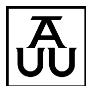
UPPSALA
UNIVERSITET

Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 1392

Languages, Logics, Types and Tools for Concurrent System Modelling

RAMŪNAS GUTKOVAS



ACTA
UNIVERSITATIS
UPSALIENSIS
UPPSALA
2016

ISSN 1651-6214
ISBN 978-91-554-9628-9
urn:nbn:se:uu:diva-300029

Dissertation presented at Uppsala University to be publicly examined in ITC/2446, Lägerhyddsvägen 2, Uppsala, Friday, 9 September 2016 at 10:15 for the degree of Doctor of Philosophy. The examination will be conducted in English. Faculty examiner: Associate Professor Thomas T. Hildebrandt (IT University of Copenhagen).

Abstract

Gutkovas, R. 2016. Languages, Logics, Types and Tools for Concurrent System Modelling. *Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology* 1392. 60 pp. Uppsala: Acta Universitatis Upsaliensis. ISBN 978-91-554-9628-9.

A concurrent system is a computer system with components that run in parallel and interact with each other. Such systems are ubiquitous and are notably responsible for supporting the infrastructure for transport, commerce and entertainment. They are very difficult to design and implement correctly: many different modeling languages and verification techniques have been devised to reason about them and verifying their correctness. However, existing languages and techniques can only express a limited range of systems and properties.

In this dissertation, we address some of the shortcomings of established models and theories in four ways: by introducing a general modal logic, extending a modelling language with types and a more general operation, providing an automated tool support, and adapting an established behavioural type theory to specify and verify systems with unreliable communication.

A modal logic for transition systems is a way of specifying properties of concurrent system abstractly. We have developed a modal logic for nominal transition systems. Such systems are common and include the pi-calculus and psi-calculi. The logic is adequate for many process calculi with regard to their behavioural equivalence even for those that no logic has been considered, for example, CCS, the pi-calculus, psi-calculi, the spi-calculus, and the fusion calculus.

The psi-calculi framework is a parametric process calculi framework that subsumes many existing process calculi. We extend psi-calculi with a type system, called sorts, and a more general notion of pattern matching in an input process. This gives additional expressive power allowing us to capture directly even more process calculi than was previously possible. We have reestablished the main results of psi-calculi to show that the extensions are consistent.

We have developed a tool that is based on the psi-calculi, called the psi-calculi workbench. It provides automation for executing the psi-calculi processes and generating a witness for a behavioural equivalence between processes. The tool can be used both as a library and as an interactive application.

Lastly, we developed a process calculus for unreliable broadcast systems and equipped it with a binary session type system. The process calculus captures the operations of scatter and gather in wireless sensor and ad-hoc networks. The type system enjoys the usual property of subject reduction, meaning that well-typed processes reduce to well-typed processes. To cope with unreliability, we also introduce a notion of process recovery that does not involve communication. This is the first session type system for a model with unreliable communication.

Keywords: process calculus, modal logic, session types, tool

Ramūnas Gutkovas, Department of Information Technology, Box 337, Uppsala University, SE-75105 Uppsala, Sweden.

© Ramūnas Gutkovas 2016

ISSN 1651-6214
ISBN 978-91-554-9628-9
urn:nbn:se:uu:diva-300029 (http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-300029)

Dedicated to fellow doctoral students.

List of papers

This thesis is based on the following papers, which are referred to in the text by their Roman numerals.

I Joachim Parrow, Johannes Borgström, Lars-Henrik Eriksson, Ramūnas Gutkovas, Tjark Weber. *Modal Logics for Nominal Transition Systems.* Draft prepared for a journal submission.

Based on a version that appeared in the proceedings of 26th International Conference on Concurrency Theory (CONCUR), 2015.

II Johannes Borgström, Ramūnas Gutkovas, Joachim Parrow, Björn Victor, Johannes Åman Pohjola. *A sorted semantic framework for applied process calculi.* Logical Methods in Computer Science (LMCS), Volume 12, Number 1, 2016

Based on an *extended abstract* version that appeared in the proceedings of 8th International Symposium on Trustworthy Global Computing (TGC), 2013.

III Ramūnas Gutkovas, Dimitrios Kouzapas, Simon J. Gay. *A Session Type System for Unreliable Broadcast Communication.* Draft.

A further development of *Session Types for Broadcasting* that was presented at 7th Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software (PLACES), 2014.

IV Johannes Borgström, Ramūnas Gutkovas, Iona Rodhe, Björn Victor. *The Psi-Calculi Workbench: A Generic Tool for Applied Process Calculi.* ACM Transactions on Embedded Computing, ACSD Special Issue, 2015.

Based on *A Parametric Tool for Applied Process Calculi*, a version that appeared in the proceedings of 13th International Conference on Application of Concurrency to System Design (ACSD), 2013.

Reprints were made with permission from the publishers.

Other Papers not Included

V Volkan Cambazoglu, Ramūnas Gutkovas, Johannes Åman Pohjola, and Björn Victor. *Modelling and Analysing a WSN Secure Aggregation Protocol: A Comparison of Languages and Tool Support*. Technical report 2015-033, Department of Information Technology, Uppsala University.

Synopsis
We modelled the Secure in-network aggregation protocol for wireless sensor networks in PWB, mCRL2 and Proverif. We successfully verified a security property in a small instance of the protocol in PWB. We compared the convenience and applicability of the tools for this specific problem. The work resulted in enchantments to PWB.

VI Dimitrios Kouzapas, Ramūnas Gutkovas, Simon J. Gay. *Session Types for Broadcasting*. Presented at 7th Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software (PLACES), 2014.

Synopsis
This is a precursor of Paper III, where we used an encoding to broadcast psi-calculi to provide semantics for a similar calculus as in Paper III, however, this resulted in a somewhat complicated system. We notably introduced a reduction semantics for broadcast psi-calculi.

VII Ramūnas Gutkovas. *Symbolic Semantics for Nominal SOS*. Draft, 2015.

Synopsis
We introduce a rule format for nominal structural operational semantics. We derive concrete and symbolic transitions system from the rules and interpret them in nominal permissive logic. We show that the symbolic transition system is complete with regard to the concrete transition system.

Acknowledgements

First and foremost, I am grateful to my advisors Johannes Borgström and Björn Victor for giving me the opportunity to do this work and the support I received from them over the years. I would like to thank the Mobility group's current and former members: Joachim Parrow, Johannes Åman Pohjola, Volkan Cambazoglu, Tjark Weber, Sofia Cassel, Sophia Knight, Palle Raabjerg, Iona Rodhe, Lars-Henrik Eriksson, Jan Kudlicka, Magnus Johansson, and Jesper Bengtson.

I would also like to thank my Glasgow colleagues Dimitrios Kouzapas and Simon J. Gay for many fruitful discussions and hospitality. A word of thanks goes to the COST BETTY project for funding my trips to Glasgow, and organising excellent summer schools.

The ProFUN project funded my work on Paper VI, Paper V, and partly Paper IV, for which I am grateful.

Paul Subotic, thanks for all the coffee breaks that we had discussing many things and somehow always ended discussing some kind of research. Thanks Stephan Brandauer for the logics club and many discussion that we had about linear types.

I would like to thank my family, and especially Mia for all her support throughout this long journey.

Sammanfattning på Svenska

Datorsystem är en stor del av vårt dagliga liv. De stöder infrastruktur för både transport, handel och underhållning. Vi förlitar oss exempelvis på distribuerade datorsystem för att bearbeta kreditkortsbetalningar på ett tillförlitligt och säkert sätt. Vi litar på att flygplanets datorsystem för autopiloten fungerar som avsett vid alla tillfällen. Vi förväntar oss att det distribuerade mobiltelefonnätet ska hantera telefonsamtal eller dataöverföring medan vi rör oss från ett område till ett annat. Vi förväntar oss också att den moderna datorns processor korrekt kan beräkna resultat genom att skifta vår data mellan sina många multi-core processorer.

Det vanligaste sättet att fastställa att sådana system fungerar korrekt, dvs som det är tänkt, är testning. Det vill säga, genom att helt enkelt förse ett system med test-indata, och sedan kontrollera att responsen är det förväntade resultatet, då utför vi testning. Emellertid har testning sina begränsningar. För komplexa system är det inte görbart att räkna upp alla möjliga inputs och kontrollera om systemets svar är korrekt. Att exempelvis ringa alla möjliga nummer i ett mobilnät samtidigt som du besöker varje möjlig plats för att kontrollera att rätt mottagare tar emot samtalet, är helt enkelt inte genomförbart. Således vad gäller testning, kan vi aldrig hoppas på att visa en komplett korrekthet för ett system som används i praktiken. En av de datavetenskapliga pionjärerna, E. W. Dijkstra, uttryckte det vältaligt i sin essä om strukturerad programmering: "testning av program kan användas för att påvisa förekomsten av buggar, men aldrig deras frånvaro!"

Ett alternativt, men kompletterande, tillvägagångssätt är att se ett datorsystem som en matematisk modell, en modell som man kan resonera kring med hjälp av matematiska verktyg. Modellerna ger oss möjlighet att få en mer exakt förståelse och försäkran om korrekthet. En modell är ett abstrakt beteende av ett system med den exakta innebörden av dess funktion. Till exempel kan vi överväga en modell av ett nätverkssystem med kapacitet för in- och utmatning av data, över en gemensam kommunikationskanal vars komponenter kommunicerar parallellt som vi abstraherar från, genom att bortse från transportmedlen för dessa meddelanden över ett nätverk.

Vi kan ha modeller på olika abstraktionsnivåer: från ganska enkla modeller som är lätta att förstå och bedöma korrekthet, till modeller som ligger nära hur själva systemet realiseras. Genom att ha dessa formella modeller av system på olika abstraktionsnivåer, kan vi utforska olika aspekter av ett system och även relatera dem på rigorösa sätt. En av önskningarna är att hitta en motsvarighet mellan de modeller som identifierar matchande komplext beteende med abstrakt beteende samtidigt som korrekthetsegenskaper behålls. Till exempel, i

den abstrakta modellen kan vi ha ett uttalande om att skicka ett meddelande, medan meddelandet i en konkret modell kan kopieras till en buffert som sedan hanteras av en delprocess som använder internetprotokoll för att överföra den via internet. Det väsentliga här är att ett meddelande skickas. En lämplig likvärdighet skulle relatera dessa modeller. Således handlar tillvägagångssättet om att hantera komplexitet. Vanligtvis finns det två modeller: en abstrakt och en konkret, som kallas specifikation och genomförande, respektive. Eftersom den abstrakta modellen oftast har mindre beteende och komplexitet, är det mer hanterbart att upprätta korrekthetsegenskaper, och för detta kan många tekniker användas. Problemet att fastställa korrektheten i genomförandet blir sedan att hitta en lämplig likvärdighet mellan specifikation och genomförande.

Genom att ha formella modeller, kan vi också resonera om system med större självförtroende och specificera intuitiva egenskaper som ska hålla över system. Vi kan kortfattat säga, till exempel, vad det innebär för ett system att hamna i dödläge, och vilka villkor ett system bör ha för att detta aldrig ska inträffa. Vi kan konstatera och hitta exakt om ett system är säkert, det vill säga om det finns möjlighet att systemet någonsin når ett tillstånd som är felaktig. Vi kan också konstatera om ett system har en livlig egenskap, det vill säga huruvida någonting måste hända i systemet. En abstrakt modell är värdefull i sin egen rätt oavsett om det är formellt verifierat att matcha genomförandet eller inte. Exempelvis kunde den ledande leverantören av cloud computing, Amazon, utveckla och implementera aggressiva prestandaoptimeringar i sina system där självförtroende vunnits genom att ha en formell modell, som inte har en formell korrespondens till genomförandet.

Studiet av grunden till dessa system är också viktigt. Genom att ha en solid grund kan vi förstå hur kraftfulla systemen är, det vill säga vad vi kan göra med dem; vi kan också avgöra vilken typ av korrekthetsegenskaper som vi någonsin kommer förmå visa.

Det finns en uppsjö av modelleringsspråk och teorier om system. Befintliga språk och tekniker kan dock bara uttrycka en begränsad krets av system eller egenskaper, och saknar automatiskt resonemang.

