*** *If* $\Gamma \vdash S \leq T$ *then* $\Gamma \models S \leq T$.

Proof: $\Gamma \vdash S \leq T$ means $\vdash_a \mathrm{E}^{\emptyset}_{\Gamma}(S) \leq \mathrm{E}^{\emptyset}_{\Gamma}(T)$. $\mathrm{E}^{\emptyset}_{\Gamma}(S)$ and $\mathrm{E}^{\emptyset}_{\Gamma}(T)$ are closed and by Lemma C.2.8 we can unfold them to give guarded types S' and T' such that $\vdash_a S' \leq T'$. By Lemma C.2.9, this gives us $\mathrm{Tree}(S') \leq \mathrm{Tree}(T')$.

Now, by the definition of the Tree function, unfolding does not affect trees, so Theorem C.1.3 gives us $\mathrm{Tree}_{\Gamma}(S) \leq \mathrm{Tree}_{\Gamma}(T)$, which is what $\Gamma \models S \leq T$ means. ■

We will need the following lemma in the proof of Theorem C.3.6.

Lemma C.2.11 *Say* $\Gamma \vdash S \leq T$ *and* $\Gamma \vdash T \leq U$. *Then* $\Gamma \vdash S \leq U$.

Proof: By soundness, we know that $\Gamma \models S \leq T$ and $\Gamma \models T \leq U$. By inspection of the rules of tree simulation, it is clear that tree simulation is transitive. This means that $\Gamma \models S \leq U$. By completeness, we have $\Gamma \vdash S \leq U$ as required. ■

C.3 Soundness of the Type Safety System

In this section, we provide the proofs of results 5.4.1–5.4.4.

We introduce an abbreviation to help with the readability of the proofs of the following theorems. Every object o given a type in the type dictionary must be given a class type. Say $o{:}\, C \in \Phi$ and say C's definition in Γ is

$$\texttt{class}\ C\ \{a_1{:}\,\mathrm{AtT}_1 \ldots a_n{:}\,\mathrm{AtT}_n\ \mathrm{mdef}_1 \ldots \mathrm{mdef}_{n'}\}$$

Then we let Att_o be the set of attribute typings of o's class, i.e.

$$\mathrm{Att}_o = \{a_1{:}\,\mathrm{AtT}_1, \ldots a_n{:}\,\mathrm{AtT}_n\}$$

5.4.1
p145

Lemma C.3.1 *If* $\Gamma \vdash^{\triangleright}_{\mathrm{d}} g\{^{\Phi}_{\Delta}$ *then* $\Gamma \triangleright g\{^{\Phi}_{\Delta}$ *is not type violating.*

Proof: Say $o[p]$ is an agent in g. We know $\Gamma \vdash^{\triangleright}_{\mathrm{d}} g\{^{\Phi}_{\Delta}$, so $\Gamma, \Phi \vdash g$ and therefore $\Gamma, \Phi \vdash^{\triangleright}_{\mathrm{d}} o[p]$. We consider the form of $o[p]$. Say that $o[p]$ is invoking a method m of an object o', i.e. $p = o'.m!\langle v_1, \dots v_n\rangle?(r_1, \dots r_{n'})\ p'$. Thus we have

$$\Gamma, \Phi \cup \mathrm{Att}_o \vdash^{\triangleright}_{\mathrm{d}} o'.m!\langle v_1, \dots v_n\rangle?(r_1, \dots r_{n'})\ p'$$

By DTS-INV, this means that we must have

$$\Gamma, \Phi \cup \mathrm{Att}_o \vdash o' \colon \mathtt{Intf}\{m?(T_1, \dots T_n)!\langle T'_1, \dots T'_{n'}\rangle\}$$

and $\Gamma, \Phi \cup \mathrm{Att}_o \vdash v_i \colon T_i$ for each $0 \leq i \leq n$.

Now from the first statement, we can conclude that o''s class in Φ must have a method m with signature type $m?(S_1, \dots S_n)!\langle S'_1 \dots S'_{n'}\rangle$ where $\Gamma \vdash T_i \leq S_i$ for each i. Thus the agent cannot be performing a feature or arity mismatch. Using the TAS-SUB rule of the typing system, we have

$$\frac{\Gamma \vdash T_i \leq S_i \qquad \Gamma, \Phi \cup \mathrm{Att}_o \vdash v_i \colon T_i}{\Gamma, \Phi \cup \mathrm{Att}_o \vdash v_i \colon S_i}$$

Hence the agent isn't performing a value mismatch. This means that the agent isn't type violating.

The cases for the send and receive are similar. Next, we consider attribute update. Say $o[p]$ is updating an attribute a, i.e. $p = a!v\ p'$. Thus we have

$$\Gamma, \Phi \cup \mathrm{Att}_o \vdash^{\triangleright}_{\mathrm{d}} a!v\ p'$$

By DTS-UPD, we must have $\Gamma, \Phi \cup \mathrm{Att}_o \vdash a \colon \mathtt{Attr}\{T\}$ and $\Gamma, \Phi \cup \mathrm{Att}_o \vdash v \colon T$. Thus, we know that o's class must have an attribute a of type $\mathtt{Attr}\{T\}$. So the agent isn't performing either a feature mismatch or a value mismatch.

The case for attribute access is similar. It follows that no agent in the system is type violating, so $\Gamma \rhd g\{^{\Phi}_{\Delta}$ isn't type violating. ■

Lemma C.3.2 *If* $\Gamma, \Phi \vdash p$ *then* $\Gamma, \Phi \mathrel{\overset{\rhd}{\underset{\mathrm{d}}{\vdash}}} p$.

Proof: A proof tree in the static type safety system is also a proof tree in the dynamic type safety system. In each of the four rules which have changed, we can use an identity substitution to justify instances of their standard form in the dynamic type safety system. ■

5.4.2
p145

Lemma C.3.3 *If* $\Phi_0 \vdash \Gamma$ *then* $\Gamma \mathrel{\overset{\rhd}{\underset{\mathrm{d}}{\vdash}}} g_\Gamma\{^{\Phi_0}_{\emptyset}$.

Proof: We know that the state is fine, since it is empty. We have to consider the initial agents.

Let $\{C_1, \ldots C_n\}$ be those classes in the definition set Γ which have a method with signature main?()!⟨⟩ and let the bodies of those methods be $p_1, \ldots p_n$. Then, $g_\Gamma = \mathit{dummy_1}[p_1]| \ldots \mathit{dummy_n}[p_n]$. By the requirements for the initial system, we know that C_i has no attributes and p_i doesn't use `return`. Since $\Phi_0 \vdash \Gamma$, we know $\Gamma, \Phi_0 \vdash C_i$ for all i. Thus, $\Gamma, \Phi_0 \vdash$ main_def where main_def is the method definition of the main method of C_i. Hence $\Gamma, \Phi_0 \vdash p_i$ and by Lemma C.3.2 we have $\Gamma, \Phi_0 \mathrel{\overset{\rhd}{\underset{\mathrm{d}}{\vdash}}} p_i$. This gives us $\Gamma, \Phi_0 \mathrel{\overset{\rhd}{\underset{\mathrm{d}}{\vdash}}} \mathit{dummy_i}[p_i]$ for all i and, hence, $\Gamma, \Phi_0 \mathrel{\overset{\rhd}{\underset{\mathrm{d}}{\vdash}}} g_\Gamma$. Finally, we can say $\Gamma \mathrel{\overset{\rhd}{\underset{\mathrm{d}}{\vdash}}} g_\Gamma\{^{\Phi_0}_{\emptyset}$ as required. ■

C.3.1 Preservation of Dynamic Type Safety

Before tackling the main results of this section, we need to prove certain properties of the dynamic type safety system.

Lemma C.3.4

1. Weaken: *If* $\Gamma, \Phi \models^{\rhd}_{\mathrm{d}} p$ *and* $\Phi \pitchfork \Phi'$ *then* $\Gamma, \Phi \cup \Phi' \models^{\rhd}_{\mathrm{d}} p$.

2. Strengthen: *If* $\Gamma, \Phi \cup \Phi' \models^{\rhd}_{\mathrm{d}} p$ *and none of the names in* Φ' *occur in* p *then* $\Gamma, \Phi \models^{\rhd}_{\mathrm{d}} p$.

3. Switch:
   If $\Gamma, \Phi \cup \{v{:}T\} \models^{\rhd}_{\mathrm{d}} p$ *and* $\Gamma, \Phi \vdash v'{:}T$ *then* $\Gamma, \Phi \cup \{v{:}T\} \models^{\rhd}_{\mathrm{d}} p\{\!|^{v'}/v|\!\}$.

4. Rename: *If* $\Gamma, \Phi \cup \{v{:}T\} \models^{\rhd}_{\mathrm{d}} p$ *then* $\Gamma, \Phi \cup \{v'{:}T\} \models^{\rhd}_{\mathrm{d}} p\{\!|^{v'}/v|\!\}$ *for new* v'.

5. Weaken 2: *If* $\Gamma, \Phi \cup \{v{:}T\} \models^{\rhd}_{\mathrm{d}} p$ *and* $\Gamma \vdash T' \leq T$ *then* $\Gamma, \Phi \cup \{v{:}T'\} \models^{\rhd}_{\mathrm{d}} p$.

Proof: Cases 1–3 can be proved by a standard inductive argument on the structure of proof trees.

Case 4 is proved in terms of the first three. Say $\Gamma, \Phi \cup \{v{:}T\} \models^{\rhd}_{\mathrm{d}} p$ then $\Gamma, \Phi \cup \{v{:}T\} \cup \{v'{:}T\} \models^{\rhd}_{\mathrm{d}} p$ by Weaken. $\Gamma, \Phi \cup \{v{:}T\} \cup \{v'{:}T\} \vdash v'{:}T$ by TAS-ASS, so $\Gamma, \Phi \cup \{v{:}T\} \cup \{v'{:}T\} \models^{\rhd}_{\mathrm{d}} p\{\!|^{v'}/v|\!\}$ by Switch and $\Gamma, \Phi \cup \{v'{:}T\} \models^{\rhd}_{\mathrm{d}} p\{\!|^{v'}/v|\!\}$ by Strengthen.

Case 5 is proved in terms of the first four. Say $\Gamma, \Phi \cup \{v{:}T\} \models^{\rhd}_{\mathrm{d}} p$ and $\Gamma \vdash T' \leq T$. Then $\Gamma, \Phi \cup \{v{:}T\} \cup \{v'{:}T'\} \models^{\rhd}_{\mathrm{d}} p$ for some new v' by Weaken. By TAS-ASS, we have $\Gamma, \Phi \cup \{v{:}T\} \cup \{v'{:}T'\} \vdash v'{:}T'$ and by TAS-SUB, we have $\Gamma, \Phi \cup \{v{:}T\} \cup \{v'{:}T'\} \vdash v'{:}T$. We use Switch to give us $\Gamma, \Phi \cup \{v{:}T\} \cup \{v'{:}T'\} \models^{\rhd}_{\mathrm{d}} p\{\!|^{v'}/v|\!\}$. Next, Strengthen to $\Gamma, \Phi \cup \{v'{:}T'\} \models^{\rhd}_{\mathrm{d}} p\{\!|^{v'}/v|\!\}$ and Rename v' back to v to give $\Gamma, \Phi \cup \{v{:}T'\} \models^{\rhd}_{\mathrm{d}} p$. ■

Lemma C.3.5 *If we have* $\Gamma, \Phi \models^{\rhd}_{\mathrm{d}} \Delta$ *and* $\Phi \pitchfork \Phi'$, *then* $\Gamma, \Phi \cup \Phi' \models^{\rhd}_{\mathrm{d}} \Delta$.

5.4.3 p145

Theorem C.3.6 *If* $\Phi_0 \vdash \Gamma$ *and* $\Gamma \models^{\rhd}_{\mathrm{d}} g\{^{\Phi}_{\Delta}$ *and* $\Gamma \rhd g\{^{\Phi}_{\Delta} \longrightarrow g'\{^{\Phi'}_{\Delta'}$ *then* $\Gamma \vdash g'\{^{\Phi'}_{\Delta'}$.

Proof: We use induction on the proof tree of $\Gamma \rhd g\{^{\Phi}_{\Delta} \longrightarrow g'\{^{\Phi'}_{\Delta'}$. There are ten cases to consider.

Left Equivalence

Say the last step of the proof tree was an instance of EQV-LFT:

$$\frac{g'' \equiv g \qquad \Gamma \rhd g''\{^{\Phi}_{\Delta} \longrightarrow g'\{^{\Phi'}_{\Delta'}}{\Gamma \rhd g\{^{\Phi}_{\Delta} \longrightarrow g'\{^{\Phi'}_{\Delta'}}$$

The equivalence of g and g'' means that they differ only in the reordering and regrouping of their constituent agents. This reordering and regrouping will not affect type safety, so we have $\Gamma \models^{\rhd}_{\mathrm{d}} g''\{^{\Phi}_{\Delta}$. Thus, the induction hypothesis applies to the right-hand subtree, giving us $\Gamma \models^{\rhd}_{\mathrm{d}} g'\{^{\Phi'}_{\Delta'}$, as required.

Right Equivalence

Say the last step of the proof tree was an instance of EQV-RHT:

$$\frac{g'' \equiv g' \qquad \Gamma \rhd g\{^{\Phi}_{\Delta} \longrightarrow g''\{^{\Phi'}_{\Delta'}}{\Gamma \rhd g\{^{\Phi}_{\Delta} \longrightarrow g'\{^{\Phi'}_{\Delta'}}$$

The induction hypothesis will apply to the right-hand subtree, giving us $\Gamma \models^{\rhd}_{\mathrm{d}} g''\{^{\Phi'}_{\Delta'}$. However, by the equivalence of g'' and g', we have $\Gamma \models^{\rhd}_{\mathrm{d}} g'\{^{\Phi'}_{\Delta}$.

Parallelism

Say the last step of the proof tree was an instance of PAR:

$$\frac{\Gamma \rhd g_1\{^{\Phi}_{\Delta} \longrightarrow g'_1\{^{\Phi'}_{\Delta'}}{\Gamma \rhd (g_1|g_2)\{^{\Phi}_{\Delta} \longrightarrow (g'_1|g_2)\{^{\Phi'}_{\Delta'}}$$

where $g = (g_1|g_2)$ and $g' = (g'_1|g_2)$. By the premise of the theorem, we have $\Gamma \models^{\rhd}_{\mathrm{d}} (g_1|g_2)\{^{\Phi}_{\Delta}$ giving us $\Gamma, \Phi \models^{\rhd}_{\mathrm{d}} (g_1|g_2)$ and $\Gamma, \Phi \models^{\rhd}_{\mathrm{d}} \Delta$. So we have $\Gamma, \Phi \models^{\rhd}_{\mathrm{d}} g_1$ and $\Gamma, \Phi \models^{\rhd}_{\mathrm{d}} g_2$. Thus the induction hypothesis applies to the subtree, giving us $\Gamma \models^{\rhd}_{\mathrm{d}} g'_1\{^{\Phi'}_{\Delta'}$. Therefore $\Gamma, \Phi' \models^{\rhd}_{\mathrm{d}} g'_1$ and $\Gamma, \Phi' \models^{\rhd}_{\mathrm{d}} \Delta'$. The type dictionary, Φ, is only enlarged by the operational semantics, so we can use Lemma C.3.4, case

1, to weaken $\Gamma, \Phi \models^{\rhd}_{\mathrm{d}} g_2$ to give us $\Gamma, \Phi' \models^{\rhd}_{\mathrm{d}} g_2$. Hence $\Gamma, \Phi' \models^{\rhd}_{\mathrm{d}} (g'_1|g_2)$. Thus we have $\Gamma \models^{\rhd}_{\mathrm{d}} g'\{^{\Phi'}_{\Delta'}$ as required.

End

Say the proof tree is an instance of END. This case is trivial as nil is a type safe agent and the state remains unchanged.