I denna avhandling tar vi itu med några av nackdelarna med modellsystem genom att utvidga uttrycksfullheten i etablerade modelleringsspråk, utveckla verktyg för automatisering, undersöka modallogik för övergångssystem, och anpassa ett väletablerat session typ system för att hantera opålitlig kommunikation. Vi undersöker i första hand samtidigt kommunicerande system. I sådana system kommunicerar processer genom meddelandeöverföring (skickar meddelanden) parallellt. Ett exempel som vi redan nämnt: den tidigare nämnda mobiltelefonnätet, multi-core processorer, nätverksdatorsystem, och många andra.

Vi förlänger ramverket för psi-calculi, ett modelleringsspråk för samtidiga system, med en mer kraftfull funktion för inmatning, och utrustar psi-calculi med ett typsystem. Vi utvecklar ett verktyg för psi-calculi som beräknar körningar av en psi-calculi process, och genererar beteendemässiga överens-

stämmelser. Vi har infört det binära session typ systemet för system med opålitlig kommunikation, och slutligen har vi undersökt logik för godtyckliga övergångssystem.

Contents

1 Introduction ... 15
1.1 Thesis Organisation ... 20

2 Background on Concurrent System Modelling ... 21
2.1 Process Calculi ... 21
2.1.1 Pure Process Calculi ... 23
2.1.2 Applied Process Calculi ... 32
2.2 Formal Semantics ... 37
2.2.1 Structural Operational Semantics ... 37
2.2.2 Reduction Semantics ... 40
2.3 A Behavioural Equivalence: Bisimilarity ... 41
2.4 Logic for Transition Systems ... 44
2.5 Session Types ... 46

3 Summary of Contributions ... 50
3.1 Paper I: Modal Logics for Nominal Transition Systems ... 50
3.1.1 Comments on My Participation ... 51
3.2 Paper II: A Sorted Semantic Framework for Applied Process Calculi ... 51
3.2.1 Comments on My Participation ... 53
3.3 Paper III: A Session Type System for Unreliable Broadcast Communication ... 53
3.3.1 Comments on My Participation ... 54
3.4 Paper IV: The Psi-Calculi Workbench: A Generic Tool for Applied Process Calculi ... 54
3.4.1 Comments on My Participation ... 55

4 Conclusion and Future Work ... 56

References ... 59

1. Introduction

Computer systems are a prominent fixture in our daily lives. Computer systems support infrastructure for transport, commerce and entertainment. We rely on distributed computing systems to process credit card payments reliably and securely. We trust that the aircraft's onboard autopilot computer system works as intended at all possible times. We expect the distributed mobile phone network to seamlessly carry phone calls or data transfer while we are roaming about. We also expect that the modern computer processor to correctly compute results by shifting our data between its multiple core processing units.

The most common way of determining that such systems work correctly, i.e. as intended, is testing. That is, we perform testing by simply probing a system with test input data, and then checking that the response is the expected result. However, testing has its limitations. For complex systems, it is not possible to list all possible inputs and check whether the system response was correct. For example, dialling every possible number in a mobile network while visiting every possible location and checking that the right target receives the call is simply infeasible. Thus with testing, we can never hope to show the complete correctness of a real-world system. One of the computer science pioneers E. W. Dijkstra put it eloquently in his essay on structured programming[1] "program testing can be used to show the presence of bugs, but never to show their absence!"

An alternative, but supplementary, approach is when one sees a computer system as a mathematical model that can be used to reason about with mathematical tools. The models allow us to gain more precise understanding and assurance of correctness. A model is an abstraction of a behaviour of a system with a precise meaning of its operations. For example, we may consider a model of a networked system to have capabilities of inputting and outputting data over a shared communication channel, and parallel interacting components, where we abstract from, by disregarding, the means of transporting those messages over a network.

We can have models at different abstractions levels: starting from quite simple models that are more manageable to understand and assess for correctness, to models that are close to how the actual system is realised. By having formal models of systems at various abstractions, we can explore different aspects of a system and also relate them in rigorous ways. One of the desires is to find

[1] The note EWD-268.

an equivalence between the models that equates matching complex behaviour with abstract behaviour while preserving correctness properties. For instance, in the abstract model, we may have a statement of sending a message, while in a concrete model the message may be copied to a buffer that is then handled by a subprocess which uses the internet protocol to transfer it over the internet. The essential is that a message is sent. An appropriate equivalence would relate these models. Thus, the approach is about managing complexity. Typically, there are two models: an abstract and concrete that are referred to as specification and implementation, respectively. Since the abstract model usually has less behaviour and complexity, establishing the correctness property is more manageable for which many techniques can be used. Then, the problem of establishing the correctness of implementation becomes finding an appropriate equivalence between specification and implementation.

By having formal models, we can also reason about systems with more confidence and pin down intuitive properties that should hold across systems. We can concisely state, for example, what it means for a system to deadlock, and what conditions should hold for a system for this to never occur. We can state and find precisely whether a system is safe, that is, whether nothing bad will happen. We can also state whether a system has a liveness property, that is, whether something must happen in the system. An abstract model is valuable in its own right whether it is formally verified to match implementation or not. For example, the leading cloud computing provider Amazon was able to engineer and implement aggressive performance optimisations in their systems with confidence gained by having a formal model [24], which does not have a formal correspondence to the implementation.

The study of foundations for these models is also important. By having solid foundations, we can understand how powerful the models are, that is, what we can do with them; we can also determine what kind of correctness properties we can ever be able to show.

There are a plethora of modelling languages and theories of systems. However, existing languages and techniques can only express a limited class of systems or properties, and lack automated reasoning.

In this thesis, we address some of the disadvantages of modelling systems by extending the expressiveness of established modelling languages, developing tools for automation, investigating modal logics for transition systems, and adapting a well-established session type system to cope with unreliable communication. We primarily investigate modelling languages for concurrent communicating systems. In such systems, processes communicate by using message passing (sending messages) concurrently. Example of which we already mentioned: the aforementioned mobile phone network, multi-core processors, networked computer system, and many others.

We extend the psi-calculi framework, a modelling language for concurrent systems, with a more powerful operation for input, and equip psi-calculi with a type system. We develop a tool for psi-calculi that computes executions of

a psi-calculi process, and generates behavioural equivalences. We have introduced a binary session type system for a system with unreliable communication, and finally we have investigated logics for arbitrary transition systems. We give a brief introduction to each in the following.

Psi-calculi

One of the established modelling languages and techniques for modelling concurrent communicating systems is by using a process calculus. A process calculus is a formal language where as opposed to a programming language the number of operations is kept to a minimum while still retaining full expressive power. A process calculus is typically equipped with a notion of a transition system that describes the means that a process evolves from one state to another, denoting a behaviour of a process, and a behavioural equivalence with algebraic properties. In a process calculus for communicating systems, one finds operations for sending and receiving data and a notion of concurrency. There have been many process calculi introduced to model various phenomenon found in concurrent system programming like the location of processes, security related primitives, point-to-point communication, broadcast communication and many others. Psi-calculi is a process calculus framework that provides a single theory to unify those many process calculi, and to reduce the effort of developing new process calculi as a psi-calculus. Many desirable properties, standard for process calculi, hold for every single instantiation of this framework provided certain requirements are met.

Types for psi-calculi

A type system is a means of assigning a type to various constructs of programming language. A type system ensures that the program composes according to the type system rules, making the runtime behaviour of a programming language safe with respect to the type system. Programs that satisfy the rules of the type system are called well-typed. For example, in many typed languages there is no type assignment for the expression `5 + "foobar"` that applies addition to `5` of type integer and `"foobar"` of type string. Thus, the type system rules out programs with runtime errors by making the programs type safe. A commonly desired property of type systems is that the well-typed programs evolve to well-typed programs. This feature is called the subject reduction property. For example, `5 + 3` has type integer, and program evolving from this, by computing, to the value `8` has also type integer. Most of the mainstream languages make use of type systems of varying degree of power, like the C programming language, C++, and Java.

The original psi-calculi theory is untyped [5]. In a sense, every construct and operation in the psi-calculi is deemed safe. In other words, all the channels and data have the same single type. This makes using psi-calculi theory somewhat complicated. One then needs to resort to expressing the data invari-

ants, such as the one mentioned in the above paragraph, operationally, that is, by treating malformed data specially in the model.

We extend psi-calculi with a simple type system, called sorts. We sort the channels and data terms, that is, names, terms, patterns and variables. We force the substitution function to respect the sorts. This allows us to capture even more process calculi much closer than it was possible with the original psi-calculi, such as value passing CCS, polyadic synchronisation pi-calculus, and polyadic pi-calculus. This, gives assurance that the theory is general enough and would be useful for defining new process calculi. We, of course, show that sorted psi-calculi (with pattern matching) has the subject reduction property.

Pattern matching in psi-calculi

In this thesis, we have extended psi-calculi with a more general operation for data input. The input operator now can use pattern matching for matching data that not only matches on the structure of the data but also can use arbitrary computation to determine the matches. This extension allows us to capture as instances of psi-calculi even more process calculi and more faithfully. The extension also introduces non-deterministic behaviour on input, which is useful for some applications. Most significantly it allows for straightforward modelling of security primitives, which are essential for many distributed computing systems, such as many of the protocols used on the Internet.

A tool for psi-calculi

We have implemented a tool for automated reasoning using the psi-calculi framework including the sort system. The tool is called the psi-calculi workbench, or PWB for short. The tool provides an interactive interface to execute and inspect all the possible state transitions that a psi-calculi process can make. We also implement an automatic behavioural equivalence generation. That is, the tool checks whether two psi-calculi processes are behaviourally equivalent by generating a witness relation[2]. We implement two versions of both the execution of transitions and equivalence generation: (1) the *strong* version where transitions are followed exactly; and (2) a *weak* version where the transitions corresponding to internal actions are ignored.

The language of psi-calculi that the tool implements is slightly restricted from the full psi-calculi; however, on top of the standard point-to-point semantics, the tool implements the unreliable broadcasting communication of broadcast psi-calculi [7].

The tool is parametric in the same way as psi-calculi. That is, it is possible to implement various other calculi by using the same code-base just by instantiating the provided API. To make execution of transitions computable, we formalised and implemented what is called symbolic semantics. That is, the tool generates execution of transitions with a logical condition that have to

[2]A bisimulation relation.

be satisfied for the transition to be valid. For solving the satisfaction problem of those conditions, one can interface PWB with an external constraint solver such as SMT solver.

Session types for unreliable broadcast communication

Standard type systems assign types to constructs and values of computation. Session types, a kind of behavioural types, instead describe the steps of computation that are taken to produce a result, i.e., its behaviour. These kind of types guarantee even stronger type safety properties. For example, that there are no communication mismatch errors where processes deadlock because both of them are expecting a value to be sent from each other. Session types are also an abstract specification of a protocol where the type-checking relates an implementation to its specification.

When modelling systems at a low abstraction level, the communication is unreliable, that is, the messages that are sent in the system may be lost. Ethernet is an example of such a system. Thus far, session type systems were formalised for reliable systems only. However, unreliable systems also feature structured communication that is amenable to abstract specification using session types.

Ad-hoc and wireless sensor networks on top of unreliable communication also have broadcast communication, where nodes send messages to their neighbouring nodes all at once.

We introduce a process calculus for systems with unreliable broadcast. We identify two common operations in those systems, namely, scatter and gather. Scatter corresponds to simply broadcasting a message, while gather aggregates messages received from multiple neighbouring nodes. The processes are capable of sending and receiving, as well as initiating a session with nodes by unreliable broadcast. The process calculus also contains a notion of process location and a connectivity graph on locations that affects communication range. To cope with unreliability, we have introduced recovery processes. A process may non-deterministically recover at any time and drop all current sessions. We have used the standard (binary) session types for our calculus. Our type system enjoys the subject reduction property.

Modal Logics

Another way of expressing and checking properties like safety and liveness is a modal logic. Usually, the modal formulas describe the necessity and possibility of a system to do a certain action in a certain state, that is, the logic acts as an observer of a system's behaviour. The other formulas typically are that of standard propositional or predicate logic, which includes predicates that hold only in certain states of a system, and logical conjunction, disjunction, and negation.

Modal logic formulas thus can be seen as abstract specifications of a system. With a powerful enough logic we can express properties of a system such as

invariants that hold for every state of a system (also known as safety property), and properties that a system must satisfy eventually (also known as liveness property). Modal logics such as CTL and LTL have been successfully applied in computer system verification with push-button automatic verification know as model checking.