Fork

Say the proof tree is an instance of FRK:

$$\Gamma \rhd o[\mathtt{fork}\{p_1\}\ p_2]\{^{\Phi}_{\Delta} \longrightarrow o[p_1]|o[p_2]\{^{\Phi}_{\Delta}$$

By the premise of the theorem $\Gamma \models^{\rhd}_{\mathrm{d}} o[\mathtt{fork}\{p_1\}p_2]\{^{\Phi}_{\Delta}$ so $\Gamma, \Phi \models^{\rhd}_{\mathrm{d}} o[\mathtt{fork}\{p_1\}p_2]$ and $\Gamma, \Phi \models^{\rhd}_{\mathrm{d}} \Delta$. From the former we have $\Gamma, \Phi \cup \mathrm{Att}_o \models^{\rhd}_{\mathrm{d}} \mathtt{fork}\{p_1\}p_2$. By the DTS-FRKrule, we know that $\Gamma, \Phi \cup \mathrm{Att}_o \models^{\rhd}_{\mathrm{d}} p_1$ and $\Gamma, \Phi \cup \mathrm{Att}_o \models^{\rhd}_{\mathrm{d}} p_2$. Therefore $\Gamma, \Phi \models^{\rhd}_{\mathrm{d}} o[p_1]$ and $\Gamma, \Phi \models^{\rhd}_{\mathrm{d}} o[p_2]$ and thus $\Gamma, \Phi \models^{\rhd}_{\mathrm{d}} (o[p_1]|o[p_2])$ The state remains unchanged, so $\Gamma \models^{\rhd}_{\mathrm{d}} (o[p_1]|o[p_2])\{^{\Phi}_{\Delta}$, as required.

New Channel

Say the proof tree is an instance of NEW:

$$\Gamma \rhd o[\mathtt{new}\ c\colon \mathrm{ChT}\ p]\{^{\Phi}_{\Delta} \longrightarrow o[p\{\!|^{c'}/c|\!\}]\{^{\Phi \cup \{c'\colon \mathrm{ChT}\}}_{\Delta}$$

where c' is the new channel name "chosen" by the axiom. By the premise of the theorem, we have $\Gamma \models^{\rhd}_{\mathrm{d}} o[\mathtt{new}\ c\colon \mathrm{ChT}\ p]\{^{\Phi}_{\Delta}$, so $\Gamma, \Phi \models^{\rhd}_{\mathrm{d}} o[\mathtt{new}\ c\colon \mathrm{ChT}\ p]$, and $\Gamma, \Phi \models^{\rhd}_{\mathrm{d}} \Delta$. Thus we have $\Gamma, \Phi \cup \mathrm{Att}_o \models^{\rhd}_{\mathrm{d}} \mathtt{new}\ c\colon \mathrm{ChT}\ p$. By the DTS-NEW rule, we have $\Gamma, \Phi \cup \mathrm{Att}_o \cup \{c''\colon \mathrm{ChT}\} \models^{\rhd}_{\mathrm{d}} p\{\!|^{c''}/c|\!\}$. By Lemma C.3.4, case 4, we can use a different new name, so we pick c', giving us $\Gamma, \Phi \cup \mathrm{Att}_o \cup \{c'\colon \mathrm{ChT}\} \models^{\rhd}_{\mathrm{d}} p\{\!|^{c'}/c|\!\}$. Then $\Gamma, \Phi \cup \{c'\colon \mathrm{ChT}\} \models^{\rhd}_{\mathrm{d}} o[p\{\!|^{c'}/c|\!\}]$. We can

weaken $\Gamma, \Phi \models^{\triangleright}_{\mathrm{d}} \Delta$ to give $\Gamma, \Phi \cup \{c' \colon \mathrm{ChT}\} \models^{\triangleright}_{\mathrm{d}} \Delta$ by Lemma C.3.5. Hence $\Gamma \models^{\triangleright}_{\mathrm{d}} o[p\{\!|{}^{c'}/c|\!\}]\{^{\Phi \cup \{c' : \mathrm{ChT}\}}_{\Delta}$ as required.

Communication

Say the proof tree is an instance of COM:

$$\Gamma \quad \triangleright \quad \begin{array}{l} o_1[c!\langle v_1, \ldots v_n\rangle\ p_1] \mid \\ o_2[c?(r_1\colon T_1, \ldots r_n\colon T_n)\ p_2] \end{array} \quad \left\{ \begin{array}{l} \Phi \\ \Delta \end{array} \right.$$

$$\longrightarrow \quad \begin{array}{l} o_1[p_1] \mid \\ o_2[p_2\{\!|{}^{v_1}/r_1, \ldots {}^{v_n}/r_n|\!\}] \end{array} \quad \left\{ \begin{array}{l} \Phi \\ \Delta \end{array} \right.$$

By the premise of the theorem, we have

$$\Gamma \models^{\triangleright}_{\mathrm{d}} o_1[c!\langle v_1, \ldots v_n\rangle\ p_1] | o_2[c?(r_1\colon T_1, \ldots r_n\colon T_n)\ p_2]\{^{\Phi}_{\Delta}$$

so we must have both

$$\Gamma, \Phi \cup \mathrm{Att}_{o_1} \models^{\triangleright}_{\mathrm{d}} c!\langle v_1, \ldots v_n\rangle\ p_1$$
$$\Gamma, \Phi \cup \mathrm{Att}_{o_2} \models^{\triangleright}_{\mathrm{d}} c?(r_1\colon T_1, \ldots r_n\colon T_n)\ p_2$$

The first of these, by the DTS-SND rule, gives us

$$\Gamma, \Phi \cup \mathrm{Att}_{o_1} \vdash c\colon \mathtt{OuCh}\langle S_1 \ldots S_n\rangle$$

and $\Gamma, \Phi \cup \mathrm{Att}_{o_1} \vdash v_i\colon S_i$ for $1 \leq i \leq n$, and the second gives us

$$\Gamma, \Phi \cup \mathrm{Att}_{o_2} \vdash c\colon \mathtt{InCh}\langle T_1 \ldots T_n\rangle$$

The only way we could assign these two types to c in the typing system would be if $\Gamma \vdash S_i \leq T_i$ so we also have $\Gamma, \Phi \cup \mathrm{Att}_{o_1} \vdash v_i\colon T_i$ using TAS-SUB.

The proof tree of $\Gamma, \Phi \cup \mathrm{Att}_{o_2} \models^{\triangleright}_{\mathrm{d}} c?(r_1\colon T_1, \ldots r_n\colon T_n)\ p_2$ has a right subtree which concludes $\Gamma, \Phi \cup \mathrm{Att}_{o_2} \cup \{r_1'\colon T_1 \ldots r_n'\colon T_n\} \models^{\triangleright}_{\mathrm{d}} p_2\{\!|{}^{r_1'}/r_1 \ldots {}^{r_n'}/r_n|\!\}$. Then,

since the v_i have suitable typings in Φ, we can apply Lemma C.3.4, case 3 and case 2 to use the v_i instead of the r_i to give $\Gamma, \Phi \cup \mathrm{Att}_{o_2} \models^{\triangleright}_{\mathrm{d}} p_2\{\!|^{v_1}/_{r_1} \ldots {}^{v_n}/_{r_n}|\!\}$. Thus $\Gamma, \Phi \models^{\triangleright}_{\mathrm{d}} o_2[p_2\{\!|^{v_1}/_{r_1} \ldots {}^{v_n}/_{r_n}|\!\}]$.

Also, $\Gamma, \Phi \cup \mathrm{Att}_{o_1} \models^{\triangleright}_{\mathrm{d}} c!\langle v_1, \ldots v_n \rangle\ p_1$ has a right subtree which concludes $\Gamma, \Phi \cup \mathrm{Att}_{o_1} \models^{\triangleright}_{\mathrm{d}} p_1$ so $\Gamma, \Phi \models^{\triangleright}_{\mathrm{d}} o_1[p_1]$. Using this and the above result, gives us $\Gamma, \Phi \models^{\triangleright}_{\mathrm{d}} o_1[p_1]|o_2[p_2\{\!|^{v_1}/_{r_1} \ldots {}^{v_n}/_{r_n}|\!\}]$. The state remains unchanged, so we have $\Gamma \models^{\triangleright}_{\mathrm{d}} o_1[p_1]|o_2[p_2\{\!|^{v_1}/_{r_1} \ldots {}^{v_n}/_{r_n}|\!\}]\{^{\Phi}_{\Delta}$ as required.

Method Invocation

Say the proof tree is an instance of INV:

$$\Gamma \quad \triangleright \quad o[o_1.m!\langle v_1, \ldots v_l \rangle?(s_1, \ldots s_n)\ p] \quad \begin{cases} \Phi \\ \Delta \end{cases}$$

$$\longrightarrow \quad \begin{array}{l} o[x?(s_1\colon T_1, \ldots s_n\colon T_n)\ p] \mid \\ o_1[p_1\{\!|^{v_1}/_{r_1}, \ldots {}^{v_l}/_{r_l}, {}^{o_1}/_{\mathtt{this}}, {}^{x}/_{\mathtt{return}}|\!\}] \end{array} \quad \begin{cases} \Phi' \\ \Delta \end{cases}$$

where $\Phi' = \Phi \cup \{x\colon \mathtt{Chan}\langle T_1, \ldots T_n \rangle\}$ and x is the return channel "chosen" by the axiom. The main premise gives us $\Gamma \models^{\triangleright}_{\mathrm{d}} o[o_1.m!\langle v_1, \ldots v_l \rangle?(s_1, \ldots s_n)\ p]\{^{\Phi}_{\Delta}$, so $\Gamma, \Phi \models^{\triangleright}_{\mathrm{d}} o[o_1.m!\langle v_1, \ldots v_l \rangle?(s_1, \ldots s_n)\ p]$, and hence we have

$$\Gamma, \Phi \cup \mathrm{Att}_o \models^{\triangleright}_{\mathrm{d}} o_1.m!\langle v_1, \ldots v_l \rangle?(s_1, \ldots s_n)\ p$$

From this, we can conclude the following three facts:

$$\Gamma, \Phi \cup \mathrm{Att}_o \vdash o_1\colon \mathtt{Intf}\{m?(S_1, \ldots S_n)!\langle S'_1, \ldots S'_l \rangle\} \tag{A}$$

$$\Gamma, \Phi \cup \mathrm{Att}_o \cup \{x'\colon \mathtt{Chan}\langle S'_1 \ldots S'_n \rangle\} \models^{\triangleright}_{\mathrm{d}} x'?(s_1\colon S'_1, \ldots s_n\colon S'_n)\ p \tag{B}$$

$$\Gamma, \Phi \cup \mathrm{Att}_o \vdash v_i\colon S_i \tag{C}$$

Now $o_1\colon C \in \Phi$ and the fact that the invocation rule applied means that C must have a method with a signature of the form

$$m?(r_1\colon U'_1 \ldots r_l\colon U'_l)!\langle y_1\colon T_1 \ldots y_n\colon T_n \rangle$$

By (A), using the subtyping system, we know that

$$\Gamma \vdash m?(U'_1 \ldots U'_l)!\langle T_1 \ldots T_n\rangle \leq m?(S_1, \ldots S_n)!\langle S'_1, \ldots S'_l\rangle$$

which means that $\Gamma \vdash S_i \leq U'_i$ and $\Gamma \vdash T_i \leq S'_i$.

By (B), using DTS-RCV, we have

$$\Gamma, \Phi \cup \mathrm{Att}_o \cup \{x'\colon \mathtt{Chan}\langle S'_1 \ldots S'_n\rangle\} \cup \{s'_1\colon S'_1 \ldots s'_n\colon S'_n\} \vdash^{\triangleright}_{\mathrm{d}} p\{\!|{}^{s'_1}/s_1 \ldots {}^{s'_n}/s_n|\!\}$$

By using Lemma C.3.4, case 2, we can remove x''s typing from the context, to give $\Gamma, \Phi \cup \mathrm{Att}_o \cup \{s'_1\colon S'_1 \ldots s'_n\colon S'_n\} \vdash^{\triangleright}_{\mathrm{d}} p\{\!|{}^{s'_1}/s_1 \ldots {}^{s'_n}/s_n|\!\}$. We can now use Lemma C.3.4, case 5, to strengthen the typings of the s'_i, giving us $\Gamma, \Phi \cup \mathrm{Att}_o \cup \{s'_1\colon T_1 \ldots s'_n\colon T_n\} \vdash^{\triangleright}_{\mathrm{d}} p\{\!|{}^{s'_1}/s_1 \ldots {}^{s'_n}/s_n|\!\}$. Now use Lemma C.3.4, case 1, to give us

$$\Gamma, \Phi \cup \mathrm{Att}_o \cup \{x\colon \mathtt{Chan}\langle T_1 \ldots T_n\rangle\} \cup \{s'_1\colon T_1 \ldots s'_n\colon T_n\} \vdash^{\triangleright}_{\mathrm{d}} p\{\!|{}^{s'_1}/s_1 \ldots {}^{s'_n}/s_n|\!\}$$

from which, using DTS-RCV, we can conclude

$$\Gamma, \Phi \cup \mathrm{Att}_o \cup \{x\colon \mathtt{Chan}\langle T_1 \ldots T_n\rangle\} \vdash^{\triangleright}_{\mathrm{d}} x?(s_1\colon T_1 \ldots s_n\colon T_n)\ p$$

Thus $\Gamma, \Phi \cup \{x\colon \mathtt{Chan}\langle T_1 \ldots T_n\rangle\} \vdash^{\triangleright}_{\mathrm{d}} o[x?(s_1\colon T_1 \ldots s_n\colon T_n)\ p]$.

By (C), we know that $\{v_i\colon U_i\} \in \Phi$ and that $\Gamma \vdash U_i \leq S_i$. Therefore, by Lemma C.2.11, $\Gamma \vdash U_i \leq U'_i$. Now we know that o_1 has a method with a signature $m?(r_1\colon U'_1 \ldots r_l\colon U'_l)!\langle y_1\colon T_1 \ldots y_n\colon T_n\rangle$ and if it has code p_1, we know that

$$\Gamma, \Phi_0 \cup \{r_1\colon U'_1 \ldots r_l\colon U'_l\} \cup \{\mathtt{return}\colon \mathtt{OuCh}\langle T_1 \ldots T_n\rangle\} \cup \{\mathtt{this}\colon C\} \vdash p_1$$

We bring this into the dynamic type safety system and apply Lemma C.3.4 to give

$$\Gamma, \Phi_0 \cup \{v_1\colon U_1 \ldots v_l\colon U_l\} \cup \{x\colon \mathtt{Chan}\langle T_1 \ldots T_n\rangle\} \cup \{o_1\colon C\} \vdash^{\triangleright}_{\mathrm{d}} p_1\{\!|{}^{v_1}/r_1 \ldots {}^{v_l}/r_l, {}^{x}/\mathtt{return}, {}^{o_1}/\mathtt{this}|\!\}$$

and again

$$\Gamma, \Phi \cup \mathrm{Att}_o \cup \{x\colon \mathtt{Chan}\langle T_1 \dots T_n\rangle\} \models^{\triangleright}_{\mathrm{d}} p_1\{\!|{}^{v_1}/r_1 \dots {}^{v_l}/r_l, {}^{x}/\mathtt{return}, {}^{o_1}/\mathtt{this}|\!\}$$

Thus

$$\Gamma, \Phi \cup \{x\colon \mathtt{Chan}\langle T_1 \dots T_n\rangle\} \models^{\triangleright}_{\mathrm{d}} o_1[p_1\{\!|{}^{v_1}/r_1 \dots {}^{v_l}/r_l, {}^{x}/\mathtt{return}, {}^{o_1}/\mathtt{this}|\!\}]$$

Putting this and the earlier fact together, we have

$$\Gamma, \Phi \cup \{x\colon \mathtt{Chan}\langle T_1 \dots T_n\rangle\} \models^{\triangleright}_{\mathrm{d}} \begin{array}{l} o[x?(s_1\colon T_1 \dots s_n\colon T_n)\ p]| \\ o_1[p_1\{\!|{}^{v_1}/r_1 \dots {}^{v_l}/r_l, {}^{x}/\mathtt{return}, {}^{o_1}/\mathtt{this}|\!\}] \end{array}$$

The state hasn't changed, so we have $\Gamma \models^{\triangleright}_{\mathrm{d}} g'\{^{\Phi'}_{\Delta'}$ as required.

Object Creation

Say the proof tree is an instance of CRT:

$$\Gamma \quad \triangleright \quad o[\mathtt{create}\ o_1\colon C\ p] \begin{cases} \Phi \\ \Delta \end{cases}$$

$$\longrightarrow \quad o[p\{\!|{}^{o'_1}/o_1|\!\}] \begin{cases} \Phi \cup \{o'_1\colon C\} \\ \Delta \cup \{o'_1 = \emptyset\} \end{cases}$$

where o'_1 is the new object name "chosen" by the axiom. From the main premise of the theorem we have $\Gamma \models^{\triangleright}_{\mathrm{d}} o[\mathtt{create}\ o_1\colon C\ p]\{^{\Phi}_{\Delta}$ which gives us both $\Gamma, \Phi \models^{\triangleright}_{\mathrm{d}} o[\mathtt{create}\ o_1\colon C\ p]$ and $\Gamma, \Phi \models^{\triangleright}_{\mathrm{d}} \Delta$. So $\Gamma, \Phi \cup \mathrm{Att}_o \models^{\triangleright}_{\mathrm{d}} \mathtt{create}\ o_1\colon C\ p$. This implies that $\Gamma, \Phi \cup \mathrm{Att}_o \cup \{o''_1\colon C\} \models^{\triangleright}_{\mathrm{d}} p\{\!|{}^{o''_1}/o_1|\!\}$. So by Lemma C.3.4, case 4, we can use o'_1 instead o''_1 to give $\Gamma, \Phi \cup \mathrm{Att}_o \cup \{o'_1\colon C\} \models^{\triangleright}_{\mathrm{d}} p\{\!|{}^{o'_1}/o_1|\!\}$ so $\Gamma, \Phi \cup \{o'_1\colon C\} \models^{\triangleright}_{\mathrm{d}} o[p\{\!|{}^{o'_1}/o_1|\!\}]$. The addition of the empty assignment to the typing state, will not affect it's dynamic type safety, so we have:

$$\Gamma \models^{\triangleright}_{\mathrm{d}} o[p\{\!|{}^{o'_1}/o_1|\!\}]\{^{\Phi \cup \{o'_1\colon C\}}_{\Delta \cup \{o'_1 = \emptyset\}}$$

as required.

Attribute Access

Say the proof tree is an instance of ACC:

$$\Gamma \rhd o[a?r\ p]\{^{\Phi}_{\Delta} \longrightarrow p\{\!|^{v}/r|\!\}\{^{\Phi}_{\Delta}$$

where $v = \Delta(o)(a)$. The premise of the theorem gives us $\Gamma \models^{\rhd}_{\mathrm{d}} o[a?r\ p]\{^{\Phi}_{\Delta}$ so we have $\Gamma, \Phi \models^{\rhd}_{\mathrm{d}} o[a?r\ p]$ and $\Gamma, \Phi \models^{\rhd}_{\mathrm{d}} \Delta$. From the first statement, we can infer $\Gamma, \Phi \cup \mathrm{Att}_o \models^{\rhd}_{\mathrm{d}} a?r\ p$. By DTS-ACC, this implies $\Gamma, \Phi \cup \mathrm{Att}_o \vdash a{:}\,\mathtt{Attr}\{T\}$ and $\Gamma, \Phi \cup \mathrm{Att}_o \cup \{r'{:}\,T\} \models^{\rhd}_{\mathrm{d}} p\{\!|^{r'}/r|\!\}$. Since the state is dynamically type safe, we have $\Gamma, \Phi \cup \mathrm{Att}_o \cup \{r'{:}\,T\} \vdash v{:}\,T$. Then by Lemma C.3.4, case 3 and case 2, $\Gamma, \Phi \cup \mathrm{Att}_o \models^{\rhd}_{\mathrm{d}} p\{\!|^{v}/r|\!\}$. So $\Gamma, \Phi \models^{\rhd}_{\mathrm{d}} o[p\{\!|^{v}/r|\!\}]$. Thus $\Gamma \models^{\rhd}_{\mathrm{d}} o[p\{\!|^{v}/r|\!\}]\{^{\Phi}_{\Delta}$ as required.

Attribute Update

Say the proof tree is an instance of DTS-UPD:

$$\Gamma \rhd o[a!v\ p]\{^{\Phi}_{\Delta} \longrightarrow o[p]\{^{\Phi}_{\Delta'}$$

where $o \in \mathrm{Dom}(\Delta)$ and $\Delta'(o_1)(a_1) = \Delta(o_1)(a_1)$ if $o_1 \neq o$ or $a_1 \neq a$ and v otherwise. The premise of the theorem gives us: $\Gamma \models^{\rhd}_{\mathrm{d}} o[a!v\ p]\{^{\Phi}_{\Delta}$ from which we deduce $\Gamma, \Phi \models^{\rhd}_{\mathrm{d}} o[a!v\ p]$ and $\Gamma, \Phi \models^{\rhd}_{\mathrm{d}} \Delta$. From the first of these we get $\Gamma, \Phi \cup \mathrm{Att}_o \models^{\rhd}_{\mathrm{d}} a!v\ p$. By DTS-UPD, this gives us $\Gamma, \Phi \cup \mathrm{Att}_o \vdash v{:}\,T$, $\Gamma, \Phi \cup \mathrm{Att}_o \vdash a{:}\,\mathtt{Attr}\{T\}$ and $\Gamma, \Phi \cup \mathrm{Att}_o \models^{\rhd}_{\mathrm{d}} p$. This last point gives us $\Gamma, \Phi \models^{\rhd}_{\mathrm{d}} o[p]$. Considering state, we can see that $\Gamma, \Phi \models^{\rhd}_{\mathrm{d}} \Delta'$, since v is appropriately typed. This, with the above dynamic agent type safety statement, gives us $\Gamma \models^{\rhd}_{\mathrm{d}} o[p]\{^{\Phi}_{\Delta'}$ as required.

That was the last of the axioms, so we're done. ■

5.4.4 p145

Theorem C.3.7 ***(Type Safety)*** *If* $\Phi_0 \vdash \Gamma$ *and* $\Gamma \rhd g_\Gamma\{^{\Phi_0}_{\emptyset} \Longrightarrow g\{^{\Phi}_{\Delta}$ *then* $\Gamma \mathrel{\substack{\rhd\\ \models\\ \mathrm{d}}} g\{^{\Phi}_{\Delta}$ *is not type violating.*

Proof: This follows from Lemma C.3.3 and Theorem C.3.6, using an obvious inductive argument on the derivation of $\Gamma \rhd g_\Gamma\{^{\Phi_0}_{\emptyset} \Longrightarrow g\{^{\Phi}_{\Delta}$. ■

C.4 Configurations and the Type System

In this section, we provide the proofs of Theorem 5.4.5 and Corollary 5.4.6.

Lemma C.4.1

$$\Gamma, \Phi \vdash v{:}\, T \Rightarrow \Gamma, \Phi\rho \vdash v\rho{:}\, T$$

Proof: By a simple induction on the proof tree of $\Gamma, \Phi \vdash v{:}\, T$. ■

Lemma C.4.2

$$\Gamma, \Phi \models^{\rhd}_{\mathrm{d}} \Delta \Rightarrow \Gamma, \Phi\rho \models^{\rhd}_{\mathrm{d}} \Delta\rho$$

Proof:

$\Gamma, \Phi \models^{\rhd}_{\mathrm{d}} \Delta$

$\Rightarrow$ $\langle$dynamic type safety of state dictionaries$\rangle$

$\forall o \in \mathrm{Dom}(\Phi).\exists C.$

$(o{:}\, C \in \Phi$

$\wedge \mathtt{class}\ C\ \{a_1 : \mathtt{Attr}\{T_1\}, \ldots g_n : \mathtt{Attr}\{T_n\}\} \in \Gamma$

$\wedge \forall a \in \mathrm{Dom}(\Delta(o)).(\exists i.a = a_i \wedge \Gamma, \Phi \vdash o(a){:}\, T_i))$

$\Rightarrow$ $\langle$renaming, set theory, logic$\rangle$

$\forall o' \in \mathrm{Dom}(\Phi\rho).\exists C.$

$(o'{:}\, C \in \Phi\rho$

$$\wedge \texttt{class}\ C\ \{a_1 : \texttt{Attr}\{T_1\}, \ldots g_n : \texttt{Attr}\{T_n\}\} \in \Gamma$$
$$\wedge \forall a \in \mathrm{Dom}(\Delta\rho(o')).(\exists i.a = a_i \wedge \Gamma, \Phi\rho \vdash o'(a) : T_i))$$
$$\Rightarrow \quad \langle \text{dynamic type safety of state dictionaries} \rangle$$
$$\Gamma, \Phi\rho \models^{\triangleright}_{\mathrm{d}} \Delta\rho$$

■

Lemma C.4.3

$$\Gamma, \Phi \models^{\triangleright}_{\mathrm{d}} p \Rightarrow \Gamma, \Phi\rho \models^{\triangleright}_{\mathrm{d}} p\rho$$

Proof: By induction on the proof of $\Gamma, \Phi \models^{\triangleright}_{\mathrm{d}} p$. We only consider one case here.

Say that the last step of the proof is an instance of the new channel rule

$$\frac{\Gamma, \Phi \cup \{c' : T\} \models^{\triangleright}_{\mathrm{d}} p'\{\!|^{c'}/c|\!\}}{\Gamma, \Phi \models^{\triangleright}_{\mathrm{d}} \texttt{new}\ c : T\ p'}$$

So $p = \texttt{new}\ c : T\ p'$, hence

$$\begin{aligned} p\rho &= (\texttt{new}\ c : T\ p')\rho \\ &= \texttt{new}\ (c\delta\rho) : T\ (p'\delta\rho) \end{aligned}$$

Say that $c\delta = c''$, and let c''' be new. We note that $p\delta = p\{\!|^{c''}/c|\!\}$ because the shove will only effect c. Then, by the induction hypothesis three times, we have

$$\Gamma, (\Phi \cup \{c' : T\})(c'c'')\rho(c''c''') \models^{\triangleright}_{\mathrm{d}} (p'\{\!|^{c'}/c|\!\})(c'c'')\rho(c''c''')$$

i.e.

$$\begin{aligned} \Gamma, \Phi\rho \cup \{c''' : T\} &\models^{\triangleright}_{\mathrm{d}} p'\{\!|^{c''}/c|\!\}\rho(c''c''') \\ &= p'\delta\rho\{\!|^{c'''}/c''|\!\} \end{aligned}$$

We can apply the dynamic type safety rule for new channel to this to give

$$\begin{aligned} \Gamma, \Phi\rho &\models^{\triangleright}_{\mathrm{d}} \texttt{new}\ c'' : T\ p'\delta\rho \\ &= \texttt{new}\ c\delta\rho : T\ p'\delta\rho \\ &= (\texttt{new}\ c : T\ p')\rho \end{aligned}$$

which gives us the result in this case. ■

Lemma C.4.4

$$\Gamma, \Phi \models^{\triangleright}_{\mathrm{d}} g \Rightarrow \Gamma, \Phi\rho \models^{\triangleright}_{\mathrm{d}} g\rho$$

Proof: If $g = \mathrm{nil}$, then the result is obvious, so we assume that g contains one or more primitive agents. Say $o[p] \in g$. Then $\Gamma, \Phi \models^{\triangleright}_{\mathrm{d}} o[p]$ so $\Gamma, \Phi \cup \mathrm{Att}_o \models^{\triangleright}_{\mathrm{d}} p$ so, by Lemma C.4.3, we have $\Gamma, \Phi\rho \cup \mathrm{Att}_{o\rho} \models^{\triangleright}_{\mathrm{d}} p\rho$. This means that $\Gamma, \Phi\rho \models^{\triangleright}_{\mathrm{d}} o\rho[p\rho]$ i.e. $\Gamma, \Phi\rho \models^{\triangleright}_{\mathrm{d}} (o[p])\rho$. Since this is true for all the agents of g, we have our result. ■

Corollary C.4.5

$$\Gamma \models^{\triangleright}_{\mathrm{d}} g\{^{\Phi}_{\Delta} \Rightarrow \Gamma \models^{\triangleright}_{\mathrm{d}} g\rho\{^{\Phi\rho}_{\Delta\rho}$$

Proof: This is an immediate consequence of lemmas C.4.4 and C.4.2. ■

5.4.5 p146

Theorem C.4.6 *Say* $\Phi \vdash \Gamma$ *and* $\Gamma, \Phi \models^{\blacktriangleright}_{\mathrm{d}} K$ *then*

1. *If* $\Gamma, \Phi \blacktriangleright K \longrightarrow K'$ *then* $\Gamma, \Phi \models^{\blacktriangleright}_{\mathrm{d}} K'$.
2. *If* $\Gamma, \Phi \blacktriangleright K \xrightarrow{c!\langle v_1, \ldots v_n\rangle} K'$ *then* $\Gamma, \Phi \models^{\blacktriangleright}_{\mathrm{d}} K'$.
3. *If* $\Gamma, \Phi \blacktriangleright K \xrightarrow{c?\langle v_1, \ldots v_n\rangle} K'$ *and* $c?\langle v_1, \ldots v_n\rangle$ *is a type safe reception in* Γ, Φ, *then* $\Gamma, \Phi \models^{\blacktriangleright}_{\mathrm{d}} K'$.
4. *If* $\Gamma, \Phi \blacktriangleright K \xrightarrow[x:T]{o.m!\langle v_1, \ldots v_n\rangle} K'$ *then* $\Gamma, \Phi \models^{\blacktriangleright}_{\mathrm{d}} K'$.
5. *If* $\Gamma, \Phi \blacktriangleright K \xrightarrow[x:T]{o.m?\langle v_1, \ldots v_n\rangle} K'$ *and* $o.m?\langle v_1, \ldots v_n\rangle$ *is a type safe reception in* Γ, Φ, *then* $\Gamma, \Phi \models^{\blacktriangleright}_{\mathrm{d}} K'$.

Proof: Consider the first case. Say

$$\Gamma, \Phi \blacktriangleright K_1 \longrightarrow K_2$$

and

$$\Gamma, \Phi \models^{\blacktriangleright}_{\mathrm{d}} K_1$$

and

$$\begin{array}{rcl}\mathrm{Flatten}(K_1\{^{\Phi}_{\emptyset}) & = & g_1\{^{\Phi_1}_{\Delta_1} \\ \mathrm{Flatten}(K_2\{^{\Phi}_{\emptyset}) & = & g_2\{^{\Phi_2}_{\Delta_2}\end{array}$$

Then

$$\Gamma \models^{\triangleright}_{\mathrm{d}} g_1\{^{\Phi_1}_{\Delta_1}$$

By Theorem B.4.11, there exists some renaming ρ such that

$$\Gamma \triangleright g_1\{^{\Phi_1}_{\Delta_1} \longrightarrow g_2\rho\{^{\Phi_2\rho}_{\Delta_2\rho}$$

By Theorem C.3.6, we know

$$\Gamma \models^{\triangleright}_{\mathrm{d}} g_2\rho\{^{\Phi_2\rho}_{\Delta_2\rho}$$

and by Lemma C.4.5, this means

$$\Gamma \models^{\triangleright}_{\mathrm{d}} g_2\{^{\Phi_2}_{\Delta_2}$$

i.e.

$$\Gamma \models^{\triangleright}_{\mathrm{d}} \mathrm{Flatten}(K_2\{^{\Phi}_{\emptyset})$$

which means $\Gamma, \Phi \models^{\blacktriangleright}_{\mathrm{d}} K_2$ as required.