In process calculi and many other models of concurrent systems, the notion of a name plays an important role. Names are used to represent channels, variables, and binding occurrences. In such nominal systems, the names are also emitted as part of the observable behaviour, actions, that typically include an extrusion of the scope of a name. We introduce the notion of a nominal transition system that captures the name binding in the actions, and an infinitary modal logic with modality formulas capable of observing such actions. We formalise the labelled bisimulation for the nominal transition systems and show that the logic is not capable of differentiating between behaviourally equivalent systems, and that logically equivalent processes are behaviourally equivalent. Thus, our logic is adequate with regard to the behavioural equivalence. Our logic is more powerful than previous logics for nominal transitions systems, as formulas that quantify over names are expressible in our logic. Many standard formulas like universal quantification, and recursion (least and greatest fixpoint) are easily expressible in our logic too. Our logic subsumes many existing logics for nominal transition systems, and their adequacy results follow from our adequacy results.

1.1 Thesis Organisation

This dissertation is a comprehensive summary: it is split into two major parts. In the first part, we cover the background in Chapter 2 and contributions in Chapter 3. In the second part, we include the copies of papers that constitute the thesis.

Chapter 2 gives some background to the reader on modelling languages and techniques that we use in the papers. In Section 2.1, we gradually and informally introduce CCS, the pi-calculus, psi-calculi process calculi.

In Section 2.2, we introduce prominent techniques of giving meaning to process calculi: structural operational semantics and reduction semantics. In Section 2.3, we provide a description of bisimilarity. In Section 2.4, we give a description of modal logics for transition systems. Finally, we end the background section 2.5 on binary session types.

2. Background on Concurrent System Modelling

2.1 Process Calculi

In the thesis, we use formal language based approach for modelling concurrent systems. In particular, we use the approach which is generally referred to as process calculi, introduced and coined by Robin Milner with the work on the calculus of communicating systems [21]. A process calculus is a formal language much like a programming language having syntax and behaviour but where the number of operators is kept to a minimum. In this analogy, a process (agent) is a program. A concurrent system is then modelled as a process. This kind of approach is advantageous over other frameworks (e.g., automata theory, Petri nets) that it is inherently compositional in the sense that we model larger systems by composing them from smaller processes. Likewise, we can study a system by decomposing it into smaller systems that may be more amenable to formal treatment.

Formally, this is done by defining the syntax of operators representing states, and operational semantics by defining a mathematical relation denoting the evolution of a system from state to state. The use of syntax gives us powerful and yet simple to use mathematical tools for defining relations and proving properties of systems, namely structural recursion and induction. This method called structural operational semantics was first presented by Plotkin [28] for programming languages. It is usually a maxim to minimise the number of operators in the language in such a way that the captures the modelled system adequately. In practice, this translates to simpler definitions, proofs and tools.

There is a prominent alternative means to give meaning to the syntax called denotational semantics that, instead of describing state transitions, maps the processes to a mathematical object defined in, for example, set theory, domain theory [29], etc. However, for concurrent systems, such denotations tend to be more complicated than the operational semantics (cf. denotational semantics for the pi-calculus [12, 15]).

It is worth pointing out that there is strongly related line of work to process calculi, which is generally referred to as process algebra [3]. It based on the universal algebra view of processes. The processes are again a composition of operators, however, they are typically first-order algebra operators and specified using algebraic equational theories. However, the line is blurred and both lines of work use similar methods.

The idea is then to capture phenomena occurring in a distributed concurrent system by determining operators and giving them semantics that would express the behaviour of the studied system. The process calculi that we will consider build on well-recognised phenomena such as (1) *message passing* where processes interact by transmitting message over some communication medium, (2) *concurrency* where the interaction of processes is interleaved, (3) *non-determinism* where a behaviour of a process is not uniquely determined by the state, (4a) *point-to-point communication* where communication is only possible between two processes, (4b) *broadcast communication* where the communication is generalised to one-to-many communication, (5a) *synchronous communication* where the processes exchange messages at the same time, and (5b) *asynchronous communication* where a message received may have been sent in the past.

Process calculi may be broadly distinguished between pure and applied process calculi. The former designed to study the phenomena of concurrent systems in the purest form with minimal operators and semantics that describes it, while the latter sacrifices purity for the breadth of expressible systems, and uncomplicated description of models. Many pure process calculi have been devised over the years and used to study various concurrent systems: the aforementioned calculus of communicating systems [21] that captures synchronisation between two processes, the pi-calculus [22, 23] with point-to-point synchronous message-passing communication with mobility (briefly, reconfiguration of the process connectivity), the asynchronous pi-calculus [17, 8] obtained by omitting (!) operators from the pi-calculus to achieve *asynchronous* point-to-point communication semantics, the broadcast pi-calculus [11] generalisation to one-to-many, the concurrent constraint calculus [9] with synchronisation enabled by constraints, the ambient calculus [10] explores the notion of hierarchical locality, the distributed pi-calculus features locations and process migration between them.

There is a plethora of applied process calculi, since they are typically introduced in publications to study a particular system or a verification technique. There are simply too many to mention, and we note just the most prominent ones that strive for generality. One common feature is that most of them are based on the Milner, Parrow and Walker's pi-calculus, which is a testament to a good balance of abstraction and language primitives. The applied pi-calculus devised by Abadi and Fournet [1] extends the pi-calculus with what is essentially a concurrent storage, with structured data for channels, and with formulas for branching conditions. The spi-calculus by Abadi and Gordon [2] is similar in scope and precursor to the applied pi-calculus but it is more direct at introducing cryptographic primitives.

The proliferation of process calculi that are based on the pi-calculus suggests that there may be a unifying theory, a super pi-calculus if you will. This kind of unification is indeed a goal of the psi-calculi devised by Bengtson et al. [5]. The psi-calculi framework is a generalisation of the pi-calculus. It gen-

eralises the pi-calculus with arbitrary data and logic but keeps the semantics as close as possible to the original. It does lose some of the simplicity of the pi-calculus and thus falls under the applied calculi. However, many pure and applied calculi mentioned above are expressible to various degrees of accuracy in psi-calculi [5].

This thesis is concerned with the latter kind of process calculi, the applied process calculi. Papers II and IV are directly concerned with the psi-calculi. Paper I can be seen as a study of a semantics of more advanced process calculi. Finally, Paper III introduces a process calculus with arbitrary data and unreliable broadcast communication based on the pi-calculus.

In the following subsection, we give a brief informal description of syntax and semantics of these process calculi.

2.1.1 Pure Process Calculi

The purpose of this section is to informally introduce the kind of operators that are standard in process calculi. We start with the most prominent pure process calculi: CCS (the Calculus of Communicating Systems) [21] and the pi-calculus [22, 23], as they are almost universally used as the basis for other pure and applied process calculi, and the ideas are the simplest to appreciate.

In the following, we first describe CCS. With CCS, we can express point-to-point synchronisation (also known as rendezvous), non-determinism, and concurrency. We then show how the pi-calculus generalises and builds on CCS.

Calculus of Communicating Systems

CCS processes perform actions. An action can be thought as a signal propagating via the system of processes in such a way that its constituent processes can observe it. Actions also formalise the notion of an external observer of a process. Every action has two identities that are dual to each other. We can think of one side as being an input and the other as an output. In this regard, process actions are message passing, however, in CCS no data is exchanged. This is perhaps the most basic operation we could imagine a process can perform. Formally, CCS does not fix the set of actions; the set is left open as a parameter that we denote by $\mathscr{A}$. In CCS, this operation is also coupled with the sequencing operator. So, given any CCS process P, the following process

$$a.P$$

is an input prefixed process that can perform an input action a and continue as P, and the following process

$$\overline{a}.P$$

is an output prefixed process that can perform an output action $\overline{a}$ and likewise continue as P. These two operators are collectively known as *prefixes*.

We make use of the arrow notation when describing processes performing actions. We write $P \xrightarrow{a} P'$ to mean that a process P performs an action a and continues as P'. The above input prefix process behaviour can be written as $a.P \xrightarrow{a} P$, and the output prefix process can be written as $\overline{a}.P \xrightarrow{\overline{a}} P$.

The most basic process in CCS is the process that does nothing and is denoted by $\mathbf{0}$.

To give an example of a CCS process, consider a process that generates a clock signal twice. It can be described as two subsequent outputs of an action $\overline{\mathsf{tick}}$, as follows $\text{CLOCK} = \overline{\mathsf{tick}}.\overline{\mathsf{tick}}.\mathbf{0}$. The CLOCK process performs an action $\overline{\mathsf{tick}}$ and continues as $\overline{\mathsf{tick}}.\mathbf{0}$, and again performs an action $\overline{\mathsf{tick}}$ and continue as $\mathbf{0}$, that is, it stops performing any actions:

$$\text{CLOCK} \xrightarrow{\overline{\mathsf{tick}}} \overline{\mathsf{tick}}.\mathbf{0} \xrightarrow{\overline{\mathsf{tick}}} \mathbf{0}.$$

A corresponding component process would be inputting ticks with tick, the dual action of $\overline{\mathsf{tick}}$, for example, $\text{COMPONENT} = \mathsf{tick}.\mathbf{0}$. The COMPONENT process performs the tick action and stops as $\mathbf{0}$:

$$\text{COMPONENT} \xrightarrow{\mathsf{tick}} \mathbf{0}.$$

Clearly, the notion of an action is an abstraction. In the example, the dual actions of output $\overline{\mathsf{tick}}$ and input tick represent signals on a wire originating from opposite ends. It is helpful to think in terms of actions to build an understanding of how a process constructed from other processes behaves. This is one of the ideas used to formalise the semantics of processes as we shall later see in Section 2.2.

In the above example, we described processes and their behaviour individually. In order, to introduce concurrent interaction between processes we use parallel composition of CCS. Given any CCS process P and Q parallel composition is the following

$$P \mid Q.$$

The composition behaves in terms of its constituent processes. It can be read as P can perform an action and continue as P' and Q stays still, thus the composite system continues as $P' \mid Q$, using the notation

$$P \mid Q \xrightarrow{a} P' \mid Q.$$

Symmetrically, Q performs an action and continues as Q' while P does nothing, and the whole system then continues as $P \mid Q'$:

$$P \mid Q \xrightarrow{a} P \mid Q.$$

The symmetry captures that there is no significance in which order components of a parallel composition appear syntactically.

The composition may also result in synchronisation (communication): if P performs an input action a and continues as P', that is, $P \xrightarrow{a} P'$, and Q performs the *dual* output action $\overline{a}$ and continues as Q', that is $Q \xrightarrow{\overline{a}} Q'$, then the composition continues as $P' \mid Q'$ by performing the silent action τ:

$$P \mid Q \xrightarrow{\tau} P' \mid Q'.$$

The silent action is a predefined CCS action that is distinct from other actions, and is used to signify synchronisation (communication). The silent action has no dual. We also include a process prefixed with the τ action of the form $\tau.P$.

Let us return to the clock example. Consider the following system of parallel components:

$$\textsc{Component} \mid \textsc{Clock}.$$

Let us expand the definitions to:

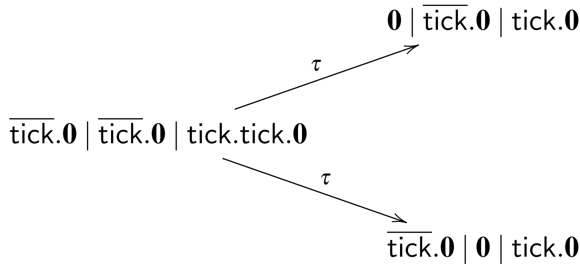
$$\overline{\mathsf{tick}}.\mathbf{0} \mid \mathsf{tick}.\mathsf{tick}.\mathbf{0}.$$

Also, let us only consider the behaviour resulting from synchronisations, that is, we ignore all actions except for the silent synchronisation action τ. Then, we have

$$\overline{\mathsf{tick}}.\mathbf{0} \mid \mathsf{tick}.\mathsf{tick}.\mathbf{0} \xrightarrow{\tau} \mathbf{0} \mid \mathsf{tick}.\mathbf{0}.$$

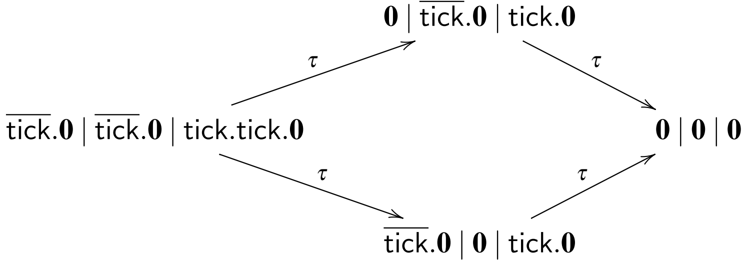
The first parallel component emitted the $\overline{\mathsf{tick}}$ action, the second component emitted the tick action, and then the derivative is a result of synchronisation.

The complexity of a parallel system may grow significantly. For example, if we simply add another $\textsc{Component}$, that is, $\textsc{Component} \mid \textsc{Component} \mid \textsc{Clock}$, then there are two possible continuations at this point: the first, $\mathbf{0} \mid \overline{\mathsf{tick}}.\mathbf{0} \mid \mathsf{tick}.\mathbf{0}$, and the second $\overline{\mathsf{tick}}.\mathbf{0} \mid \mathbf{0} \mid \mathsf{tick}.\mathbf{0}$. By using the arrow notation, we depict the transitions as follows



That is, either the first parallel component synchronises with the third, or the second component synchronises with the third. Both states then lead to the

same kind of process $\mathbf{0} \mid \mathbf{0} \mid \mathbf{0}$:



Thus, there are 4 possible states including the initial state. When the processes get more complex, the number of possible states rises exponentially due to all the interleaving one needs to consider. For example, just by adding an extra component to the above example process, i.e., $\overline{\mathsf{tick}}.\mathbf{0} \mid \overline{\mathsf{tick}}.\mathbf{0} \mid \overline{\mathsf{tick}}.\mathbf{0} \mid \mathsf{tick}.\mathsf{tick}.\mathbf{0}$, the system results in 10 possible states. The potential number of possible states that a process can evolve to makes the verification of such a system challenging since it is not always possible to list and check every state explicitly.

A CCS process can also act non-deterministically. Given processes P and Q, a choice (or, sum) is the process

$$P + Q.$$

The choice behaves as either P or Q. That is, if P performs an action and continues as P', then $P+Q$ performs the same action discarding Q and also continues as P', notationally if $P \xrightarrow{a} P'$, then $P+Q \xrightarrow{a} P'$. Similarly, if Q performs an action and continues as Q', then $P+Q$ also performs the very same action and continues as Q', notationally if $Q \xrightarrow{a} Q'$ then $P+Q \xrightarrow{a} Q'$. Thus, there are two possible evolutions of the system. This is the same idea as in non-deterministic automata where a state may have several outgoing transitions with the same action.

There are several standard ways to introduce unbounded behaviours in CCS and other process calculi. Perhaps the most common is to use a replication process of the following form

$$!P$$

where P is a process. Intuitively, the $!P$ process behaves as infinitely many copies of P in a parallel composition

$$P \mid P \mid \cdots$$

It is possible to describe this behaviour of such an operator in finite terms. Observe that at any given moment at most two processes can interact, and thus we can spawn a finite number of parallel components P while treating specially $!P$ as the rest of the infinite number of components: whenever $P \mid\ !P \xrightarrow{a} P'$ then $!P \xrightarrow{a} P'$.

$$
\begin{array}{rcll}
a & \in & \mathscr{A} & \text{action} \\
\\
P,Q & ::= & a.P & \text{input} \\
 & | & \overline{a}.P & \text{output} \\
 & | & \mathbf{0} & \text{inaction} \\
 & | & \tau.P & \text{silent} \\
 & | & P \mid Q & \text{parallel} \\
 & | & P+Q & \text{sum/choice} \\
 & | & !P & \text{replication}
\end{array}
$$

Figure 2.1. The grammar of a CCS process.

Another way is to introduce recursion, which is familiar to anyone who has used a modern programming language with recursion. We give names to processes as

$$A \stackrel{\text{def}}{=} P,$$

and then introduce a process

$$A$$

that invokes a process by its name. The process A behaves as its defining process P. The two notions of replication and recursion are strongly related. We can define the replicated process $!P$ with the definition $A \stackrel{\text{def}}{=} P \mid A$, and then invoke the process A. The other direction is slightly more involved. Returning to our example, we can define a process that generates indefinitely many clock signals

$$\textsc{Clock}' = !\mathsf{tick}.\mathbf{0}.$$

The system $\textsc{Clock}' \mid \textsc{Component}_1 \mid \ldots \mid \textsc{Component}_n$ then can accommodate as many components as necessary.

To summarise, the subset of CCS syntax that we presented here is given in *Figure 2.1.* Clearly, CCS is quite abstract and basic, however, it conveniently captures concurrent system phenomena for modelling: concurrency and synchronous communication. It is certainly more adept and direct at describing concurrent systems than for example non-deterministic finite automata theory. In the next section, we will see how CCS can be generalised further to systems with mobility.

The pi-calculus

The pi-calculus [22, 23][1] generalises CCS by introducing message passing between processes while preserving synchronous behaviour. The pi-calculus

[1] The work that introduces the pi-calculus is split into two papers.

process communication is linked via channels and they may change that linkage dynamically. The change of the communication structure is referred to as mobility.

We may still think in terms of actions. The basic data in pi-calculus is that of a name. A name is an abstract atomic entity like that of a CCS action in the previous section that represents both a channel and variable. We denote the (countably) infinite set of names as $\mathscr{N}$ and we use $a, b, c, \ldots$ to range over this set. Like CCS, the pi-calculus has actions for output, input and synchronisation. The input and output actions are compound, consisting of two names. The output action is denoted by $\overline{a}\,b$ for any names a and b, and intuitively means the name (data) b is outputted via the channel a. Similarly, the input action is denoted by $a\,b$ for any names a and b with the meaning that the name b is received via the channel a. The pi-calculus retains the silent action τ. The pi-calculus has one more action, which we are going to discuss later in this section.

The output prefixed process in the pi-calculus is, for any process P and any names a and b,

$$\overline{a}\,b.P.$$

The process performs an output action $\overline{a}\,b$, and continues as P:

$$\overline{a}\,b.P \xrightarrow{\overline{a}\,b} P.$$

Or more intuitively, a process outputs on the channel a the name b. The input prefixed process is

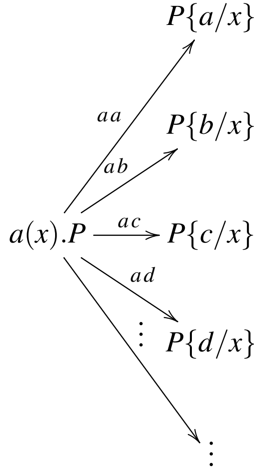
$$a(x).P$$

for any names a and x, and a pi-calculus process P. The input process performs an input action $a\,b$ on the channel a above for any name b, and then proceeds as P', which is a process where all free occurrences of x in P are substituted with b, denoted by $P' = P\{b/x\}$:

$$a(x).P \xrightarrow{a\,b} P\{b/x\}.$$

In other words, the process receives on the channel a a name b and continues using the received name in place of x. The pi-calculus input process can be viewed as a function with one parameter where the channel a represents the name of the function, x is the parameter, and the behaviour of input is an application of that function to the argument b. The difference with the usual notion of a function is that the argument b is not supplied in the syntax, but is part of the action. Thus, the argument may be supplied by a parallel process with an output action for the input to carry out the application of the parameter. The input process denotes infinitely many transitions that account for every

possible application of a name:



Unavoidably, the generalisation introduces some technicalities. In the input process above, the name x is a name that binds all free occurrences of x in P, and the name x itself is a binding name. The free occurrences of names are the set of names that are not bound. This may sound self-referential but it is not. The set of free names of $a(x).P$ are all the free names of P minus x and plus a; and the set of free names of $\overline{a}b.P$ is the set of free names of P plus a and b. For all other operators, that we have seen thus far, the free names are simply the free names of their subprocesses. The set of free names of a process P is denoted by $\mathrm{fn}(P)$. The distinction between a free and bound names is important because intuitively it does not matter what name we actually use for bound names while the free names are behaviourally significant.

Thus, for example, all of the following processes are considered equivalent $a(b).\overline{b}d.\mathbf{0}$, and $a(c).\overline{c}d.\mathbf{0}$, and $a(e).\overline{e}d.\mathbf{0}$. The equivalence identifying such renaming of bound names is called alpha-equivalence, and renaming of a process to an alpha-equivalent process is called alpha-conversion. The set of free names of the above processes consists of only a and d. Note that we avoided renaming bound variable to d, that is, capturing the name d in the above processes, as the process $a(d).\overline{d}d.\mathbf{0}$ is behaviourally distinct from the above processes. The capture-avoiding substitution $P\{a/x\}$ then renames bound names in P to avoid capture of a, that is, not to make a bound; also substitution never substitutes anything for bound names.

In the pi-calculus, the input and output processes synchronise in the very same way as CCS processes synchronise, that is, whenever the subprocesses perform dual actions. The parallel composition behaves as $P \mid Q \xrightarrow{\tau} P' \mid Q'$ whenever $P \xrightarrow{ab} P'$ and $Q \xrightarrow{\overline{a}b} Q'$. For example, $a(x).\overline{x}y.\mathbf{0} \mid \overline{a}b.\mathbf{0}$ synchronises by performing the silent τ action and continues as $\overline{b}y \mid \mathbf{0}$, because the left hand side parallel component performs the input action ab (among all the other

possible input actions) $a(x).\overline{x}y.\mathbf{0} \xrightarrow{ab} by.\mathbf{0}$ where $by.\mathbf{0} = (\overline{x}y.\mathbf{0})\{b/x\}$; and the right hand side performs $\overline{a}b$ by $\overline{a}b.\mathbf{0} \xrightarrow{\overline{a}b} \mathbf{0}$. Since synchronisation exchanges data, it is also called communication.

The pi-calculus processes can reconfigure their communication structure dynamically. The channel linkage determines the communication structure, and since the channels can be substituted by the input prefixed process with a received name, the process then can continue communicating on a received channel. For example, the process $a(x).\overline{x}b$ first inputs on channel a some value, say y, and then continues to output on the channel y. Thus, it does not output to a fixed channel but to a channel determined by another process. The dynamic reconfiguration of processes is referred to as mobility.

The pi-calculus is a generalisation of CCS as we still are able to recover CCS processes. Let w be a name for which we follow a simple rule that we never use it as a bound name nor as a channel. Then, the CCS operator $\overline{a}.P$ is captured by the pi-calculus operator $\overline{a}w.P$; and the CCS operator $a.P$ is captured by $a(x).P$ with the condition that x is not among the free names of P.

The pi-calculus also adds the following operators for testing name (channel) equality

$$[a = b]P,$$

and

$$[a \neq b]P$$

that means: if a is equal to b then behave as P, and, respectively, if a is not equal to b then behave as P. With these operators we can express the familiar conditional operator **if** $a = b$ **then** P **else** Q as $[a = b]P + [a \neq b]Q$.

A restricted process is

$$(\nu b)P$$

for any name a and pi-calculus process P. Restriction is a binder that binds the free occurrences of b in P. Intuitively, the name b represents a name that no process outside P has knowledge of. Thus, only processes that compose P can use it for sending as a value or communicating on it as a channel. However, the process P may communicate b to the outside process and thus extending the scope of b to include other processes. However, this is the only way to convey knowledge of b to other processes.

The output of a restricted name is described using a bound output action $\overline{a}(\nu b)b$ for a and b names, where (νb) denotes that the name b is restricted. For example, $(\nu b)\overline{a}b.P \xrightarrow{\overline{a}(\nu b)b} P$. In this transition, the name b in the action $\overline{a}(\nu b)b$ also binds into P.

The restricted name is seen as private. For example, in the process

$$(\nu b)(\overline{b}a.P \mid b(x).Q) \mid b(x).R$$

the scope of b extends to the first two parallel components. The bound b in the first two parallel components is considered to be distinct from b in $b(x).R$.

Thus, the following communication is possible

$$(\nu b)(P \mid Q\{a/x\}) \mid b(x).R$$

but not the first parallel component communicating with the third.