Now consider the second case. So we have

$$\Gamma, \Phi \blacktriangleright K_1 \xrightarrow{c!\langle v_1,\ldots v_n\rangle} K_2$$

and

$$\Gamma, \Phi \models^{\blacktriangleright}_{\mathrm{d}} K_1$$

and

$$\begin{array}{rcl}\mathrm{Flatten}(K_1) & = & g_1\{^{\Phi_1}_{\Delta_1} \\ \mathrm{Flatten}(K_2) & = & g_2\{^{\Phi_2}_{\Delta_2} \\ \mathrm{Flatten}(K_1\{^{\Phi}_{\emptyset}) & = & g_1\rho_1\{^{\Phi_1\rho_1\cup\Phi}_{\Delta_1\rho_1} \\ \mathrm{Flatten}(K_2\{^{\Phi}_{\emptyset}) & = & g_2\rho_2\{^{\Phi_2\rho_2\cup\Phi}_{\Delta_2\rho_2}\end{array}$$

By Lemma B.4.7, we know there exists a renaming ρ such that

$$\begin{array}{rcl} \Phi_2\rho & = & \Phi_1 \\ \Delta_2\rho & = & \Delta_1 \\ g_1 & \overset{\triangleright}{\equiv} & o[c!\langle v_1,\ldots v_n\rangle\ p]|g \\ g_2\rho & \overset{\triangleright}{\equiv}_\alpha & o[p]|g \end{array}$$

Now

$$\Gamma \models^{\triangleright}_{\mathrm{d}} g_1\rho_1\{^{\Phi_1\rho_1\cup\Phi}_{\Delta_1\rho_1}$$

so

$$\begin{array}{rcl} \Gamma,\Phi_1\rho_1\cup\Phi & \models^{\triangleright}_{\mathrm{d}} & \Delta_1\rho_1 \\ \Gamma,\Phi_1\rho_1\cup\Phi & \models^{\triangleright}_{\mathrm{d}} & g_1\rho_1 \end{array}$$

so

$$\Gamma,\Phi_1\rho_1\cup\Phi \models^{\triangleright}_{\mathrm{d}} (o[c!\langle v_1,\ldots v_n\rangle\ p]|g)\rho_1$$

hence

$$\begin{array}{rcl} \Gamma,\Phi_1\rho_1\cup\Phi & \models^{\triangleright}_{\mathrm{d}} & (o\rho_1)[c!\langle v_1,\ldots v_n\rangle\ (p\rho_1)] \\ \Gamma,\Phi_1\rho_1\cup\Phi & \models^{\triangleright}_{\mathrm{d}} & g\rho_1 \end{array}$$

Thus

$$\Gamma,\Phi_1\rho_1\cup\Phi\cup\mathrm{Att}_{o\rho_1} \models^{\triangleright}_{\mathrm{d}} c!\langle v_1,\ldots v_n\rangle\ (p\rho_1)$$

from which we can conclude

$$\begin{array}{rcl} \Gamma,\Phi_1\rho_1\cup\Phi\cup\mathrm{Att}_{o\rho_1} & \models^{\triangleright}_{\mathrm{d}} & p\rho_1 \\ \Gamma,\Phi_1\rho_1\cup\Phi & \models^{\triangleright}_{\mathrm{d}} & o\rho_1[p\rho_1] \\ \Gamma,\Phi_1\rho_1\cup\Phi & \models^{\triangleright}_{\mathrm{d}} & o\rho_1[p\rho_1]|g\rho_1 \\ \Gamma & \models^{\triangleright}_{\mathrm{d}} & (o\rho_1[p\rho_1]|g\rho_1)\{^{\Phi_1\rho_1\cup\Phi}_{\Delta_1\rho_1} \end{array}$$

Let

$$\rho^\alpha = (\rho_2\circ\rho^{-1}\circ{\rho_1}^{-1})\big|_{\Phi_1\rho_1} + 1|_\Phi$$

Then, by Lemma C.4.5, we have

$$\Gamma \models^{\triangleright}_{\mathrm{d}} (o\rho_1[p\rho_1]|g\rho_1)\rho^\alpha\{^{(\Phi_1\rho_1\cup\Phi)\rho^\alpha}_{\Delta_1\rho_1\rho^\alpha}$$

which gives us

$$\Gamma \models^{\triangleright}_{\mathrm{d}} g_2\rho_2\{^{\Phi_2\rho_2\cup\Phi}_{\Delta_2\rho_2}$$

i.e.

$$\Gamma \models^{\triangleright}_{\mathrm{d}} \mathrm{Flatten}(K_2\{^{\Phi}_{\emptyset})$$

which means $\Gamma, \Phi \models^{\blacktriangleright}_{\mathrm{d}} K_2$ as required. ■

Corollary C.4.7 *If* $\Gamma, \Phi \models^{\blacktriangleright}_{\mathrm{d}} K_1$ *and* $\Gamma, \Phi \blacktriangleright K_1 \xRightarrow[L_2]{L_1} K_2$ *and all the receptions in* L_1 *are type safe, then* $\Gamma, \Phi \models^{\blacktriangleright}_{\mathrm{d}} K_2$.

5.4.6 p147

Appendix D

Simulations

This appendix provides many of the simulations mentioned in the main text. The definition of satisfying simulation can be found on page 167. The definition of refining simulation can be found on page 178.

Section D.2 gives the satisfying simulations used in the Scheduler example of Chapter 6. Section D.1 gives the refining simulations used in that example. In Section D.3, we give the refining simulations used for the concurrent dictionary example of Chapter 7.

D.1 Refining Simulations for the Scheduler Example

First we justify the statement $\text{Sched}^0 \sqsubseteq \text{Sched}^1_\emptyset$, given the definitions:

$$\begin{array}{lcl} \text{Sched}^0 & \stackrel{\text{def}}{=} & \bigsqcap_{j \in 1..n} b_j.\text{Sched}^0 \sqcap \bigsqcap_{i \in 1..n} a_i.\text{Sched}^0 \\ \text{Sched}^1_X & \stackrel{\text{def}}{=} & \bigsqcap_{j \in X} b_j.\text{Sched}^1_{X-j} \sqcap \bigsqcap_{i \notin X} a_i.\text{Sched}^1_{X \cup i} \end{array}$$

where $X \subseteq \{1..n\}$. We use the following refining simulation:

$$\begin{array}{ll}
 & \{(\mathrm{Sched}^0, \mathrm{Sched}^1_X) \mid X \subseteq \{1..n\}\} \\
\cup & \{(\bigsqcap_{j\in 1..n} b_j.\mathrm{Sched}^0 \sqcap \bigsqcap_{i\in 1..n} a_i.\mathrm{Sched}^0, \mathrm{Sched}^1_X) \mid X \subseteq \{1..n\}\} \\
\cup & \{(\bigsqcap_{j\in 1..n} b_j.\mathrm{Sched}^0, \bigsqcap_{j\in X} b_j.\mathrm{Sched}^1_{X-j}) \mid \emptyset \neq X \subseteq \{1..n\}\} \\
\cup & \{(\bigsqcap_{i\in 1..n} a_i.\mathrm{Sched}^0, \bigsqcap_{i\notin X} a_i.\mathrm{Sched}^1_{X\cup i}) \mid \{1..n\} \neq X \subseteq \{1..n\}\} \\
\cup & \{(b_j.\mathrm{Sched}^0, b_j.\mathrm{Sched}^1_{X-j}) \mid j \in X \subseteq \{1..n\}, \} \\
\cup & \{(a_i.\mathrm{Sched}^0, a_i.\mathrm{Sched}^1_{X\cup i}) \mid i \notin X \subseteq \{1..n\}, \}
\end{array}$$

Next we justify the statement $\mathrm{Sched}^1_\emptyset \sqsubseteq \mathrm{Sched}^2_\emptyset$ given the definition:

$$\mathrm{Sched}^2_X \stackrel{\mathrm{def}}{=} \square_{j\in X} b_j.\mathrm{Sched}^2_{X-j} \,\square\, (\bigsqcap_{i\notin X} a_i.\mathrm{Sched}^2_{X\cup i} \sqcap 0)$$

We use the following refining simulation:

$$\begin{array}{ll}
 & \{(\mathrm{Sched}^1_X, \mathrm{Sched}^2_X) \mid X \subseteq \{1..n\}\} \\
\cup & \{(\bigsqcap_{j\in X} b_j.\mathrm{Sched}^1_{X-j} \sqcap \bigsqcap_{i\notin X} a_i.\mathrm{Sched}^1_{X\cup i}, \mathrm{Sched}^2_X) \mid X \subseteq \{1..n\}\} \\
\cup & \{(\bigsqcap_{j\in X} b_j.\mathrm{Sched}^1_{X-j}, \square_{j\in X} b_j.\mathrm{Sched}^2_{X-j}) \mid \emptyset \neq X \subseteq \{1..n\}\} \\
\cup & \{(\bigsqcap_{i\notin X} a_i.\mathrm{Sched}^1_{X\cup i}, \bigsqcap_{i\notin X} a_i.\mathrm{Sched}^2_{X\cup i}) \mid \{1..n\} \neq X \subseteq \{1..n\}\} \\
\cup & \{(b_j.\mathrm{Sched}^1_{X-j}, b_j.\mathrm{Sched}^2_{X-j}) \mid j \in X \subseteq \{1..n\}\} \\
\cup & \{(a_i.\mathrm{Sched}^1_{X\cup i}, a_i.\mathrm{Sched}^2_{X\cup i}) \mid i \notin X \subseteq \{1..n\}\}
\end{array}$$

Now we consider the statement $\mathrm{Sched}^2_\emptyset \sqsubseteq \mathrm{Sched}^3_\emptyset$ given the definition:

$$\mathrm{Sched}^3_X \stackrel{\mathrm{def}}{=} \begin{cases} \square_{j\in X} b_j.\mathrm{Sched}^3_{X-j} \,\square\, \bigsqcap_{i\notin X} a_i.\mathrm{Sched}^3_{X\cup i} & \text{if } X \neq \{1..n\} \\ \square_{j\in X} b_j.\mathrm{Sched}^3_{X-j} & \text{otherwise} \end{cases}$$

We use the following refining simulation:

$$
\begin{array}{ll}
 & \{(\mathrm{Sched}^2_X, \mathrm{Sched}^3_X) \mid X \subseteq \{1..n\}\} \\
\cup & \{(\Box_{j \in X} b_j.\mathrm{Sched}^2_{X-j} \;\Box\; (\bigsqcap_{i \notin X} a_i.\mathrm{Sched}^2_{X \cup i} \sqcap 0), \mathrm{Sched}^3_X) \mid X \subseteq \{1..n\}\} \\
\cup & \{(\Box_{j \in X} b_j.\mathrm{Sched}^2_{X-j}, \Box_{j \in X} b_j.\mathrm{Sched}^3_{X-j}) \mid \emptyset \neq X \subseteq \{1..n\}\} \\
\cup & \{(\bigsqcap_{i \notin X} a_i.\mathrm{Sched}^2_{X \cup i} \sqcap 0, \bigsqcap_{i \notin X} a_i.\mathrm{Sched}^3_{X \cup i}) \mid \{1..n\} \neq X \subseteq \{1..n\}\} \\
\cup & \{(\bigsqcap_{i \notin X} a_i.\mathrm{Sched}^2_{X \cup i} \sqcap 0, 0) \mid X = \{1..n\}\} \\
\cup & \{(\bigsqcap_{i \notin X} a_i.\mathrm{Sched}^2_{X \cup i}, \bigsqcap_{i \notin X} a_i.\mathrm{Sched}^3_{X \cup i}) \mid \{1..n\} \neq X \subseteq \{1..n\}\} \\
\cup & \{(b_j.\mathrm{Sched}^2_{X-j}, b_j.\mathrm{Sched}^3_{X-j}) \mid j \in X \subseteq \{1..n\}\} \\
\cup & \{(a_i.\mathrm{Sched}^2_{X \cup i}, a_i.\mathrm{Sched}^3_{X \cup i}) \mid i \notin X \subseteq \{1..n\}\}
\end{array}
$$

Next we consider the statement $\mathrm{Sched}^2_\emptyset \sqsubseteq \mathrm{Sched}^4_{1,\emptyset}$ given the definition:

$$
\mathrm{Sched}^4_{i,X} \stackrel{\mathrm{def}}{=} \Box_{j \in X} b_j.\mathrm{Sched}^4_{i,X-j} \;\Box\; (\bigsqcap_{(k \notin X, k=i)} a_k.\mathrm{Sched}^4_{k+1, X \cup k} \sqcap 0)
$$

where $i \in \{1..n\}$. We use the following refining simulation:

$$
\begin{array}{ll}
 & \{(\mathrm{Sched}^2_X, \mathrm{Sched}^4_{i,X}) \mid X \subseteq \{1..n\}, i \in \{1..n\}\} \\
\cup & \{(\Box_{j \in X} b_j.\mathrm{Sched}^2_{X-j} \;\Box\; (\bigsqcap_{i \notin X} a_i.\mathrm{Sched}^2_{X \cup i} \sqcap 0), \mathrm{Sched}^4_{i,X}) \mid X \subseteq \{1..n\}\} \\
\cup & \{(\Box_{j \in X} b_j.\mathrm{Sched}^2_{X-j}, \Box_{j \in X} b_j.\mathrm{Sched}^4_{i,X-j}) \mid \emptyset \neq X \subseteq \{1..n\}, i \in \{1..n\}\} \\
\cup & \{(\bigsqcap_{k \notin X} a_k.\mathrm{Sched}^2_{X \cup k} \sqcap 0, \bigsqcap_{(k \notin X, k=i)} a_k.\mathrm{Sched}^4_{k+1, X \cup k} \sqcap 0) \\
 & \qquad \mid \{1..n\} \neq X \subseteq \{1..n\}, i \in \{1..n\}\} \\
\cup & \{(\bigsqcap_{k \notin X} a_k.\mathrm{Sched}^2_{X \cup k}, \bigsqcap_{(k \notin X, k=i)} a_k.\mathrm{Sched}^4_{k+1, X \cup k}) \\
 & \qquad \mid \{1..n\} \neq X \subseteq \{1..n\}, i \notin X\} \\
\cup & \{(0, 0)\} \\
\cup & \{(b_j.\mathrm{Sched}^2_{X-j}, b_j.\mathrm{Sched}^4_{i,X-j}) \mid j \in X \subseteq \{1..n\}\} \\
\cup & \{(a_k.\mathrm{Sched}^2_{X \cup k}, a_k.\mathrm{Sched}^4_{k+1, X \cup i}) \\
 & \qquad \mid \{1..n\} \neq X \subseteq \{1..n\}, i \in \{1..n\}, i = k \notin X\}
\end{array}
$$

Lastly, we consider the statement $\text{Sched}^4_{1,\emptyset} \sqsubseteq \text{Sched}^5_{1,\emptyset}$ given the definition:

$$\text{Sched}^5_{i,X} \stackrel{\text{def}}{=} \begin{cases} \Box_{j \in X} b_j.\text{Sched}^5_{i,X-j} \;\Box\; a_i.\text{Sched}^5_{i+1,X\cup i} & i \notin \text{X} \\ \Box_{j \in X} b_j.\text{Sched}^5_{i,X-j} & i \in \text{X} \end{cases}$$