We also have a specialised communication rule to account for the possible bound actions and closing of the opened scope by extending it to the receiver: $P \mid Q \xrightarrow{\tau} (\nu b)(P' \mid Q')$ whenever $P \xrightarrow{\overline{a}(\nu b)b} P'$ and $Q \xrightarrow{a\,b} Q'$, in other words if P outputs a restricted name b on channel a, and Q inputs on channel a the name b (which is restricted) then the scope of the restriction of b is extended to also include the continuation of Q namely Q'. There may be a name b already present in Q (which is in this framework taken to be distinct from restricted name b), and we need to rename restricted b with some name that does not occur freely in Q.

In a distributed system, it is common to establish a session between a server and a client. A session records the communicating parties. It is natural to model this in the pi-calculus with the restriction mechanism. Take for example the following server and client processes where the server sends a private channel s on the server channel *srv* as a session and then continues exchanging on that channels with a client and then recurses. While the client requests a session by inputting a session channel x on the server channel *srv* and continues interacting on x.

$$\begin{aligned} \textsc{Server} &= (\nu s)\overline{srv}\, s.s(r).\overline{s}\,\mathsf{response}.\textsc{Server} \\ \textsc{Client} &= srv(x).\overline{x}\,\mathsf{request}.x(r).\mathbf{0} \end{aligned}$$

The following is a possible interaction. Note that the server and one of the clients have established a private session, and may continue interact in that session while the other client has no way of interfering or establishing its session with the server while the server is busy.

$$\begin{aligned} &\textsc{Server} \mid \textsc{Client} \mid \textsc{Client} \xrightarrow{\tau} \\ &\quad (\nu s)(s(r).\overline{s}\,\mathsf{response}.\textsc{Server} \mid \overline{s}\,\mathsf{request}.s(r).\mathbf{0}) \mid \textsc{Client} \end{aligned}$$

With the pi-calculus, we can capture a wider class of concurrent systems than with CCS. We gained the possibility of describing systems that change their communication structure dynamically, and model the notion of private communication. Also, computationally the pi-calculus is a powerful language and more expressive than CCS. In fact, it is as expressive as any general purpose programming language. This is shown by encoding [22] the Church's untyped lambda calculus [4], which is Turing complete, to the pi-calculus. In summary, the language of the pi-calculus that we have defined thus far is found in the *Figure 2.2*. The pi-calculus is the basis for many applied calculi. In the next section, we describe a family of such calculi: psi-calculi.

$$
\begin{array}{rcll}
a,b,x & \in & \mathcal{N} & \text{name} \\
\\
P,Q & ::= & a(x).P & \text{input} \\
 & | & \overline{a}\,b.P & \text{output} \\
 & | & P \mid Q & \text{parallel} \\
 & | & P+Q & \text{sum/choice} \\
 & | & [a=b]P & \text{match} \\
 & | & [a\neq b]P & \text{mismatch} \\
 & | & !P & \text{replication} \\
 & | & (\nu x)P & \text{restriction} \\
 & | & \mathbf{0} & \text{inaction}
\end{array}
$$

Figure 2.2. The grammar of pi-calculus processes.

2.1.2 Applied Process Calculi

Typically, when modelling real world concurrent systems one needs to model not only the communication and concurrency aspects but also data and computation on the data that is exchanged, e.g., integers, strings, lists, routing tables, cryptographic operations, etc. It is well known that we can encode all computable functions in the pi-calculus (via the encoding of the lambda calculus [22]), however, modelling with encodings may be too complex.

Let us illustrate with a simple example. Suppose we want to model a server that computes a Boolean conjunct from the values it receives. We can model it as the following process

$$\text{AND}(x,y,z) = x(a).([a=\mathsf{T}]\,y(w).\bar{z}w \;+\; [a=\mathsf{F}]\,y(w).\bar{z}\mathsf{F}.\mathbf{0})$$

that receives in sequence a Boolean value from channel x, and then one from channel y. Since the only data in the pi-calculus are names, we treat the distinct names T and F specially as the Boolean values. So the server first receives a value and then does a case analysis on the received name. If the value received is true (i.e., equal to T), then it simply forwards the value received on the other channel y to z; if the value is false, then always output F on z no matter what value arrived on y; otherwise, the process is stuck. It is not at all straightforward to tell whether the AND process realises the Boolean connective. Of course, we could build a library of processes modelling various kinds of data. However, it is a significant effort to find such encodings and verify that they are correct. What is more, often encodings require communication to drive computation, thus introducing additional behaviour.

The approach taken by applied process calculi is to instead reuse the theories that have been developed for data and to extend, or add, operations to

handle the data. In an applied process calculus with the Boolean $a \wedge b$ operation, the AND server could instead be modelled as

$$\text{AND}(x,y,z) = x(a).y(b).\bar{z}(a \wedge b).$$

The spi-calculus [2], for example, introduces, on top of the pi-calculus, primitives necessary to model cryptographic protocols. It extends the data domain with natural numbers, ordered pairs, shared cryptographic keys, public key cryptography, among others. It also adds operators for handling the cryptographic data like encryption and decryption of messages using a shared key, and others.

The applied pi-calculus [1] goes further. It has no built-in operations on data but parametrises the data with user supplied operations and equations that act as computation rules. For example, symmetric key cryptography can be specified with two operations decrypt and encrypt and the equation $\mathsf{decrypt}(\mathsf{encrypt}(M,k),k) = M$. In this way, it is possible to express the kind of primitives the spi-calculus has. The applied pi-calculus introduces a compositional store as a process called an active substitution $\{^{M}/_{x}\}$ for data M and variable x. Processes may read information stored therein via the variable x if they are in a parallel composition with an active substitution. In this way, it is possible to share data among processes such as a private key.

It is not at all obvious how one would encode the kind of operations described above in the pi-calculus. Furthermore, having the right primitives allows for straightforward modelling and analysis of a concurrent system.

The challenge then is to find appropriate operators that are general enough and show that extensions preserve the behaviour and do not interfere with the basic operators. There is a need for systematic study of such extensions, and this is what Bengtson et al. [5] does with the psi-calculi. The papers II and IV are directly concerned with the psi-calculi. In the next section, we introduce psi-calculi briefly.

Psi-calculi

The Psi-calculi framework generalises the pi-calculus even further and is a theory that unifies many extensions of the pi-calculus that have been introduced (see [5] and Paper II for examples).

Psi-calculi generalises the pi-calculus by extending the data domain beyond names to arbitrary sets, and by extending the tests that the processes can perform not just on the name equality or disequality but arbitrary formulas. Similarly to the applied pi-calculus, the psi-calculi introduces a process that contains some state about the data that processes share, which contains arbitrary logical assertions that affect the tests that the processes can perform.

Psi-calculi has the same kind of actions as the pi-calculus for input, output, bound output, and communication. However, the names are generalised to

an arbitrary set of terms[2] denoted by **T** ranged over by M,N, which is a parameter in the psi-calculi framework. The terms themselves may contain names. The input action is MN where M and N are terms with a similar meaning to the pi-calculus, where the data N is received via the channel M. Thus, in psi-calculi it is possible to use structured data like integers, lists, and trees as channels and transmitted data. Output and bound output generalise in a similar fashion. Output is $\overline{M}N$ and the bound output is $\overline{M}(\nu a_1,\ldots,a_n)N$ for terms M and N. Since N in bound output may contain multiple names and thus extend the scope for more than one name, bound output uses a sequence of names $a_1,\ldots,a_n$ to record this fact.

The psi-calculi generalises the pi-calculus input by replacing the names with terms. The input construct of psi-calculi also admit pattern matching, thus input not only receives a value but matches it accordingly and binds it to the names that are considered as pattern variables. The syntax is the following

$$M(\lambda x_1,\ldots,x_n)N.P$$

where $x_1,\ldots,x_n$ are the pattern variables, N is the pattern and M is the channel.

The behaviour of the input process is also generalised in psi-calculi. Since the channels may be structured, a condition is introduced, called channel equivalence, to determine their equality[3] $M \leftrightarrow N$ for channels M and N. The psi-calculi also introduces a logic as a parameter: a set of conditions **C** ranged over by φ, a set of assertions **A** ranged over by Ψ, and an entailment relation $\Psi \vdash \varphi$ determining the condition φ that is made true by the assertion Ψ.

Channel equivalence is also a condition that is asserted by the entailment relation. The substitution function is also a parameter in psi-calculi, written as $[x_1 := N_1,\ldots,x_n := N_n]$ for $x_1,\ldots,x_n$ names and $N_1,\ldots,N_n$ terms. In psi-calculi, the transitions are indexed by assertions. So, operationally, the input process works as follows

$$\Psi \rhd M(\lambda x_1,\ldots,x_n)N.P \xrightarrow{M'N'} P[x_1 := N_1,\ldots,x_n := N_n]$$

where the process $M(\lambda x_1,\ldots,x_n)N.P$ performs an action $M'N'$ such that M and M' are channel equivalent in the current assertion environment Ψ, that is, $\Psi \vdash M \leftrightarrow M'$. Furthermore, the pattern N pattern matches the data N' by binding the matches $N_1,\ldots,N_n$ to the variables $x_1,\ldots,x_n$, formally, $N[x_1 := N_1,\ldots,x_n := N_n] = N'$.

The psi-calculi processes for the output are similar to those of the pi-calculus with the obvious generalisation:

$$\overline{M}N.P$$

[2] In fact, the sets are required to be nominal sets [13, 27], however, this is a very mild restriction. A nominal set is simply a set that may mention atomic objects that are not sets and are taken to represent names of a process calculus, and for each element of a nominal set, there is a permutation action that permutes the mentioned names of that element. Trivially, every set is a nominal set for which permutation action is the identity function.

[3] Partial equivalence.

The behaviour of this process is similar to the pi-calculus as well

$$\Psi \rhd \overline{M} N.P \xrightarrow{\overline{M'} N} P$$

where $\Psi \vdash M \leftrightarrow M'$.

The **case** $\varphi_1 : P_1 \,[]\, \cdots \,[]\, \varphi_n : P_n$ construct is a generalisation of the non deterministic choice, match and mismatch operators of the pi-calculus where the name equality and disequality are replaced by the arbitrary conditions $\varphi \in \mathbf{C}$. For example, if we have negation $\neg$ in the condition language defined in $\mathbf{C}$, then if-then-else **if** φ **then** P **else** Q can be defined as **case** $\varphi : P \,[]\, \neg\varphi : Q$. The choice of the pi-calculus can be recovered as follows. Given that there is an always entailed condition $\mathtt{true}$ (i.e., for any Ψ, $\Psi \vdash \mathtt{true}$), then $P + Q$ can be expressed in psi-calculi as **case** $\mathtt{true} : P \,[]\, \mathtt{true} : Q$.

The novel construct in the psi-calculi is the assertion process:

$$(\!|\Psi|\!)$$

By itself it has no behaviour but what it does is to contribute to the current assertion environment of the parallel processes via the assertion composition $\otimes$, which is a parameter in psi-calculi.

The behaviour of the parallel operator in psi-calculi is generalised to account for the presence of assertions as parallel components and to propagate the current assertion to processes.

Let us take a simple example. Let terms $\mathbf{T}$ be the name set; assertions $\mathbf{A}$ be the finite sets of names, and conditions $\mathbf{C}$ be simply pairs of names. Then, define $a \leftrightarrow b$ to be (a,b), and $\Psi \vdash (a,b)$ if $\{a,b\} \subseteq \Psi$, and lastly $\Psi \otimes \Psi' = \Psi \cup \Psi'$. In the following we assume that $a \neq b$. Consider the following process

$$R = (\!|\{a\}|\!) \mid (\!|\{b\}|\!) \mid a(\lambda x)x.P \mid \overline{b}c.Q.$$

The process can behave as

$$R \xrightarrow{\tau} (\!|\{a\}|\!) \mid (\!|\{b\}|\!) \mid P[x := c] \mid Q$$

due to the following facts. The shared assertion environment Ψ is a composition of all parallel assertions $\{a\} \otimes \{b\}$. By definition, $\Psi = \{a\} \otimes \{b\} = \{a\} \cup \{b\} = \{a,b\}$. Then, we need to check that the channel equivalence is entailed $\Psi \vdash a \leftrightarrow b$ by expanding the definitions $\{a,b\} \vdash (a,b)$ iff $\{a,b\} \subseteq \{a,b\}$, which is true. Finally, pattern matching is successful, that is, $x[x := c] = c$.