We use the following refining simulation:

$$\begin{aligned}
& \{(\text{Sched}^4_{i,X}, \text{Sched}^5_{i,X}) \mid X \subseteq \{1..n\}, i \in \{1..n\}\} \\
\cup \;& \{(\Box_{j\in X} b_j.\text{Sched}^4_{i,X-j} \;\Box\; (\textstyle\prod_{(k \notin X, k=i)} a_k.\text{Sched}^4_{k+1,X\cup k} \sqcap 0), \text{Sched}^5_{i,X}) \\
& \qquad \mid X \subseteq \{1..n\}, i \in \{1..n\}\} \\
\cup \;& \{(\Box_{j\in X} b_j.\text{Sched}^4_{i,X-j}, \Box_{j\in X} b_j.\text{Sched}^5_{i,X-j}) \mid \emptyset \neq X \subseteq \{1..n\}, i \in \{1..n\}\} \\
\cup \;& \{(\textstyle\prod_{(k \notin X, k=i)} a_k.\text{Sched}^4_{k+1,X\cup k} \sqcap 0, a_i.\text{Sched}^5_{i+1,X\cup i}) \\
& \qquad \mid \{1..n\} \neq X \subseteq \{1..n\}, i \notin X\} \\
\cup \;& \{(\textstyle\prod_{(k \notin X, k=i)} a_k.\text{Sched}^4_{k+1,X\cup k} \sqcap 0, 0) \mid \{1..n\} \neq X \subseteq \{1..n\}, i \in X\} \\
\cup \;& \{(\textstyle\prod_{(k \notin X, k=i)} a_k.\text{Sched}^4_{k+1,X\cup k}, a_i.\text{Sched}^5_{i+1,X\cup i}) \\
& \qquad \mid \{1..n\} \neq X \subseteq \{1..n\}, i \notin X\} \\
\cup \;& \{(0,0)\} \\
\cup \;& \{(b_j.\text{Sched}^4_{i,X-j}, b_j.\text{Sched}^5_{i,X-j}) \mid X \subseteq \{1..n\}, j \in X\} \\
\cup \;& \{(a_k.\text{Sched}^4_{k+1,X\cup k}, a_i.\text{Sched}^5_{i+1,X\cup i}) \\
& \qquad \mid X \subseteq \{1..n\}, i \in \{1..n\}, k \notin X, k = i\}
\end{aligned}$$

D.2 Satisfying Simulations for the Scheduler Example

First we justify the statement $\text{design}^1 \;\tilde{\mathsf{sat}}\; \text{Sched}^1_\emptyset$ where we convert design^1 to normal form as follows:

$$\text{d}^1_1 \stackrel{\text{def}}{=} a_1.\text{d}^1_2 \quad \text{d}^1_2 \stackrel{\text{def}}{=} a_2.\text{d}^1_3 \quad \text{d}^1_3 \stackrel{\text{def}}{=} b_2.\text{d}^1_4 \quad \text{d}^1_4 \stackrel{\text{def}}{=} b_1.\text{d}^1_1$$

We use the following satisfying simulation:

$$
\begin{aligned}
&\{(\mathrm{Sched}^1_\emptyset, \mathrm{d}^1_1), (\mathrm{Sched}^1_\emptyset, a_1.\mathrm{d}^1_2),\\
&\quad (\textstyle\bigsqcap_{j\in\emptyset} b_j.\mathrm{Sched}^1_{\emptyset-j} \sqcap \bigsqcap_{i\notin\emptyset} a_i.\mathrm{Sched}^1_{\{i\}}, a_1.\mathrm{d}^1_2), (\bigsqcap_{i\notin\emptyset} a_i.\mathrm{Sched}^1_{\{i\}}, a_1.\mathrm{d}^1_2),\\
&\quad (a_1.\mathrm{Sched}^1_{\{1\}}, a_1.\mathrm{d}^1_2), (\mathrm{Sched}^1_{\{1\}}, \mathrm{d}^1_2), (\mathrm{Sched}^1_{\{1\}}, a_2.\mathrm{d}^1_3),\\
&\quad (\textstyle\bigsqcap_{j\in\{1\}} b_j.\mathrm{Sched}^1_{\{1\}-j} \sqcap \bigsqcap_{i\notin\{1\}} a_i.\mathrm{Sched}^1_{\{1,i\}}, a_2.\mathrm{d}^1_3),\\
&\quad (\textstyle\bigsqcap_{i\notin\{1\}} a_i.\mathrm{Sched}^1_{\{1,i\}}, a_2.\mathrm{d}^1_3), (a_2.\mathrm{Sched}^1_{\{1,2\}}, a_2.\mathrm{d}^1_3),\\
&\quad (\mathrm{Sched}^1_{\{1,2\}}, \mathrm{d}^1_3), (\mathrm{Sched}^1_{\{1,2\}}, b_2.\mathrm{d}^1_4),\\
&\quad (\textstyle\bigsqcap_{j\in\{1,2\}} b_j.\mathrm{Sched}^1_{\{1,2\}-j} \sqcap \bigsqcap_{i\notin\{1,2\}} a_i.\mathrm{Sched}^1_{\{1,2,i\}}, b_2.\mathrm{d}^1_4),\\
&\quad (\textstyle\bigsqcap_{j\in\{1,2\}} b_j.\mathrm{Sched}^1_{\{1,2\}-j}, b_2.\mathrm{d}^1_4), (b_2.\mathrm{Sched}^1_{\{1\}}, b_2.\mathrm{d}^1_4),\\
&\quad (\mathrm{Sched}^1_{\{1\}}, \mathrm{d}^1_4), (\mathrm{Sched}^1_{\{1\}}, b_1.\mathrm{d}^1_1),\\
&\quad (\textstyle\bigsqcap_{j\in\{1\}} b_j.\mathrm{Sched}^1_{\{1\}-j} \sqcap \bigsqcap_{i\notin\{1\}} a_i.\mathrm{Sched}^1_{\{1,i\}}, b_1.\mathrm{d}^1_1),\\
&\quad (\textstyle\bigsqcap_{j\in\{1\}} b_j.\mathrm{Sched}^1_{\{1\}-j}, b_1.\mathrm{d}^1_1), (b_1.\mathrm{Sched}^1_\emptyset, b_1.\mathrm{d}^1_1)\}
\end{aligned}
$$

Next we consider the statement $\mathrm{design}^2\ \tilde{\mathsf{sat}}\ \mathrm{Sched}^1_\emptyset$ where we convert design^2 to normal form as follows:

$$\mathrm{d}^2_1 \stackrel{\mathrm{def}}{=} a_1.\mathrm{d}^2_2 \qquad\qquad \mathrm{d}^2_2 \stackrel{\mathrm{def}}{=} b_1.\mathrm{d}^2_1$$

We use the following satisfying simulation:

$$
\begin{aligned}
&\{(\mathrm{Sched}^1_\emptyset, \mathrm{d}^1_1), (\mathrm{Sched}^1_\emptyset, a_1.\mathrm{d}^2_2),\\
&\quad (\textstyle\bigsqcap_{j\in\emptyset} b_j.\mathrm{Sched}^1_{\emptyset-j} \sqcap \bigsqcap_{i\notin\emptyset} a_i.\mathrm{Sched}^1_{\{i\}}, a_1.\mathrm{d}^2_2), (\bigsqcap_{i\notin\emptyset} a_i.\mathrm{Sched}^1_{\{i\}}, a_1.\mathrm{d}^2_2),\\
&\quad (a_1.\mathrm{Sched}^1_{\{1\}}, a_1.\mathrm{d}^2_2), (\mathrm{Sched}^1_{\{1\}}, \mathrm{d}^2_2), (\mathrm{Sched}^1_{\{1\}}, b_1.\mathrm{d}^2_1),\\
&\quad (\textstyle\bigsqcap_{j\in\{1\}} b_j.\mathrm{Sched}^1_{\{1\}-j} \sqcap \bigsqcap_{i\notin\{1\}} a_i.\mathrm{Sched}^1_{\{1,i\}}, b_1.\mathrm{d}^2_1),\\
&\quad (\textstyle\bigsqcap_{j\in\{1\}} b_j.\mathrm{Sched}^1_{\{1\}-j}, b_1.\mathrm{d}^2_1), (b_1.\mathrm{Sched}^1_\emptyset, b_1.\mathrm{d}^2_1)\}
\end{aligned}
$$

We justify the statement that $\mathrm{design}^1\ \tilde{\not{\mathsf{sat}}}\ \mathrm{Sched}^2_\emptyset$ by showing that no satisfying simulation can exist which contains $(\mathrm{Sched}^2_\emptyset, \mathrm{d}^1_1)$. Say $\mathcal{Q}$ is such a

satisfying simulation. As a consequence of the definition of satisfying simulation, it is easy to show that we must have $(\text{Sched}^2_{\{1\}}, a_2.\text{d}^1_3) \in \mathcal{Q}$. This leads us to having $(\Box_{j\in\{1\}} b_j.\text{Sched}^1_{\{1\}-j} \,\Box\, (\sqcap_{i\notin\{1\}} a_i.\text{Sched}^1_{\{1,i\}} \sqcap 0), a_2.\text{d}^1_3) \in \mathcal{Q}$, which fails the satisfying simulation property. This is a contradiction, so we conclude that $\text{design}^1 \not\widetilde{\mathsf{sat}}\ \text{Sched}^2_\emptyset$.

Next we show that $\text{design}^2\ \widetilde{\mathsf{sat}}\ \text{Sched}^2_\emptyset$. We use the following satisfying simulation:

$$
\begin{aligned}
&\{(\text{Sched}^2_\emptyset, \text{d}^2_1), (\text{Sched}^2_\emptyset, a_1.\text{d}^2_2),\\
&(\Box_{j\in\emptyset} b_j.\text{Sched}^2_{\emptyset-j} \,\Box\, (\sqcap_{i\notin\emptyset} a_i.\text{Sched}^2_{\{i\}} \sqcap 0), a_1.\text{d}^2_2),\\
&(0,0), (\sqcap_{i\notin\emptyset} a_i.\text{Sched}^2_{\{i\}} \sqcap 0, a_1.\text{d}^2_2),\\
&(a_1.\text{Sched}^2_{\{1\}}, a_1.\text{d}^2_2), (\text{Sched}^2_{\{1\}}, \text{d}^2_2), (\text{Sched}^2_{\{1\}}, b_1.\text{d}^2_1),\\
&(\Box_{j\in\{1\}} b_j.\text{Sched}^2_{\{1\}-j} \,\Box\, (\sqcap_{i\notin\{1\}} a_i.\text{Sched}^2_{\{1,i\}} \sqcap 0), b_1.\text{d}^2_1),\\
&(\Box_{j\in\{1\}} b_j.\text{Sched}^2_{\{1\}-j}, b_1.\text{d}^2_1), (\sqcap_{i\notin\{1\}} a_i.\text{Sched}^2_{\{1,i\}} \sqcap 0, 0),\\
&(b_1.\text{Sched}^2_\emptyset, b_1.\text{d}^2_1)\}
\end{aligned}
$$

Now we justify the statement $\text{design}^3_X\ \widetilde{\mathsf{sat}}\ \text{Sched}^3_\emptyset$ given the definition:

$$
\text{design}^3_X \stackrel{\text{def}}{=} \begin{cases} \sum_{j\in X} b_j.\text{design}^3_{X-j} + a_{(\min \overline{X})}.\text{design}^3_{X\cup(\min \overline{X})} & \text{if } X \neq \{1..n\} \\ \sum_{j\in X} b_j.\text{design}^3_{X-j} & \text{otherwise} \end{cases}
$$

We use the following satisfying simulation:

$$
\begin{array}{ll}
& \{(\mathrm{Sched}^3_X, \mathrm{design}^3_X) \mid X \subseteq \{1..n\}\} \\
\cup & \{(\mathrm{Sched}^3_X, \sum_{j\in X} b_j.\mathrm{design}^3_{X-j} + a_{(\min \overline{X})}.\mathrm{design}^3_{X\cup(\min \overline{X})}) \mid X \neq \{1..n\}\} \\
\cup & \{(\mathrm{Sched}^3_X, \sum_{j\in X} b_j.\mathrm{design}^3_{X-j}) \mid X = \{1..n\}\} \\
\cup & \{(\Box_{j\in X} b_j.\mathrm{Sched}^3_{X-j} \;\Box\; \prod_{i\notin X} a_i.\mathrm{Sched}^3_{X\cup i}, \\
& \quad \sum_{j\in X} b_j.\mathrm{design}^3_{X-j} + a_{(\min \overline{X})}.\mathrm{design}^3_{X\cup(\min \overline{X})}) \\
& \qquad |X \neq \{1..n\}\} \\
\cup & \{(\Box_{j\in X} b_j.\mathrm{Sched}^3_{X-j}, \sum_{j\in X} b_j.\mathrm{design}^3_{X-j}) \mid \;\neq X \subseteq \{1..n\}\} \\
\cup & \{(\prod_{i\notin X} a_i.\mathrm{Sched}^3_{X\cup i}, a_{(\min \overline{X})}.\mathrm{design}^3_{X\cup(\min \overline{X})}) \mid X \neq \{1..n\}\} \\
\cup & \{(b_j.\mathrm{Sched}^3_{X-j}, b_j.\mathrm{design}^3_{X-j}) \mid j \in X \subseteq \{1..n\}\} \\
\cup & \{(a_i.\mathrm{Sched}^3_{X\cup i}, a_i.\mathrm{design}^3_{X\cup(\min \overline{X})}) \mid X \neq \{1..n\}, i = (\min \overline{X})\}
\end{array}
$$

Next we justify the statement design^4_1 $\mathsf{s\tilde{a}t}$ $\mathrm{Sched}^4_{1,\emptyset}$. We convert design^4_i to normal form using:

$$
\mathrm{d}^4_{1,i} \stackrel{\mathrm{def}}{=} a_i.\mathrm{d}^4_{2,i} \qquad\qquad \mathrm{d}^4_{2,i} \stackrel{\mathrm{def}}{=} b_i.\mathrm{d}^4_{1,i+1}
$$

We use the following satisfying simulation:

$$
\begin{aligned}
& \{(\mathrm{Sched}^4_{i,\emptyset}, \mathrm{d}^4_{1,i}) \mid i \in \{1..n\}\} \\
\cup\ & \{(\mathrm{Sched}^4_{i,\emptyset}, a_i.\mathrm{d}^4_{2,i}) \mid i \in \{1..n\}\} \\
\cup\ & \{(\Box_{j\in\emptyset} b_j.\mathrm{Sched}^4_{i+1,\emptyset-j} \Box (\textstyle\bigsqcap_{(k\notin\emptyset,k=i)} a_k.\mathrm{Sched}^4_{k+1,\{k\}} \sqcap 0), a_i.\mathrm{d}^4_{2,i}) \\
& \qquad |i \in \{1..n\}\} \\
\cup\ & \{(\textstyle\bigsqcap_{(k\notin\emptyset,k=i)} a_k.\mathrm{Sched}^4_{k+1,\{k\}} \sqcap 0, a_i.\mathrm{d}^4_{2,i}) \mid i \in \{1..n\}\} \\
\cup\ & \{(\textstyle\bigsqcap_{(k\notin\emptyset,k=i)} a_k.\mathrm{Sched}^4_{k+1,\{k\}}, a_i.\mathrm{d}^4_{2,i}) \mid i \in \{1..n\}\} \\
\cup\ & \{(a_i.\mathrm{Sched}^4_{i+1,\{i\}}, a_i.\mathrm{d}^4_{2,i}) \mid i \in \{1..n\}\} \\
\cup\ & \{(\mathrm{Sched}^4_{i+1,\{i\}}, \mathrm{d}^4_{2,i}) \mid i \in \{1..n\}\} \\
\cup\ & \{(\mathrm{Sched}^4_{i+1,\{i\}}, b_i.\mathrm{d}^4_{1,i+1}) \mid i \in \{1..n\}\} \\
\cup\ & \{(\Box_{j\in\{i\}} b_j.\mathrm{Sched}^4_{i+1,\{i\}-j} \Box (\textstyle\bigsqcap_{(k\notin\{i\},k=i)} a_k.\mathrm{Sched}^4_{k+1,\{k\}} \sqcap 0), b_i.\mathrm{d}^4_{1,i+1}) \\
& \qquad |i \in \{1..n\}\} \\
\cup\ & \{(\Box_{j\in\{i\}} b_j.\mathrm{Sched}^4_{i+1,\{i\}-j}, b_i.\mathrm{d}^4_{1,i+1}) \mid i \in \{1..n\}\} \\
\cup\ & \{(\textstyle\bigsqcap_{(k\notin\emptyset,k=i)} a_k.\mathrm{Sched}^4_{k+1,\{k\}} \sqcap 0, 0) \mid i \in \{1..n\}\} \\
\cup\ & \{(b_i.\mathrm{Sched}^4_{i+1,\emptyset}, b_i.\mathrm{d}^4_{1,i+1}) \mid i \in \{1..n\}\} \\
\cup\ & \{(0,0)\}
\end{aligned}
$$

Lastly, we justify the statement that $\mathrm{design}^5_{1,\emptyset}$ $\tilde{\mathsf{sat}}$ $\mathrm{Sched}^5_{1,\emptyset}$ given the definition:

$$
\mathrm{design}^5_{i,X} \stackrel{\mathrm{def}}{=} \begin{cases} \sum_{j\in X} b_j.\mathrm{design}^5_{i,X-j} + a_i.\mathrm{design}^5_{i+1,X\cup i} & i \notin \mathrm{X} \\ \sum_{j\in X} b_j.\mathrm{design}^5_{i,X-j} & i \in \mathrm{X} \end{cases}
$$

We use the following satisfying simulation:

$$
\begin{array}{ll}
 & \{(\mathrm{Sched}^5_{i,X}, \mathrm{design}^5_{i,X}) \mid X \subseteq \{1..n\}, i \in \{1..n\}\} \\
\cup & \{(\mathrm{Sched}^5_{i,X}, \sum_{j \in X} b_j.\mathrm{design}^5_{i,X-j} + a_i.\mathrm{design}^5_{i+1,X\cup i}) \mid X \subseteq \{1..n\}, i \notin X\} \\
\cup & \{(\mathrm{Sched}^5_{i,X}, \sum_{j \in X} b_j.\mathrm{design}^5_{i,X-j}) \mid X \subseteq \{1..n\}, i \in X\} \\
\cup & \{(\Box_{j \in X} b_j.\mathrm{Sched}^5_{i,X-j} \ \Box\ a_i.\mathrm{Sched}^5_{i+1,X\cup\{i\}}, \\
 & \quad \sum_{j \in X} b_j.\mathrm{design}^5_{i,X-j} + a_i.\mathrm{design}^5_{i+1,X\cup i}) \\
 & \qquad |X \subseteq \{1..n\}, i \notin X\} \\
\cup & \{(\Box_{j \in X} b_j.\mathrm{Sched}^5_{i,X-j}, \sum_{j \in X} b_j.\mathrm{design}^5_{i,X-j}) \mid X \subseteq \{1..n\}\} \\
\cup & \{(a_i.\mathrm{Sched}^5_{i+1,X\cup\{i\}}, a_i.\mathrm{design}^5_{i+1,X\cup i}) \mid X \subseteq \{1..n\}, i \notin X\} \\
\cup & \{(b_j.\mathrm{Sched}^5_{i,X-j}, b_j.\mathrm{design}^5_{i,X-j}) \mid X \subseteq \{1..n\}, j \in X\}
\end{array}
$$

D.3 Refining Simulations for Chapter 7

To show that $\mathrm{Dspec}^0 \sqsubseteq \mathrm{Dspec}^1_{\emptyset,\emptyset}$ we use the following simulation:

$$
\begin{array}{ll}
 & \{(\ \mathrm{Dspec}^0, \mathrm{Dspec}^1_{\mathrm{R}_1,\mathrm{R}_2}\ ) \mid \mathrm{R}_1, \mathrm{R}_2 \subseteq \mathcal{N}\} \\
\cup & \{(\ \prod_{k \in \mathrm{Key}, x \in \mathcal{N}} \xrightarrow[x]{\mathtt{o.get?}\langle k \rangle} \mathrm{Dspec}^0 \\
 & \quad \sqcap \prod_{(k,v) \in \mathrm{Key} \times \mathrm{Value}, x \in \mathcal{N}} \xrightarrow[x]{\mathtt{o.set?}\langle k,v \rangle} \mathrm{Dspec}^0 \\
 & \quad \sqcap \prod_{v \in \mathrm{Value}, x \in \mathcal{N}} \xrightarrow{x!\langle v \rangle} \mathrm{Dspec}^0 \\
 & \quad \sqcap \prod_{x \in \mathcal{N}} \xrightarrow{x!\langle\rangle} \mathrm{Dspec}^0 \\
 & \quad \sqcap \prod_{i \in \mathbb{N}} \longrightarrow \mathrm{Dspec}^0, \\
 & \quad \mathrm{Dspec}^1_{\mathrm{R}_1,\mathrm{R}_2}\ ) \mid \mathrm{R}_1, \mathrm{R}_2 \subseteq \mathcal{N}\} \\
\cup & \{(\ \prod_{k \in \mathrm{Key}, x \in \mathcal{N}} \xrightarrow[x]{\mathtt{o.get?}\langle k \rangle} \mathrm{Dspec}^0, \\
 & \quad \prod_{k \in \mathrm{Key}, x\ \mathrm{new}} \xrightarrow[x]{\mathtt{o.get?}\langle k \rangle} \mathrm{Dspec}^1_{\mathrm{R}_1 \cup \{x\},\mathrm{R}_2}\ ) \mid \mathrm{R}_1, \mathrm{R}_2 \subseteq \mathcal{N}\} \\
\cup & \{(\ \prod_{(k,v) \in \mathrm{Key} \times \mathrm{Value}, x \in \mathcal{N}} \xrightarrow[x]{\mathtt{o.set?}\langle k,v \rangle} \mathrm{Dspec}^0,
\end{array}
$$

$$
\begin{aligned}
& \qquad \bigsqcap_{(k,v)\in \mathrm{Key}\times\mathrm{Value}, x\,\mathrm{new}} \xrightarrow[x]{\mathtt{o.set?}\langle k,v\rangle} \mathrm{Dspec}^1_{\mathrm{R}_1,\mathrm{R}_2\cup\{x\}} \ ) \ \mid \mathrm{R}_1,\mathrm{R}_2 \subseteq \mathcal{N}\} \\
\cup & \ \{(\ \bigsqcap_{v\in\mathrm{Value}, x\in\mathcal{N}} \xrightarrow{x!\langle v\rangle} \mathrm{Dspec}^0, \\
& \qquad \bigsqcap_{v\in\mathrm{Value}, x\in\mathrm{R}_1} \xrightarrow{x!\langle v\rangle} \mathrm{Dspec}^1_{\mathrm{R}_1\setminus\{x\},\mathrm{R}_2} \ ) \ \mid \mathrm{R}_1,\mathrm{R}_2 \subseteq \mathcal{N}, \mathrm{R}_1 \neq \emptyset\} \\
\cup & \ \{(\ \bigsqcap_{x\in\mathcal{N}} \xrightarrow{x!\langle\rangle} \mathrm{Dspec}^0, \\
& \qquad \bigsqcap_{x\in\mathrm{R}_2} \xrightarrow{x!\langle\rangle} \mathrm{Dspec}^1_{\mathrm{R}_1,\mathrm{R}_2\setminus\{x\}} \ ) \ \mid \mathrm{R}_1,\mathrm{R}_2 \subseteq \mathcal{N}, \mathrm{R}_2 \neq \emptyset\} \\
\cup & \ \{(\ \bigsqcap_{i\in\mathbb{N}} \longrightarrow \mathrm{Dspec}^0, \bigsqcap_{i\in\mathbb{N}} \longrightarrow \mathrm{Dspec}^1_{\mathrm{R}_1,\mathrm{R}_2} \ )\} \\
\cup & \ \{(\ \xrightarrow[x]{\mathtt{o.get?}\langle k\rangle} \mathrm{Dspec}^0, \ \xrightarrow[x]{\mathtt{o.get?}\langle k\rangle} \mathrm{Dspec}^1_{\mathrm{R}_1\cup\{x\},\mathrm{R}_2} \ ) \\
& \qquad \mid \mathrm{R}_1,\mathrm{R}_2 \subseteq \mathcal{N}, k \in \mathrm{Key}, x\,\mathrm{new}\} \\
\cup & \ \{(\ \xrightarrow[x]{\mathtt{o.set?}\langle k,v\rangle} \mathrm{Dspec}^0, \ \xrightarrow[x]{\mathtt{o.set?}\langle k,v\rangle} \mathrm{Dspec}^1_{\mathrm{R}_1,\mathrm{R}_2\cup\{x\}} \ ) \\
& \qquad \mid \mathrm{R}_1,\mathrm{R}_2 \subseteq \mathcal{N}, (k,v) \in \mathrm{Key}\times\mathrm{Value}, x\,\mathrm{new}\} \\
\cup & \ \{(\ \xrightarrow{x!\langle v\rangle} \mathrm{Dspec}^0, \ \xrightarrow{x!\langle v\rangle} \mathrm{Dspec}^1_{\mathrm{R}_1\setminus\{x\},\mathrm{R}_2} \ ) \\
& \qquad \mid \mathrm{R}_1,\mathrm{R}_2 \subseteq \mathcal{N}, v \in \mathrm{Value}, x \in \mathrm{R}_1\} \\
\cup & \ \{(\ \xrightarrow{x!\langle\rangle} \mathrm{Dspec}^0, \ \xrightarrow{x!\langle\rangle} \mathrm{Dspec}^1_{\mathrm{R}_1,\mathrm{R}_2\setminus\{x\}} \ ) \ \mid \mathrm{R}_1,\mathrm{R}_2 \subseteq \mathcal{N}, x \in \mathrm{R}_2\} \\
\cup & \ \{(\ \longrightarrow \mathrm{Dspec}^0, \ \longrightarrow \mathrm{Dspec}^1_{\mathrm{R}_1,\mathrm{R}_2} \ )\}
\end{aligned}
$$

Let $\mathrm{Dstate}_0 = (\emptyset,\emptyset,\emptyset,\emptyset)$. To show that $\mathrm{Dspec}^1_{\emptyset,\emptyset} \sqsubseteq \mathrm{Dspec}^2_{\mathrm{Dstate}_0}$, we use a refining simulation which requires a non-trivial equivalence step. The two subprocesses of Dspec^2 which are meets of silent actions need to be matched with a single meet of silent actions in Dspec^1. Let $S(\mathrm{Dstate}) = \mathrm{In}_1 + \mathrm{In}_2$ and for $s \in S(\mathrm{Dstate})$ let

$$
m(\mathrm{Dstate}, s) = \begin{cases} f_5(\mathrm{Dstate}, x, k, v) & \text{if } s = (x,k) \in \mathrm{In}_1 \text{ and } v \in \mathrm{Value} \\ f_6(\mathrm{Dstate}, x, k, v) & \text{if } s = (x,k,v) \in \mathrm{In}_2 \end{cases}
$$

Clearly, we can map $S(\text{Dstate})$ to a subset of $\mathbb{N}$. Thus

$$\begin{array}{rll}
\text{Dspec}^2_{\text{Dstate}} \equiv & \prod_{k\in\text{Key},x\,\text{new}} \xrightarrow[x]{\texttt{o.get?}\langle k\rangle} & \text{Dspec}^2_{f_1(\text{Dstate},x,k)} \\
\sqcap & \prod_{(k,v)\in\text{Key}\times\text{Value},x\,\text{new}} \xrightarrow[x]{\texttt{o.set?}\langle k,v\rangle} & \text{Dspec}^2_{f_2(\text{Dstate},x,k,v)} \\
\sqcap & \prod_{(x,v)\in\text{Out}_1} \xrightarrow{x!\langle v\rangle} & \text{Dspec}^2_{f_3(\text{Dstate},x,v)} \\
\sqcap & \prod_{x\in\text{Out}_2} \xrightarrow{x!\langle\rangle} & \text{Dspec}^2_{f_4(\text{Dstate},x)} \\
\sqcap & \prod_{s\in S(\text{Dstate})} \longrightarrow & \text{Dspec}^2_{m(\text{Dstate},s)}
\end{array}$$

To relate the specifications we use the following projection

$$\pi(\text{Dstate}) = (\pi_1(\text{In}_1) \cup \pi_1(\text{Out}_1), \pi_1(\text{In}_2) \cup \text{Out}_2)$$

where π_1 projects out the first element of a tuple.

To show that $\text{Dspec}^1_{\emptyset,\emptyset} \sqsubseteq \text{Dspec}^2_{\text{Dstate}_0}$ we use the following simulation:

$$\begin{array}{ll}
 & \{(\ \text{Dspec}^1_{\pi(\text{Dstate})}, \text{Dspec}^2_{\text{Dstate}}\ )\ \mid \forall \text{Dstate}\} \\
\cup & \{(\ \prod_{k\in\text{Key},x\,\text{new}} \xrightarrow[x]{\texttt{o.get?}\langle k\rangle} \text{Dspec}^1_{\text{R}_1\cup\{x\},\text{R}_2} \\
 & \quad \sqcap \prod_{(k,v)\in\text{Key}\times\text{Value},x\,\text{new}} \xrightarrow[x]{\texttt{o.set?}\langle k,v\rangle} \text{Dspec}^1_{\text{R}_1,\text{R}_2\cup\{x\}} \\
 & \quad \sqcap \prod_{v\in\text{Value},x\in\text{R}_1} \xrightarrow{x!\langle v\rangle} \text{Dspec}^1_{\text{R}_1\setminus\{x\},\text{R}_2} \\
 & \quad \sqcap \prod_{x\in\text{R}_2} \xrightarrow{x!\langle\rangle} \text{Dspec}^1_{\text{R}_1,\text{R}_2\setminus\{x\}} \\
 & \quad \sqcap \prod_{i\in\mathbb{N}} \longrightarrow \text{Dspec}^1_{\text{R}_1,\text{R}_2}, \\
 & \quad \text{Dspec}^2_{\text{Dstate}}\ )\ \mid \forall \text{Dstate}, (\text{R}_1,\text{R}_2) = \pi(\text{Dstate})\} \\
\cup & \{(\ \prod_{k\in\text{Key},x\,\text{new}} \xrightarrow[x]{\texttt{o.get?}\langle k\rangle} \text{Dspec}^1_{\text{R}_1\cup\{x\},\text{R}_2}, \\
 & \quad \prod_{k\in\text{Key},x\,\text{new}} \xrightarrow[x]{\texttt{o.get?}\langle k\rangle} \text{Dspec}^2_{f_1(\text{Dstate},x,k)}\ ) \\
 & \qquad \mid \forall \text{Dstate}, (\text{R}_1,\text{R}_2) = \pi(\text{Dstate})\} \\
\cup & \{(\ \prod_{(k,v)\in\text{Key}\times\text{Value},x\,\text{new}} \xrightarrow[x]{\texttt{o.set?}\langle k,v\rangle} \text{Dspec}^1_{\text{R}_1,\text{R}_2\cup\{x\}},
\end{array}$$