The following process, however,

$$R' = (\!|\{a\}|\!) \mid a(\lambda x)x.P \mid \overline{b}c.Q \not\xrightarrow{\tau}$$

has no transitions since $\{a\} \vdash a \leftrightarrow b$ does not hold, as $\{a,b\} \not\subseteq \{a\}$. This example illustrates the fact that in psi-calculi communication is determined not necessarily by the identity check on the channels, and that communication can be disabled by breaking the channel linkage.

$$
\begin{array}{rcll}
x & \in & \mathcal{N} & \text{name} \\
M,N & \in & \mathbf{T} & \text{term} \\
\varphi_1,\ldots,\varphi_n & \in & \mathbf{C} & \text{condition} \\
\Psi & \in & \mathbf{A} & \text{assertion} \\
 & & & \\
\dot{\leftrightarrow} & \in & \mathbf{T}\times\mathbf{T}\to\mathbf{C} & \text{channel equivalence} \\
\otimes & \in & \mathbf{A}\times\mathbf{A}\to\mathbf{A} & \text{assertion composition} \\
\vdash & \subseteq & \mathbf{A}\times\mathbf{C} & \text{entailment} \\
[\widetilde{x}:=\widetilde{N}] & \in & \mathbf{T}\to\mathbf{T} & \text{substitution function} \\
 & & & \\
P,Q & ::= & M(\lambda\widetilde{x})N.P & \text{input} \\
 & | & \overline{M}N.P & \text{output} \\
 & | & P \mid Q & \text{parallel} \\
 & | & !P & \text{replication} \\
 & | & (\nu x)P & \text{restriction} \\
 & | & (\!|\Psi|\!) & \text{assertion} \\
 & | & \mathbf{case}\,\varphi_1 : P_1 \,[\!]\cdots[\!]\, \varphi_n : P_n & \text{case} \\
 & | & \mathbf{0} & \text{inaction}
\end{array}
$$

Figure 2.3. The parameters and grammar of psi-calculi processes.

A significant modelling flexibility of psi-calculi comes from the fact that $\Psi \vdash \varphi$ can be seen as a two-valued logic. In the above example, we interpreted the assertions as a set of names that are known to be equivalent, and the channel equivalence condition as an equality query. We could as well take the assertions to be sets of equations on terms, and the conditions to be also equations, then the $\vdash$ can be defined to be a proof derivation of this equational logic. We can take this even further, we could define assertions to be sets of predicate formulas (including the universal and existential quantifiers), and likewise conditions to be formulas, then $\vdash$ could be defined as a validity relation of the predicate logic or proof derivation relation. Thus, in psi-calculi it is quite straightforward to reuse already developed theories, e.g., of data structures, cryptographic primitives, etc.

Psi-calculi has also been extended to encompass more process calculi: the higher order communication [25], and unreliable broadcast communication [7]. The full syntax of psi-calculi is given in *Figure 2.3*, where we use $\widetilde{x}$ and $\widetilde{N}$ to denote arbitrary sequences $x_1,\ldots,x_n$ and $N_1,\ldots,N_n$.

Psi-calculi are a major part of this thesis. The Psi-calculi framework is the main subject of Paper II and Paper IV. The logic of Paper I arose from considering the generalised actions with multiple binders and the behavioural

equivalences of psi-calculi. Paper VI, the precursor of Paper III, uses psi-calculi via encoding to give meaning to a broadcast process calculus.

2.2 Formal Semantics

The two most common methods of formalising the meaning of processes in process calculi are structural operational semantics (SOS) and reduction semantics. Both of these methods formalise a transition relation that describes how a process evolves from state to state.

The idea is to induce a structure called a *labelled transition system*. A labelled transition system is a tuple $(\rightarrow, \mathscr{A}, \mathscr{P})$ where $\mathscr{A}$ is a set of actions (labels), $\mathscr{P}$ is a set of processes (also called *states*), and $\rightarrow \subseteq \mathscr{P} \times \mathscr{A} \times \mathscr{P}$ is a transition relation, written $P \xrightarrow{\alpha} P'$ for $(P, \alpha, P') \in \rightarrow$. If the set of labels is a singleton set, then the structure is called simply a *transition system* and is isomorphic to $(\rightarrow, \mathscr{P})$ where $\rightarrow \subseteq \mathscr{P} \times \mathscr{P}$. The labelled transition system can be thought of as a directed graph with labelled edges. These kind of structures are used to formalise the arrow notation that we used informally in Section 2.1.

Typically, when defining a transition system with either SOS or reduction semantics, one makes use of *structural congruence* to reduce the number of SOS rules or to convert a process into a shape matched by reduction rules. A structural congruence is a congruence relation[4], denoted by $\equiv$, that captures the most basic and intuitive invariants of a process syntax, e.g. the commutativity of a parallel operator. Usually, a structural congruence is defined inductively on the structure of processes.

For example, a structural congruence may include such facts that the parallel operator is commutative, associative and that its identity is the inaction process **0**:

$$\begin{array}{rcl} P \mid Q & \equiv & Q \mid P \\ (P \mid Q) \mid R & \equiv & P \mid (Q \mid R) \\ P \mid \mathbf{0} & \equiv & P \end{array}$$

In process calculi like the pi-calculus with a restriction operator, it is common to include in the structural congruence the scope extrusion law:

$$(\nu a)P \mid Q \equiv (\nu a)(P \mid Q) \quad \text{if } a \notin \mathrm{fn}(Q)$$

2.2.1 Structural Operational Semantics

The structural operational semantics (SOS) approach consists of defining a set of rules specifying a mathematical relation on processes to formalise the state transitions of a process. The key aspect is that the rules are defined inductively

[4] An equivalence relation that is preserved by all of the operators of the language.

on the syntax of the processes. The name structural refers to this aspect. The rules are of the form

$$\frac{p_1 \xrightarrow{\alpha_1} p'_1 \cdots p_n \xrightarrow{\alpha_n} p'_n}{p \xrightarrow{\alpha} p'}$$

where p, q, $p_1, \ldots, p_n$, and $p'_1, \ldots, p'_n$ are processes, and $\alpha_1, \ldots, \alpha_n$ are actions. Intuitively, the rule reads: whenever p_1 performs an action α_1 and continues as p'_1 and analogously for other process p_i for $i = 2, \ldots, n$, then p performs an action α and continues as p'. The terms above the line are called premises, and the term below the line is called conclusion. The premises may be empty, in that case we call the rule an axiom. The rules may also contain logical formulas, often called side conditions that may additionally constrain the type of processes and actions used.

As an example, let us take look at how one can formalise the somewhat informal description of processes behaviour that we have presented in Section 2.1. Let us take a subcalculus of CCS (Section 2.1.1) defined by the following grammar (subset of *Figure 2.1*)

$$P, Q ::= \mathbf{0} \;\Big|\; a.P \;\Big|\; \overline{a}.P \;\Big|\; P \mid Q \quad .$$

The rules for actions are simply the following:

$$\frac{}{a.P \xrightarrow{a} P} \qquad \frac{}{\overline{a}.P \xrightarrow{\overline{a}} P}$$

The parallel operator behaviour can then be described with SOS rules as follows where the α is either a or $\overline{a}$ for some action a.

$$\frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \qquad \frac{Q \xrightarrow{\alpha} Q'}{P \mid Q \xrightarrow{\alpha} P \mid Q'}$$

A parallel process behaves the same way as either of its parallel components do (see Section 2.1.1). The synchronisation of a parallel composition can be captured by the following rules:

$$\frac{P \xrightarrow{a} P' \quad Q \xrightarrow{\overline{a}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'} \qquad \frac{P \xrightarrow{\overline{a}} P' \quad Q \xrightarrow{a} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

Note the rules are solely syntax-directed: they define the behaviour of a process by its structure. That is, $P \mid Q$ transition with the action τ to $P' \mid Q'$ above is defined in terms of the transition from P to P' with the action a, and Q transition to Q' with the action $\overline{a}$.

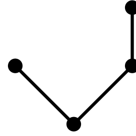
To illustrate, let us use the rules to derive the transition

$$a.\mathbf{0} \mid (\overline{b}.\mathbf{0} \mid \overline{a}.\mathbf{0}) \xrightarrow{\tau} \mathbf{0} \mid (\overline{b}.\mathbf{0} \mid \mathbf{0})$$

where the first and the last parallel components synchronise:

$$\dfrac{a.\mathbf{0} \xrightarrow{a} \mathbf{0} \quad \dfrac{\overline{a}.\mathbf{0} \xrightarrow{\overline{a}} \mathbf{0}}{\overline{b}.\mathbf{0} \mid \overline{a}.\mathbf{0} \xrightarrow{\overline{a}} \overline{b}.\mathbf{0} \mid \mathbf{0}}}{a.\mathbf{0} \mid (\overline{b}.\mathbf{0} \mid \overline{a}.\mathbf{0}) \xrightarrow{\tau} \mathbf{0} \mid (\overline{b}.\mathbf{0} \mid \mathbf{0})}$$

By enumerating all the possible applications of the above rules, we would get the possible transitions describing the behaviours of a given process. The way the rules are applied resembles a tree



where the conclusion is the root and premises are branching subtrees, and a rule without a premise is a leaf node.

This observation leads to the two most common interpretations of the rules: (1) as the smallest relation with regard to set inclusion that is satisfied by the rules, let us write it as $\rightarrow$, which is a subset of $\mathscr{P} \times \mathscr{A} \times \mathscr{P}$ where $\mathscr{P}$ is a set of processes as defined by the grammar, and $\mathscr{A}$ is the set of actions; and (2) as a tree construction by the rules which we already witness in the example above. The first interpretation gives what is known as rule induction. Thus, to prove a property of a process transitions, say the set $P \subseteq \mathscr{P} \times \mathscr{A} \times \mathscr{P}$, one only needs to show that it satisfies the rules, and thus whenever we use the smallest relation $\rightarrow$ to derive processes the property P is implied since $\rightarrow \subseteq P$.

The second interpretation allows us to use the complete induction principle[5] on natural numbers to show properties on transition systems. One can associate a number with a tree usually called depth or length that is recursively defined to be the maximum of depths of its subtrees plus 1, and leafs have 1. Intuitively, the number of a tree is just the length of its longest path to a leaf node from the root. So, a conclusion always has a depth larger than its premises, and thus one can use the following induction principle based on complete induction on the depth: to show a property P holds, one needs to show, for each rule, that it holds for the conclusion by assuming that it holds for its premises. This induction principle is typically invoked with "by induction on the depth of derivation..."

The syntax-directed nature of the rules, and the resulting intuitive induction principles are key advantages of the SOS approach, and perhaps this is why SOS is so prevalent in process calculi. We use this kind of semantics in Paper II and Paper IV.

[5] $P(n)$ for all natural numbers n follows if one can prove $P(m+1)$ by assuming that for all $i \leq m$ holds $P(i)$.

2.2.2 Reduction Semantics

Reduction semantics formalise only the process evolution that results from communication (synchronisation). As with SOS, reduction semantics defines a transition relation from a set of inductive rules. The rules again match the syntax of a process, however, behaviour is defined not based on the substructure of a process, but rather on the form of a process that can proceed in communication.

We can formalise the communication for CCS with the following rule:

$$a.P \mid \overline{a}.Q \to P \mid Q$$

The rule says that whenever there is a parallel component with processes that are prefixed with dual actions, then it can proceed by consuming those actions, i.e., synchronise. This one rule is not sufficient to allow us to reduce more complex processes, so the following rules are typically included

$$\frac{P \equiv P' \quad P' \to Q' \quad Q' \equiv Q}{P \to Q} \qquad\qquad \frac{P \to P'}{P \mid Q \to P' \mid Q}$$

where the first rule says that reductions are invariant under the structural congruence. Thus, if the structural congruence includes associativity and commutativity of parallel operator, the parallel composition is a kind of solution [6] where parallel components float freely to form a reduction term that can be matched by the basic reduction rules like the synchronisation reduction rule given above. The second rule says that we can reduce the parallel composition in terms of its parallel components (in lieu of the structural congruence rule).