$$
\begin{aligned}
& \quad \textstyle\prod_{(k,v)\in \mathrm{Key}\times\mathrm{Value}, x\,\mathrm{new}} \xrightarrow[x]{\texttt{o.set?}\langle k,v\rangle} \mathrm{Dspec}^2_{f_2(\mathrm{Dstate},x,k,v)} \quad) \\
& \qquad\qquad \mid \forall \mathrm{Dstate}, (\mathrm{R}_1,\mathrm{R}_2) = \pi(\mathrm{Dstate})\} \\
\cup \quad & \{(\quad \textstyle\prod_{v\in \mathrm{Value}, x\in \mathrm{R}_1} \xrightarrow{x!\langle v\rangle} \mathrm{Dspec}^1_{\mathrm{R}_1\smallsetminus\{x\},\mathrm{R}_2}, \\
& \quad \textstyle\prod_{(x,v)\in \mathrm{Out}_1} \xrightarrow{x!\langle v\rangle} \mathrm{Dspec}^2_{f_3(\mathrm{Dstate},x,v)} \quad) \\
& \qquad\qquad \mid \forall \mathrm{Dstate}, (\mathrm{R}_1,\mathrm{R}_2) = \pi(\mathrm{Dstate})\} \\
\cup \quad & \{(\quad \textstyle\prod_{x\in \mathrm{R}_2} \xrightarrow{x!\langle\rangle} \mathrm{Dspec}^1_{\mathrm{R}_1,\mathrm{R}_2\smallsetminus\{x\}}, \\
& \quad \textstyle\prod_{x\in \mathrm{Out}_2} \xrightarrow{x!\langle\rangle} \mathrm{Dspec}^2_{f_4(\mathrm{Dstate},x)} \quad) \quad \mid \forall \mathrm{Dstate}, (\mathrm{R}_1,\mathrm{R}_2) = \pi(\mathrm{Dstate})\} \\
\cup \quad & \{(\quad \textstyle\prod_{i\in \mathbb{N}} \longrightarrow \mathrm{Dspec}^1_{\mathrm{R}_1,\mathrm{R}_2}, \\
& \quad \textstyle\prod_{s\in S(\mathrm{Dstate})} \longrightarrow \mathrm{Dspec}^2_{m(\mathrm{Dstate},s)} \quad) \quad \mid \forall \mathrm{Dstate}, (\mathrm{R}_1,\mathrm{R}_2) = \pi(\mathrm{Dstate})\} \\
\cup \quad & \{(\quad \xrightarrow[x]{\texttt{o.get?}\langle k\rangle} \mathrm{Dspec}^1_{\mathrm{R}_1\cup\{x\},\mathrm{R}_2}, \xrightarrow[x]{\texttt{o.get?}\langle k\rangle} \mathrm{Dspec}^2_{f_1(\mathrm{Dstate},x,k)} \quad) \\
& \quad \mid \forall \mathrm{Dstate}, (\mathrm{R}_1,\mathrm{R}_2) = \pi(\mathrm{Dstate}), k \in \mathrm{Key}, x\,\mathrm{new}\} \\
\cup \quad & \{(\quad \xrightarrow[x]{\texttt{o.set?}\langle k,v\rangle} \mathrm{Dspec}^1_{\mathrm{R}_1,\mathrm{R}_2\cup\{x\}}, \xrightarrow[x]{\texttt{o.set?}\langle k,v\rangle} \mathrm{Dspec}^2_{f_2(\mathrm{Dstate},x,k,v)} \quad) \\
& \quad \mid \forall \mathrm{Dstate}, (\mathrm{R}_1,\mathrm{R}_2) = \pi(\mathrm{Dstate}), (k,v) \in \mathrm{Key}\times\mathrm{Value}, x\,\mathrm{new}\} \\
\cup \quad & \{(\quad \xrightarrow{x!\langle v\rangle} \mathrm{Dspec}^1_{\mathrm{R}_1\smallsetminus\{x\},\mathrm{R}_2}, \xrightarrow{x!\langle v\rangle} \mathrm{Dspec}^2_{f_3(\mathrm{Dstate},x,v)} \quad) \\
& \quad \mid \forall \mathrm{Dstate}, (\mathrm{R}_1,\mathrm{R}_2) = \pi(\mathrm{Dstate}), (x,v) \in \mathrm{Out}_1\} \\
\cup \quad & \{(\quad \xrightarrow{x!\langle\rangle} \mathrm{Dspec}^1_{\mathrm{R}_1,\mathrm{R}_2\smallsetminus\{x\}}, \\
& \quad \xrightarrow{x!\langle\rangle} \mathrm{Dspec}^2_{f_4(\mathrm{Dstate},x)} \quad) \quad \mid \forall \mathrm{Dstate}, (\mathrm{R}_1,\mathrm{R}_2) = \pi(\mathrm{Dstate}), x \in \mathrm{Out}_2\} \\
\cup \quad & \{(\quad \longrightarrow \mathrm{Dspec}^1_{\mathrm{R}_1,\mathrm{R}_2}, \\
& \quad \longrightarrow \mathrm{Dspec}^2_{m(\mathrm{Dstate},s)} \quad) \quad \mid \forall \mathrm{Dstate}, (\mathrm{R}_1,\mathrm{R}_2) = \pi(\mathrm{Dstate}), s \in S(\mathrm{Dstate})\}
\end{aligned}
$$

It is easy to show that $\mathrm{Dspec}^2_{\mathrm{Dstate}_0} \sqsubseteq \mathrm{Dspec}^3_{\mathrm{Dstate}_\perp}$. Relate the specifications with a projection which discards the Dict component of the state.

To show that $\mathrm{Dspec}^3_{\mathrm{Dstate}} \sqsubseteq \mathrm{Dspec}^4_{\mathrm{Dstate}}$ again requires a non-trivial equiv-

alence:

$$
\begin{aligned}
\mathrm{Dspec}^4_{\mathrm{Dstate}} \equiv \quad & (\; \Box_{k \in \mathrm{Key}, x\,\mathrm{new}} \xrightarrow[x]{\mathrm{o}.\mathtt{get}?\langle k \rangle} \mathrm{Dspec}^4_{f_1(\mathrm{Dstate},x,k)} \\
& \;\; \Box \Box_{(k,v) \in \mathrm{Key} \times \mathrm{Value}, x\,\mathrm{new}} \xrightarrow[x]{\mathrm{o}.\mathtt{set}?\langle k,v \rangle} \mathrm{Dspec}^4_{f_2(\mathrm{Dstate},x,k,v)}) \\
\Box \; & ((\; \bigsqcap_{(x,v) \in \mathrm{Out}_1} \xrightarrow{x!\langle v \rangle} \mathrm{Dspec}^4_{f_3(\mathrm{Dstate},x,v)} \\
& \;\; \sqcap \bigsqcap_{x \in \mathrm{Out}_2} \xrightarrow{x!\langle \rangle} \mathrm{Dspec}^4_{f_4(\mathrm{Dstate},x)} \\
& \;\; \sqcap \bigsqcap_{(x,k) \in \mathrm{In}_1, v = \mathtt{get}(\mathrm{Dict},k)} \longrightarrow \mathrm{Dspec}^4_{f_5(\mathrm{Dstate},x,k,v)} \\
& \;\; \sqcap \bigsqcap_{(x,k,v) \in \mathrm{In}_2} \longrightarrow \mathrm{Dspec}^4_{f_6(\mathrm{Dstate},x,k,v)}) \\
& \sqcap 0 \;)
\end{aligned}
$$

We use the following refining simulation

$$
\begin{aligned}
& \{(\; \mathrm{Dspec}^3_{\mathrm{Dstate}}, \mathrm{Dspec}^4_{\mathrm{Dstate}} \;) \mid \forall \mathrm{Dstate} \} \\
\cup \; & \{(\; \bigsqcap_{k \in \mathrm{Key}, x\,\mathrm{new}} \xrightarrow[x]{\mathrm{o}.\mathtt{get}?\langle k \rangle} \mathrm{Dspec}^3_{f_1(\mathrm{Dstate},x,k)} \\
& \quad \sqcap \bigsqcap_{(k,v) \in \mathrm{Key} \times \mathrm{Value}, x\,\mathrm{new}} \xrightarrow[x]{\mathrm{o}.\mathtt{set}?\langle k,v \rangle} \mathrm{Dspec}^3_{f_2(\mathrm{Dstate},x,k,v)} \\
& \quad \sqcap \bigsqcap_{(x,v) \in \mathrm{Out}_1} \xrightarrow{x!\langle v \rangle} \mathrm{Dspec}^3_{f_3(\mathrm{Dstate},x,v)} \\
& \quad \sqcap \bigsqcap_{x \in \mathrm{Out}_2} \xrightarrow{x!\langle \rangle} \mathrm{Dspec}^3_{f_4(\mathrm{Dstate},x)} \\
& \quad \sqcap \bigsqcap_{(x,k) \in \mathrm{In}_1, v = \mathtt{get}(\mathrm{Dict},k)} \longrightarrow \mathrm{Dspec}^3_{f_5(\mathrm{Dstate},x,k,v)} \\
& \quad \sqcap \bigsqcap_{(x,k,v) \in \mathrm{In}_2} \longrightarrow \mathrm{Dspec}^3_{f_6(\mathrm{Dstate},x,k,v)}, \\
& \quad \mathrm{Dspec}^4_{\mathrm{Dstate}} \;) \mid \forall \mathrm{Dstate} \} \\
\cup \; & \{(\; \bigsqcap_{k \in \mathrm{Key}, x\,\mathrm{new}} \xrightarrow[x]{\mathrm{o}.\mathtt{get}?\langle k \rangle} \mathrm{Dspec}^3_{f_1(\mathrm{Dstate},x,k)}, \\
& \quad \Box_{k \in \mathrm{Key}, x\,\mathrm{new}} \xrightarrow[x]{\mathrm{o}.\mathtt{get}?\langle k \rangle} \mathrm{Dspec}^4_{f_1(\mathrm{Dstate},x,k)} \;) \\
& \quad \mid \forall \mathrm{Dstate} \} \\
\cup \; & \{(\; \bigsqcap_{(k,v) \in \mathrm{Key} \times \mathrm{Value}, x\,\mathrm{new}} \xrightarrow[x]{\mathrm{o}.\mathtt{set}?\langle k,v \rangle} \mathrm{Dspec}^3_{f_2(\mathrm{Dstate},x,k,v)}, \\
& \quad \Box_{(k,v) \in \mathrm{Key} \times \mathrm{Value}, x\,\mathrm{new}} \xrightarrow[x]{\mathrm{o}.\mathtt{set}?\langle k,v \rangle} \mathrm{Dspec}^4_{f_2(\mathrm{Dstate},x,k,v)} \;)
\end{aligned}
$$

$$
\begin{aligned}
& \quad \mid \forall \text{Dstate}\} \\
\cup \quad & \{(\quad \textstyle\prod_{(x,v)\in \text{Out}_1} \xrightarrow{x!\langle v\rangle} \text{Dspec}^3_{f_3(\text{Dstate},x,v)}, \\
& \quad \textstyle\prod_{(x,v)\in \text{Out}_1} \xrightarrow{x!\langle v\rangle} \text{Dspec}^4_{f_3(\text{Dstate},x,v)} \quad) \\
& \quad \mid \forall \text{Dstate}\} \\
\cup \quad & \{(\quad \textstyle\prod_{x\in \text{Out}_2} \xrightarrow{x!\langle\rangle} \text{Dspec}^3_{f_4(\text{Dstate},x)}, \\
& \quad \textstyle\prod_{x\in \text{Out}_2} \xrightarrow{x!\langle\rangle} \text{Dspec}^4_{f_4(\text{Dstate},x)} \quad) \\
& \quad \mid \forall \text{Dstate}\} \\
\cup \quad & \{(\quad \textstyle\prod_{(x,k)\in \text{In}_1, v=\texttt{get}(\text{Dict},k)} \longrightarrow \text{Dspec}^3_{f_5(\text{Dstate},x,k,v)}, \\
& \quad \textstyle\prod_{(x,k)\in \text{In}_1, v=\texttt{get}(\text{Dict},k)} \longrightarrow \text{Dspec}^4_{f_5(\text{Dstate},x,k,v)} \quad) \\
& \quad \mid \forall \text{Dstate}\} \\
\cup \quad & \{(\quad \textstyle\prod_{(x,k,v)\in \text{In}_2} \longrightarrow \text{Dspec}^3_{f_6(\text{Dstate},x,k,v)}, \\
& \quad \textstyle\prod_{(x,k,v)\in \text{In}_2} \longrightarrow \text{Dspec}^4_{f_6(\text{Dstate},x,k,v)} \quad) \\
& \quad \mid \forall \text{Dstate}\} \\
\cup \quad & \{(\quad \xrightarrow[x]{\texttt{o.get}?\langle k\rangle} \text{Dspec}^3_{f_1(\text{Dstate},x,k)}, \\
& \quad \xrightarrow[x]{\texttt{o.get}?\langle k\rangle} \text{Dspec}^4_{f_1(\text{Dstate},x,k)} \quad) \\
& \quad \mid \forall \text{Dstate}, k \in \text{Key}, x \, \text{new}\} \\
\cup \quad & \{(\quad \xrightarrow[x]{\texttt{o.set}?\langle k,v\rangle} \text{Dspec}^3_{f_2(\text{Dstate},x,k,v)}, \\
& \quad \xrightarrow[x]{\texttt{o.set}?\langle k,v\rangle} \text{Dspec}^4_{f_2(\text{Dstate},x,k,v)} \quad) \\
& \quad \mid \forall \text{Dstate}, (k,v) \in \text{Key} \times \text{Value}, x \, \text{new}\} \\
\cup \quad & \{(\quad \xrightarrow{x!\langle v\rangle} \text{Dspec}^3_{f_3(\text{Dstate},x,v)}, \\
& \quad \xrightarrow{x!\langle v\rangle} \text{Dspec}^4_{f_3(\text{Dstate},x,v)} \quad) \\
& \quad \mid \forall \text{Dstate}, (x,v) \in \text{Out}_1\} \\
\cup \quad & \{(\quad \xrightarrow{x!\langle\rangle} \text{Dspec}^3_{f_4(\text{Dstate},x)},
\end{aligned}
$$

$$
\begin{aligned}
& \xrightarrow{x!\langle\rangle} \mathrm{Dspec}^4_{f_4(\mathrm{Dstate},x)} \quad) \\
& \mid \forall \mathrm{Dstate}, x \in \mathrm{Out}_2\} \\
\cup \quad \{(\quad & \longrightarrow \mathrm{Dspec}^3_{f_5(\mathrm{Dstate},x,k,v)}, \\
& \longrightarrow \mathrm{Dspec}^4_{f_5(\mathrm{Dstate},x,k,v)} \quad) \\
& \mid \forall \mathrm{Dstate}, (x,k) \in \mathrm{In}_1, v = \mathsf{get}(\mathrm{Dict}, k)\} \\
\cup \quad \{(\quad & \longrightarrow \mathrm{Dspec}^3_{f_6(\mathrm{Dstate},x,k,v)}, \\
& \longrightarrow \mathrm{Dspec}^4_{f_6(\mathrm{Dstate},x,k,v)} \quad) \\
& \mid \forall \mathrm{Dstate}, (x,k,v) \in \mathrm{In}_2\}
\end{aligned}
$$

It is relatively straightforward to show $\mathrm{Dspec}^4_{\mathrm{Dstate}} \sqsubseteq \mathrm{Dspec}^5_{\mathrm{Dstate}}$.

Bibliography

[Abr96] J.-R. Abrial. *The B-book: assigning programs to meanings.* Cambridge University Press, 1996.

[AC91] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. In *Conference Record of the 18th Annual Symposium on Principles of Programming Languages (POPL '91)*, pages 104–118. ACM Press, January 1991.

[AC96] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[AdBKR89] Pierre America, Jaco de Bakker, Joost N. Kok, and Jan Rutten. Denotational semantics of an object-oriented language. *Information and Computation*, 83(2):152–205, 1989.

[Ame87] Pierre America. Inheritance and subtyping in a parallel object-oriented language. In Jean Bezivin et al., editor, *European Conference on Object-Oriented Programming (ECOOP '87)*, volume 276 of *Lecture Notes in Computer Science*, pages 234–242. Springer-Verlag, 1987.

[Ame89] Pierre America. Issues in the design of a parallel object-oriented language. *Formal Aspects of Computing*, 1:366–411, 1989.

[Ame91] Pierre America. Designing an object-oriented programming language with behavioural subtyping. In W. P. de Roever J. de Bakker and G. Rozenberg, editors, *Foundations of Object-Oriented Languages*, volume 489 of *Lecture Notes in Computer Science*, pages 60–90. Springer-Verlag, 1991.

[Bac90] R. J. R. Back. Refinement calculus, part II: Parallel and reactive programs. In *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, volume 430 of *Lecture Notes in Computer Science*, pages 67–93. Springer-Verlag, 1990.

[Bac92] R. J. R. Back. Refinement of parallel and reactive programs. Technical Report Caltech-CS-TR-92-23, Computer Science Department, California Institute of Technology, 1992.

[Bar80] H.P. Barendregt. *The Lambda Calculus, its Syntax and Semantics*. North-Holland Publishing Company, 1980.

[BB90] Gérard Berry and Gérard Boudol. The chemical abstract machine. In Matthew Hennessy and James Riely, editors, *Conference Record of the Symposium on Principles of Programming Languages (POPL '90)*, pages 81–94. ACM Press, January 1990.

[BBS97] Ralph-Johan Back, Martin Büchi, and Emil Sekerinski. Action-based concurrency and synchronization for objects. In *Proceedings of the Fourth AMAST Workshop on Real-Time Systems, Concurrent, and Distributed Software (ARTS)*, volume 1231 of *Lecture Notes in Computer Science*, pages 248–262. Springer-Verlag, 1997.

[BCP97] Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. Comparing object encodings. In *Theoretical Aspects of Computer Software (TACS), Sendai, Japan*, September 1997.

[BD01] H. Bowman and J. Derrick, editors. *Formal Methods for Distributed Processing, A Survey of Object-oriented Approaches*. Cambridge University Press, September 2001.

[BKS97] Marcello M. Bonsangue, Joost N. Kok, and Kaisa Sere. An approach to object-orientation in action systems. Technical Report 150, Turku Centre for Computer Science, December 1997.

[BN83] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. In *Proceedings of the ACM Symposium on Operating System Principles*. ACM Press, 1983.

[BS95] Michael L. Brodie and Michael Stonebraker. *Migrating Legacy Systems: Gateways, Interfaces and the Incremental Approach*. Morgan Kaufmann Publishers Inc., 1995.

[BS99] Martin Büchi and Emil Sekerinski. Refining concurrent objects. Technical Report 298, Turku Centre for Computer Science, August 1999.

[BvW98] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus — a systematic introduction*. Springer-Verlag, 1998.

[Car94] Luca Cardelli. Obliq: A language with distributed scope. Technical Report 122, Digital Equipment Corporation, Systems Research Center, 1994.

[Car96] Luca Cardelli. Type systems. *ACM Computing Surveys*, 28(1):263–264, 1996.

[CG98] Luca Cardelli and Andrew D. Gordon. Mobile ambients. In *Proceedings of First International Conference of the Foundations of Software Science and Computation Structures (FOSSACS '98)*. Springer-Verlag, 1998.

[CJ95] Pierre Collette and Cliff B. Jones. Enhancing the tractability of rely/guarantee specifications in the development of interfering operations. Technical Report UMCS-95-10-3, Computer Science, University of Manchester, 1995.

[dB72] N. de Bruijn. Lambda calculus notations with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. In *Indagationes Mathematicae*, volume 34, pages 381–392, 1972.

[dNH84] Rocco de Nicola and Matthew Hennessy. Testing equivalence for processes. *Theoretical Computer Science*, 60:109–137, 1984.

[dNH87] Rocco de Nicola and Matthew Hennessy. CCS without τ's. In *Proceedings of the International Joint Conference on the Theory and Practice of Software Development (TAPSOFT '87)*, volume 249 of *Lecture Notes in Computer Science*, pages 138–152. Springer-Verlag, 1987.

[Dur00] J. R. Durbin. *Modern Algebra: an Introduction.* John Wiley & Sons, Inc., 4th edition, 2000.

[EE98] G. Eddon and H. Eddon. *Inside Distributed COM.* Microsoft Press, 1998.

[FFMS01] Cédric Fournet, Fabrice Le Fessant, Luc Maranget, and Alan Schmitt. *The JoCaml language beta release: Documentation and user's manual*. Institut National de Recherche en Informatique et Automatique, January 2001.

[FG02] C. Fournet and G. Gonthier. The join calculus: a language for distributed mobile programming. In *Proceedings of the Applied Semantics Summer School (APPSEM), Caminha*, volume 2395 of *Lecture Notes in Computer Science*, pages 268–332. Springer-Verlag, 2002.

[FGL+96] Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In *Proceedings of the 7th International Conference on Concurrency Theory (CONCUR '96)*, pages 406–421. Springer-Verlag, 1996.

[For00a] Formal Systems (Europe) Ltd. *FDR2 User Manual*, May 2000.

[For00b] Formal Systems (Europe) Ltd. *ProBE User Manual*, May 2000.

[FPT99] Marcelo Fiore, Gordon Plotkin, and Daniele Turi. Abstract syntax and variable binding. In *Proceedings of the 14th Annual Symposium on Logic in Computer Science (LICS '99)*, pages 193–202. IEEE Computer Society Press, 1999.

[GH98] Andrew B. Gordon and Paul D. Hankin. A concurrent object calculus: Reduction and typing. In *Proceedings of the International Workshop on High-Level Concurrent Languages (HLCL '98)*. Elsevier ENTCS, 1998.

[GP99] Murdoch Gabbay and Andrew Pitts. A new approach to abstract syntax involving binders. In G. Longo, editor, *Proceedings*

of the 14th Annual Symposium on Logic in Computer Science (LICS '99), pages 214–224, Trento, Italy, 1999. IEEE Computer Society Press.

[Hen87] Martin C. Henson. *Elements of Functional Languages.* Computer Science Texts. Blackwell Scientific Publications, 1987.

[Hoa84] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1984.

[Hon00] Kohei Honda. Elementary structures in process theory (1) sets with renaming. *Journal of Mathematical Structures in Computer Science*, 10:617–663, October 2000.

[HT91] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. *Lecture Notes in Computer Science*, 512:133–147, 1991.

[III97] Robert John Hathaway III. The object FAQ, December 1997. `http://www.cyberdyne-object-sys.com/oofaq2/index.htm`.

[Jef00] Alan Jeffrey. A distributed object calculus. In *Proceedings of the 7th International Workshop on Foundations of Object Oriented Languages (FOOL 7)*, 2000.

[Jon90] Cliff B. Jones. *Systematic Software Development Using VDM.* Prentice-Hall International, 2nd edition, 1990.

[Jon92] C. B. Jones. An object-based design method for concurrent programs. Technical Report UMCS-92-12-1, Department of Computer Science, University of Manchester, 1992.

[Jon93a] Cliff B. Jones. Constraining interference in an object-based design method. In *Proceedings of the International Joint Conference on the Theory and Practice of Software Development (TAPSOFT '93)*, volume 668 of *Lecture Notes in Computer Science*, pages 136–150. Springer-Verlag, 1993.

[Jon93b] Cliff B. Jones. A pi-calculus semantics for an object-based design notation. In *Proceedings of the 4th International Conference on Concurrency Theory (CONCUR '93)*, volume 715 of *Lecture Notes in Computer Science*, pages 158–172. Springer-Verlag, 1993.

[Jon93c] Cliff B. Jones. Reasoning about interference in an object-based design method. In J. C. P. Woodcock and P. G. Larsen, editors, *Proceedings of 1st International Symposium of Formal Methods Europe (FME '93)*, volume 670 of *Lecture Notes in Computer Science*, pages 1–17. Springer-Verlag, 1993.

[Jon94] Cliff B. Jones. Process algebra arguments about an object-based design notation. In *A Classical Mind*, pages 231–245. Prentice-Hall, 1994.

[Jon03] Simpon Payton Jones. *Haskell 98: Language and Libraries, the Revised Report*. Cambridge University Press, April 2003.

[Jon81] C. B. Jones. *Development Methods for Computer Programmes Including a Notion of Interference*. PhD thesis, Oxford University, Computing Laboratory, UK, June l981.

[KS98] Josva Kleist and Davide Sangiorgi. Imperative objects and mobile processes. In D. Gries and W.-P. de Roever, editors, *IFIP*

Working conference on Programming Concepts and Methods, pages 285–303. Chapman and Hall, 1998.

[LR88] L.B.Wilson and R.G.Clark. *Comparative Programming Languages*. Addison Wesley, 1988.

[LSV99] Luís Lopes, Fernando Silva, and Vasco T. Vasconcelos. *TyCO User's Manual, Version 0.2*. University of Lisbon, April 1999.

[Mey97] Bertrand Meyer. *Object-oriented software construction*. Prentice-Hall, Inc., 2nd edition, 1997.

[MH99] V. Matena and M. Hapner. *Enterprise Java Beans Specification, version 1.1*. Sun Microsystems, December 1999.

[Mil80] Robin Milner. *A Calculus for Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1980.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall International Series in Computer Science. Prentice Hall, 1989.

[Mil91] Robin Milner. The polyadic π-calculus: a tutorial. LFCS Report Series ECS-LFCS-91-180, University of Edinburgh, 1991.

[Mil99] Robin Milner. *Communicating and Mobile Systems: the π-calculus*. Cambridge University Press, 1999.

[MPW89a] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, Part I. LFCS Report Series ECS-LFCS-89-85, University of Edinburgh, June 1989.

[MPW89b] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, Part II. LFCS Report Series ECS-LFCS-89-86, University of Edinburgh, June 1989.

[Nie92] Oscar Nierstrasz. Towards an object calculus. In Mario Tokoro, Oscar Nierstrasz, and Peter Wegner, editors, *Proceedings of the ECOOP'91 Workshop on Object-Based Concurrent Computing*, volume 612, pages 1–20. Springer-Verlag, 1992.

[Nie93] Oscar Nierstrasz. Composing active objects. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Object-Based Concurrency*, pages 151–171. MIT Press, 1993.

[Ode94] Martin Odersky. A functional theory of local names. In *Proceedings of 21st Annual Symposium on Principles of Programming Languages (POPL '94)*, pages 48–59. ACM Press, 1994.

[OMG91] OMG. *CORBA 1.0 Specification*, October 1991.

[OMG98] OMG. *CORBA 2.2 Specification*, February 1998.

[OMG01] OMG. *OMG Unified Modelling Language Specification*, September 2001.

[Öve00] Gunnar Övergaard. *Formal Specification of Object-Oriented Modelling Concepts*. PhD thesis, Department of Teleinformatics, Royal Institute of Technology, Stockholm, November 2000.

[PG00] Andrew M. Pitts and Murdoch Gabbay. A metalanguage for programming with bound names modulo renaming. In *Mathematics of Program Construction*, pages 230–255, 2000.

[Pie98] Benjamin C. Pierce. *Programming in the Pi-calculus*, March 1998. `http://citeseer.nj.nec.com/pierce97programming.html`.

[Pit01] Andrew M. Pitts. Nominal logic, a first order theory of names and binding. In *4th International Symposium on Theoretical Aspects of Computer Software (TACS '01)*, volume 2215 of *Lecture Notes in Computer Science*, pages 219–242. Springer-Verlag, October 2001.

[Plo81] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, 1981.

[PS93a] Benjamin Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. In *Proceedings of the 8th Annual IEEE Symposium on Logic in Computer Science (LICS '93)*, pages 376–385. IEEE Computer Society Press, June 1993.

[PS93b] Andrew M. Pitts and Ian D. B. Stark. Observable properties of higher order functions that create local names, or: What's new? In *Proceedings of 18th International Symposium on Mathematical Foundations of Computer Science*, volume 711 of *Lecture Notes in Computer Science*, pages 122–141. Springer-Verlag, 1993.

[PS95] Joachim Parrow and Davide Sangiorgi. Algebraic theories for name-passing calculi. *Journal of Information and Computation*, 120(2):174–197, 1995.

[PT94] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, 1994.

[PT95] Benjamin C. Pierce and David N. Turner. Concurrent objects in a process calculus. In Takayasu Ito and Akinori Yonezawa, editors, *Proceedings Theory and Practice of Parallel Programming (TPPP 94)*, pages 187–215, Sendai, Japan, 1995. Springer LNCS 907.

[PT98] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. CSCI Technical Report 476, Indiana University, March 1998.

[RH98] James Riely and Matthew Hennessy. A typed language for distributed mobile processes. In *Conference Record of the 25th Symposium on Principles of Programming Languages (POPL '98)*, pages 378–390, New York, NY, 1998.

[Ros98] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1998.

[San92] Davide Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*. PhD thesis, Department of Computer Science, Edinburgh University, 1992.

[San96] Davide Sangiorgi. An interpretation of typed objects into typed π-calculus. Technical Report RR-3000, Inria, Institut National de Recherche en Informatique et en Automatique, 1996.

[Sch94] Fred B. Schneider. A role for formal methodists. In *Fourth International Workshop on Dependable Computing for Critical Applications*, pages 29–30, January 1994.

[SL96] Jean-Guy Schneider and Markus Lumpe. Modelling objects in pict. Technical Report IAM-96-004, Institute for Computer Science and Applied Mathematics, University of Berne, January 1996.

[SL00] Jean-Guy Schneider and Markus Lumpe. A metamodel for concurrent, object-based programming. In Christophe Dony and Houari A. Sahraoui, editors, *Proceedings of Languages et Modèles à Objets '00*, pages 149–165, Mont Saint-Hilaire, Québec, January 2000.

[Smi99] Graeme Smith. *The Object-Z Specification Language*. Advances in Formal Methods. Kluwer Academic Publishers, December 1999.

[Smi01] Graeme Smith. State-based approaches: from Z to object-Z. In H. Bowman and J. Derrick, editors, *Formal Methods for Distributed Processing: a Survey of Object-Oriented Approaches*, pages 105–125. Cambridge University Press, 2001.

[Spi92] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 2nd edition, 1992.

[Str91] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1991.

[SW01] Davide Sangiorgi and David Walker. *The π-Calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.

[SWM96] George Shepherd, Scott Wingo, and Dean D. McCrory. *MFC Internals: Inside the Microsoft Foundation Class Architecture*. Addison-Wesley, May 1996.

[TB02] Malcolm Tyrrell and Andrew Butterfield. Typing and subtyping for an object-oriented process algebra. Technical Report TCD-CS-2002-08, Department of Computer Science, Trinity College, Dublin, February 2002.

[THS96] J. Threet, J. Hale, and S. Shenoi. A process calculus for distributed objects, 1996. Submitted to 15th Symposium on Principles of Distributed Computing (PODC '96).

[Vas94a] V. T. Vasconcelos. Typed concurrent objects. In *Proceedings of 8th European Conference of Object-Oriented Programming (ECOOP '94)*, volume 821 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[Vas94b] Vasco T. Vasconcelos. Recursive Types in a Calculus of Objects. *Transactions of the Information Processing Society of Japan*, 35(9):1828–1836, September 1994.

[Vas01] Vasco T. Vasconcelos. *TyCO Gently*. University of Lisbon, 2001.

[VB98] Vasco T. Vasconcelos and Rui Bastos. Core-TyCO: The language definition, version 0.1. Technical Report TR-98-3, Departamento de Informática, Faculdade de Ciências da Univeridade de Lisboa, March 1998.

[vdL89] Rick F. van der Lans. *The SQL standard: a complete reference*. Academic Service, Schoonhoven, The Netherlands, 1989.

[vG97] R.J. van Glabbeek. Notes on the methodology of CCS and CSP. *Theoretical Computer Science*, 177:329–349, 1997.

[Wal95] David Walker. Objects in the pi-calculus. *Information and Computation*, 116:253–271, 1995.

[WF91] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. Technical Report COMP TR91-160, Department of Computer Science, Rice University, April 1991.

[WRW96] A. Wollrath, R. Riggs, and J. Waldo. A distributed object model for the Java system. In *2nd Conference on Object-Oriented Technologies & Systems (COOTS)*, pages 219–232. USENIX Association, 1996.

[XdRH97] Qiwen Xu, Willem-Paul de Roever, and Jifeng Hi. The rely-guarantee method for verifying shared variable concurrent programs. *Formal Aspects of Computing*, 9(2):149–174, 1997.