Let us develop a reduction of the process $a.\mathbf{0} \mid (\overline{b}.\mathbf{0} \mid \overline{a}.\mathbf{0})$ from the previous section:

$$\dfrac{a.\mathbf{0} \mid (\overline{b}.\mathbf{0} \mid \overline{a}.\mathbf{0}) \equiv (a.\mathbf{0} \mid \overline{a}.\mathbf{0}) \mid \overline{b}.\mathbf{0} \quad \dfrac{\dfrac{}{a.\mathbf{0} \mid \overline{a}.\mathbf{0} \to \mathbf{0} \mid \mathbf{0}}}{(a.\mathbf{0} \mid \overline{a}.\mathbf{0}) \mid \overline{b}.\mathbf{0} \to (\mathbf{0} \mid \mathbf{0}) \mid \overline{b}.\mathbf{0}} \quad (\mathbf{0} \mid \mathbf{0}) \mid \overline{b}.\mathbf{0} \equiv \mathbf{0} \mid (\overline{b}.\mathbf{0} \mid \mathbf{0})}{a.\mathbf{0} \mid (\overline{b}.\mathbf{0} \mid \overline{a}.\mathbf{0}) \to \mathbf{0} \mid (\overline{b}.\mathbf{0} \mid \mathbf{0})}$$

In the above, we used the structural congruence rule to shuffle the parallel components into the right order so that we can ignore the right most parallel component to get a reducible process.

For the pi-calculus, one also includes a rule that describes reduction within the scope of a restriction operator

$$\frac{P \to P'}{(\nu a)P \to (\nu a)P'}$$

and the structural congruence also includes the scope-extrusion law: $(\nu a)P \mid Q \equiv (\nu a)(P \mid Q)$ if $a \notin \mathrm{fn}(Q)$. Also, the communication rule is straightforward in the pi-calculus:

$$a(x).P \mid \overline{a}b.Q \to P\{b/x\} \mid Q$$

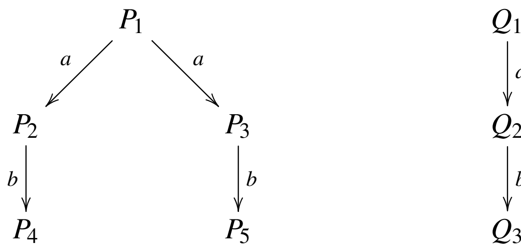
Reduction semantics are used prominently as they are fairly straightforward to understand and usually have fewer rules than SOS. However, the rules are not defined on the substructure of a process, and can be quite non-trivial to use if one is interested in proving properties with the structural induction on the process syntax since the reduction rules are only defined on particular form of a process. This also makes reduction rules more complex when the process calculus has the choice operator. The reduction semantics approach is quite attractive if one only wants to show properties on a system that has no external observer, such as the one that is found in Paper III.

2.3 A Behavioural Equivalence: Bisimilarity

The standard notion of equivalence on processes in process calculi is labelled *bisimilarity*. Bisimilarity is a behavioural equivalence: it is defined on the observed actions that are performed by the processes, and not on the structure (i.e. syntax) of processes. In fact, it is definable directly on a labelled transition system.

Bisimilarity is defined in terms of bisimulation relations. Bisimulation, as the name suggests, is in turn defined in terms of simulation, that is, bisimilar processes simulate each other in lockstep. Simulation is a property of two processes such that one can mimic the actions taken by the other process. More specifically, take two processes P and Q. We say Q simulates P, if a process P has a transition $P \xrightarrow{\alpha} P'$, then the process Q must be able to repeat this action with a transition $Q \xrightarrow{\alpha} Q'$ and furthermore Q' must be able to continue repeating actions taken by P' and so on.

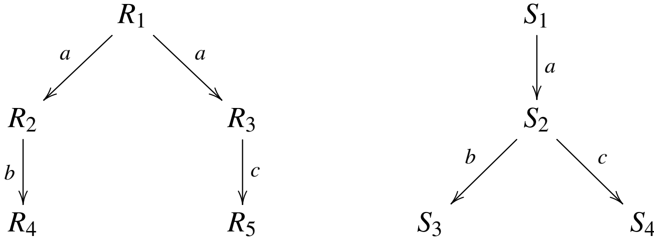
Consider the following two transition systems:



In the following we argue that the process Q_1 simulates the process P_1. The process P_1 has two possible transitions with the same action a to two states P_2 and P_3. Now, if it chooses the first transition to P_2 with the action a, then Q_1 can repeat the same action by transitioning to Q_2. Then, P_2 can only do one transition to P_4 with the action b; Q_2 easily mimics this by transitioning to Q_3. P_4 has no transition. Thus Q_3 does not need to repeat any actions, and incidentally Q_3 also has no transitions. If P_2 choose the latter transition with the action a to P_3, then Q_1 can again repeat it with the transition to Q_2, the

argument plays out in the same fashion from process P_3 as from P_2. In the same way, we see that P_1 can simulate Q_1 as well.

Now consider the following two systems (note that the action labels differ from the previous systems):

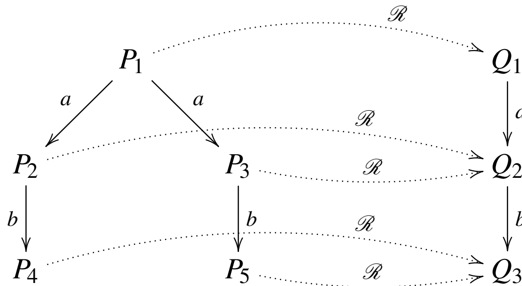


Here R_1 does *not* simulate S_1 because of the following. The only transition that S_1 can make is to S_2 with the action a, and the R_1 has to make a choice in order to be able to simulate S_1 either to R_2 or R_3. Suppose the transition to R_2 was made. However, S_2 now can transition to S_3 with b or to S_4 with c, but R_2 can only mimic a transition with b to R_4 and cannot mimic a transition with c. If R_1 had made a choice to transition to R_3, then R_3 can only mimic a transition with c but not with b.

Let us use the notation $A \mathscr{R} B$ to mean $(A,B) \in \mathscr{R}$ for a set $\mathscr{R}$ which we call binary relation. Formally, we define simulation as a property on a binary relation on processes, that is, $\mathscr{R} \subseteq \mathscr{P} \times \mathscr{P}$.

The binary relation $\mathscr{R}$ is a *simulation*, whenever for all processes P and Q, if $P \mathscr{R} Q$, then the following holds: if, for all α and P', we have $P \xrightarrow{\alpha} P'$, then there is Q' such that $Q \xrightarrow{\alpha} Q'$ and furthermore $P' \mathscr{R} Q'$.

Returning to the first example, formally Q_1 simulates P_1, as witnessed by the relation $\mathscr{R} = \{(P_1,Q_1),(P_2,Q_2),(P_3,Q_2),(P_4,Q_3),(P_5,Q_3)\}$. That is, P_1 and Q_1 are related by $P_1 \mathscr{R} Q_1$. So, the simulation relation $\mathscr{R}$ relates the nodes of the transition system as follows where the dotted arrows denote an ordered pair in the relation:



The *bisimulation* relation $\mathscr{R}$ is defined by simply requiring that the simulation relation $\mathscr{R}$ is symmetric, that is, if $P \mathscr{R} Q$, then also $Q \mathscr{R} P$. Thus, for the above example, the relation $\mathscr{R}' = \mathscr{R} \cup \mathscr{R}^{-1}$ is a bisimulation relation where $\mathscr{R}^{-1} = \{(Q,P) : (P,Q) \in \mathscr{R}\}$. Thus, P_1 and Q_1 are bisimilar, that is, $P_1 \mathscr{R}' Q_1$.

The *bisimilarity* relation denoted by $\sim$ is defined to be the largest bisimulation relation with regard to set inclusion. And it is the equivalence relation that we have been after. So, in the above example, P_1 is bisimilar to Q_1, that is, $P_1 \sim Q_1$ because $P_1 \mathscr{R}' Q_1$ implies that $P_1 \sim Q_1$. This follows from the fact that we defined bisimilarity to be the largest bisimulation, that is, any other bisimulation is included in bisimilarity $\mathscr{R}' \subseteq \sim$. So to prove that two processes are bisimilar, we need to find a bisimulation relation that includes those processes[6]. Alternatively, bisimilarity may be defined as $P \sim Q$ if there is a bisimulation relation $\mathscr{R}$ such that $P \mathscr{R} Q$. However, these two definitions are equivalent.

The bisimilarity relation is defined in the same way for the CCS process calculus. The first transition system can be expressed as the CCS process $P_1 = a.b.\mathbf{0} + a.b.\mathbf{0}$, and the second as $Q_1 = a.b.\mathbf{0}$. Thus,

$$a.b.\mathbf{0} + a.b.\mathbf{0} \ \sim \ a.b.\mathbf{0}$$

The transition systems from the second example can be described as $R_1 = a.b.\mathbf{0} + a.c.\mathbf{0}$ and $S_1 = a.(b.\mathbf{0} + c.\mathbf{0})$. Therefore, we have that

$$a.b.\mathbf{0} + a.c.\mathbf{0} \not\sim a.(b.\mathbf{0} + c.\mathbf{0})$$

as there is no bisimulation relation.

The definition of bisimulation relation becomes slightly more complicated for more advanced calculi. In the pi-calculus, we need to be careful at picking sufficiently fresh names while simulating transitions with bound output actions. In the psi-calculi, in addition to the same concerns as in the pi-calculus, the bisimulation relation is expanded with more clauses and indexed with an assertion that make sure the assertion environment of the simulating processes enables the same conditions, and that the simulation is possible even after expanding the current assertion in all possible ways.

The bisimulation relation that we defined is usually known as *strong* bisimulation as the processes mimic each other's transitions exactly. In practice, the processes may do some internal computation and generate τ transitions that we may not want the other process to mimic. Thus, strong bisimulation may be too strong and distinguish between processes that we would like to be identified. To address this the notion of *weak* bisimulation is introduced that ignores the τ transitions.

The relation $\mathscr{W}$ is a weak bisimulation if $\mathscr{W}$ is symmetric and for all $P \mathscr{W} Q$:

- if $P \xrightarrow{\tau} P'$, then there is Q' that $Q \xrightarrow{\tau} \cdots \xrightarrow{\tau} Q'$, and $P' \mathscr{W} Q'$; and
- if $P \xrightarrow{\alpha} P'$, then there is Q' that $Q \xrightarrow{\tau} \cdots \xrightarrow{\tau}\xrightarrow{\alpha}\xrightarrow{\tau} \cdots \xrightarrow{\tau} Q'$, and $P' \mathscr{W} Q'$;

Note in the above definition $\xrightarrow{\tau} \cdots \xrightarrow{\tau}$ may be an empty sequence of transitions. Thus, a process ignores the τ transitions while simulating.

Weak bisimilarity $\approx$ is defined in the same way as in the strong case as the largest weak bisimulation. For example, in CCS, $\tau.b.\mathbf{0} \ \approx \ b.\mathbf{0}$.

[6] This is an example of coinduction. Indeed, the bisimilarity relation is a coinductive relation.

Labelled bisimulation is central to the work in this thesis. We restate the definitions and theorems concerning bisimulation in Paper II to gain assurance of the correctness of our new development of sorts and pattern matching. In the tool in Paper IV, we implement both strong and weak bisimulation generation algorithms for reasoning with processes. A modal logic should not be able to distinguish between processes that are bisimilar (Section 2.4), and we carry out this test for our logic in Paper I.

2.4 Logic for Transition Systems

A prominent method of expressing properties of a transition system was introduced by Hennessy and Milner [16] as a variant of modal logic now called Hennessy-Milner Logic (HML). The logic consists of modal formulas, in addition to the standard logic connectives, for testing whether processes may or must make a transition with a specified action. A transition system is then viewed as a model of an HML formula, and reciprocally the logic is viewed as an observer of a transition system.

The logic induces an equivalence between processes such that two processes are logically equivalent whenever they are indistinguishable by the logic, i.e. processes satisfy exactly the same formulas. A logic is called adequate if the induced logic equivalence corresponds to the behavioural equivalence of a given transition system. That is, given two behaviourally equivalent processes, there is no formula in the logic that is satisfied by one process but not the other, and if the processes are logically equivalent, then they are also behaviourally equivalent. The adequacy property is desirable as it ensures that the logical properties that we verified for a process still hold for behaviourally equivalent processes.

Given a labelled transition system $(\rightarrow, \mathscr{A}, \mathscr{P})$ (Section 2.2) where $\rightarrow \subseteq \mathscr{P} \times \mathscr{A} \times \mathscr{P}$ is a transition relation, $\mathscr{A}$, ranged over by α, is a set of actions, and $\mathscr{P}$ is a set of processes. Then, the formulas of HML is defined by the following grammar

$$
\begin{array}{rcll}
A, B & ::= & \langle\alpha\rangle A & \text{may modality} \\
& \mid & [\alpha] A & \text{must modality} \\
& \mid & A \wedge B & \text{conjunction} \\
& \mid & \neg A & \text{negation} \\
& \mid & \textbf{true} & \text{truth constant}
\end{array}
$$

The meaning of formulas is given by the satisfaction relation

$$P \vDash A$$

that is defined by the following

$P \vDash \langle\alpha\rangle A$	if *exists* P' such that $P \xrightarrow{\alpha} P'$ and $P' \vDash A$
$P \vDash [\alpha]A$	if *for all* P' such that $P \xrightarrow{\alpha} P'$ then $P' \vDash A$
$P \vDash A \wedge B$	if $P \vDash A$ *and* $P \vDash B$
$P \vDash \neg A$	if it is *not* the case $P \vDash A$
$P \vDash \mathbf{true}$	is always true for any P

The satisfaction relation asserts that the may modality $\langle\alpha\rangle A$ requires the process P to have a transition with the action α to P' and then the formula A must be satisfied by P'. Similarly, the must modality $[\alpha]A$ requires that all processes P' must satisfy A resulting from all the transition of P with the action α. Formally, only one modality is necessary to be defined as they are inter-definable, i.e., $\langle\alpha\rangle A = \neg[\alpha]\neg A$, and $[\alpha]A = \neg\langle\alpha\rangle\neg A$. Other logical formulas are obtained in the standard way: $\mathbf{false} = \neg\mathbf{true}$, $A \vee B = \neg(\neg A \wedge \neg B)$, and $A \implies B = \neg A \vee B$.

Now we can define the logical equivalence relation between two processes. Processes P and Q are logically equivalent

$$P \mathrel{\text{⫤}\!\vDash} Q$$

if for all formulas A, $P \vDash A$ if and only if $Q \vDash A$.

Let us recall examples from the previous section, Section 2.3. We have that

$$a.b.\mathbf{0} \;\; \mathrel{\text{⫤}\!\vDash} \;\; a.b.\mathbf{0} + a.b.\mathbf{0}$$

as a formula A does *not* exist such that $a.b.\mathbf{0} \vDash A$ and $a.b.\mathbf{0} + a.b.\mathbf{0} \nvDash A$. However, the following are not logically equivalent

$$a.b.\mathbf{0} + a.c.\mathbf{0} \;\; \not\mathrel{\text{⫤}\!\vDash} \;\; a.(b.\mathbf{0} + c.\mathbf{0})$$

since we can find a distinguishing formula, namely $\langle a\rangle(\langle b\rangle\mathbf{true} \wedge \langle c\rangle\mathbf{true})$, such that

$$a.(b.\mathbf{0} + c.\mathbf{0}) \vDash \langle a\rangle(\langle b\rangle\mathbf{true} \wedge \langle c\rangle\mathbf{true})$$

but

$$a.b.\mathbf{0} + a.c.\mathbf{0} \nvDash \langle a\rangle(\langle b\rangle\mathbf{true} \wedge \langle c\rangle\mathbf{true}).$$

We say that a labelled transition system is *image-finite* if for all processes P and actions α the set of continuation processes $\{P' : P \xrightarrow{\alpha} P'\}$ is finite. For an image-finite labelled transition system, the above logic is adequate, or the logical relation characterises the bisimilarity relation, meaning that for any P and Q, $P \sim Q$ if and only if $P \mathrel{\text{⫤}\!\vDash} Q$. That is, the two relation coincide:

$$\sim \quad = \quad \mathrel{\text{⫤}\!\vDash}$$

To be able to characterise the bisimilarity relation in non-image-finite transition systems such as the pi-calculus and psi-calculi, the logic typically is

extended with an infinite conjunction formula that is powerful enough to enumerate the infinite number of transitions of these systems. The infinite conjunction can then be defined to be

$$\bigwedge_{i \in I} A_i$$

where each A_i is a formula and I is an indexing set that may be infinite. The meaning is then simply

$$P \vDash \bigwedge_{i \in I} A_i \quad \text{if } P \vDash A_i \text{ for all } i \in I.$$

HML as presented here is purposely minimal; it contains the formulas needed for stating the adequacy property. Typically, HML variants would have other formulas, for example, state predicates that hold for particular processes. The adequacy result is then used as a test for extensions of HML in order to get a logic that asserts properties of processes that are compatible with behavioural equivalence reasoning.

We have developed an extension of HML for nominal transition systems that we discuss in Paper I.

2.5 Session Types

Behavioural types are types that themselves can be seen as processes, or rather abstractions of a processes. Behavioural types are used as an abstract specification for distributed concurrent system protocols. The type checking of a process is then a method of checking that an implementation (i.e., a process) conforms to a specification (i.e., a type). Session types are perhaps the most prominent example of behavioural types. They typically ensure not only the correspondence between protocol specification and implementation but also properties guaranteed by type safety like the absence of deadlock due to communication mismatch. Here we present a version of the original session type system that is now known as *binary* session types introduced by Honda et al. [18].

The idea of binary session types is to describe the reciprocal behaviour of communication between two processes on a private session channel. Consider the following system expressed in a pi-calculus like language:

$$(\nu s)(P \mid Q) \mid R$$

where R represent the rest of the parallel components of the system; note that the scope of the restriction s does not extend to R. In order for P and Q to proceed on the session channel s, either P is capable of sending and Q is

capable of receiving on s, or vice versa. For example, the following is a safe interaction resulting in a communication on channel s:

$$(\nu s)(s(x).P' \mid \bar{s}m.Q') \mid R \quad \rightarrow \quad (\nu s)(P\{m/x\} \mid Q) \mid R$$

but the following results in communication mismatch on s (i.e. deadlock):

$$(\nu s)(\bar{s}m.P' \mid \bar{s}n.Q') \mid R.$$

The processes P' and Q' need to have the same property that all their communication operations are reciprocal. One abstracts the communicated messages to just their type and the operators to their capabilities of input or output.

With session types, we can describe the safe interactions within a session. The session s is assigned a session type that describes the interactions from the perspective of one session participant, and the type of the other participant can be derived by dualising all of the interactions.

For the above safe example, we can assign to s the type $\mathtt{int}.T$, written $s : \mathtt{int}.T$. The type says: receive an integer and continue as T. So, the process $s(x).P'$ is well-typed according to the type assignment where $x : \mathtt{int}$ and P' is well-typed with respect to T. The partner process $\bar{s}m.Q'$ is well-typed with $\overline{\mathtt{int}}.T'$ where $m : \mathtt{int}$ and Q' is well-typed with regard to $s : T'$. The type describes the output of an integer and continuation with the type T'. The two types $\mathtt{int}.T$ and $\overline{\mathtt{int}}.T'$ describe interactions on a opposite endpoints of the session channel. They are dual in the sense that the capabilities of input are flipped to output, and vice versa. Thus, we only need one type to obtain the type of the other participant. So, the channel s in the unsafe example above has no session type, and thus it is prevented statically by the session type system.

Let us be more formal and introduce the basic language that is typed using session types. We distinguish between several kinds of names in the calculus. The name s denotes a session channel, the name a denotes shared channels, and the name x denotes a variable. We use m to denote some data that are not names. Let $\mathscr{L}$ be a set of labels, ranged over by ℓ. Then, the calculus is defined by the following which is a version of the pi-calculus (cf. Section 2.1.1 and *Figure 2.2*)

$$\begin{array}{rcl} P, Q & ::= & a(s).P \;\big|\; \bar{a}(s).P \;\big|\; s(x).P \;\big|\; \bar{s}m.P \;\big|\; P \mid Q \;\big|\; \textbf{if } \varphi \textbf{ then } P \textbf{ else } Q \\ & \big| & A \;\big|\; (\nu s)P \;\big|\; (\nu a)P \;\big|\; \mathbf{0} \\ & \big| & s \oplus \ell.P \;\big|\; s\&\{\ell_1 : P_1, \ldots, \ell_n : P_n\} \end{array}$$

The operators are mostly as they are in the pi-calculus. The input and output operators are duplicated because they operate on different data. The first two operators are called accept and request of a session s. They have slightly different semantics from the usual communication as they describe the following interaction

$$a(s).P \mid \bar{a}(s).Q \rightarrow (\nu s)(P \mid Q)$$

where accept and request establish a private sessions s between two processes. Thus, the shared channels act as an interface for establishing sessions.

The branching operator $s\&\{\ell_1 : P_1, \ldots, \ell_n : P_n\}$ is a specialisation of the non deterministic choice that we have seen in Section 2.1.1. The branching contains labels for the possible choices, allowing an external process to trigger a branch with the selection operator $s{\oplus}\ell.P$. We can describe it as

$$s{\oplus}\ell.P \mid s\&\{\ell_1 : P_1, \ldots, \ell_n : P_n\} \rightarrow P \mid P_i$$

where $\ell = \ell_i$ for some $i = 1, \ldots, n$. So $s{\oplus}\ell.P$ forces branching on its parallel component on label ℓ via session s. The process A is just a process constant (Section 2.1.1).

Let $\mathscr{B}$ be a set of data types (base types) that, for example, may include types for integers, strings, and similar. Then, the binary session types are defined as follows[7]

$$\begin{array}{rcll}
T & ::= & \beta.T & \text{input} \\
& \mid & \overline{\beta}.T & \text{output} \\
& \mid & \textbf{end} & \text{inaction type} \\
& \mid & \oplus\{\ell_1 : T_1, \ldots, \ell_n : T_n\} & \text{selection} \\
& \mid & \&\{\ell_1 : T_1, \ldots, \ell_n : T_n\} & \text{branching} \\
& \mid & \mu t.T & \text{recursion} \\
& \mid & t & \text{type variable}
\end{array}$$

Note the similarity of the type language with CCS of Section 2.1.1, except that this language is not interpreted operationally but as a type. The input and output types are as described above. The **end** type merely signifies the termination of the type. The selection type describes the selections that the process may perform, and similarly the branching type describes the branches that the process can take. The recursion type and the type variable allows for describing unbounded behaviours of the process.

The duality of types is defined as follows where note that the capabilities (input, output, selection, and branching) are reversed.

$$\begin{array}{rcl}
\mathsf{dual}(\beta.T) & = & \overline{\beta}.\mathsf{dual}(T) \\
\mathsf{dual}(\overline{\beta}.T) & = & \beta.\mathsf{dual}(T) \\
\mathsf{dual}(\textbf{end}) & = & \textbf{end} \\
\mathsf{dual}(\oplus\{\ell_1 : T_1, \ldots, \ell_n : T_n\}) & = & \&\{\ell_1 : \mathsf{dual}(T_1), \ldots, \ell_n : \mathsf{dual}(T_n)\} \\
\mathsf{dual}(\&\{\ell_1 : T_1, \ldots, \ell_n : T_n\}) & = & \oplus\{\ell_1 : \mathsf{dual}(T_1), \ldots, \ell_n : \mathsf{dual}(T_n)\} \\
\mathsf{dual}(\mu t.T) & = & \mu t.\mathsf{dual}(T) \\
\mathsf{dual}(t) & = & t
\end{array}$$

[7] It is the subset of [18] with the types for session delegation omitted.

We assign types to shared channels as $a : T$. Then, the typing rules assign the dual types for the accept and request session channels. The request process $\overline{a}\, s.P$ is typed with the assignment $s : T$ where T is inherited from $a : T$, while the accept process $a(s).P$ is typed with the dual type assignment $s : \mathsf{dual}(T)$, where again T is inherited from $a : T$. When typing already established sessions, one needs to be careful to give the right type and dual type to the parallel components.

Formally, assigning types is defined as a relation of the form

$$\Gamma; \Delta \vdash P$$

where Γ is a list of type assignments to the shared channels, and Δ is the type assignments to the session channels.

The soundness of the type system is demonstrated by establishing the subject reduction property. Subject reduction states if $\Gamma; \Delta \vdash P$ and $P \rightarrow P'$ then there is Δ' such that $\Gamma; \Delta' \vdash P'$. This means that a well-typed process reduces to a well-typed process. In a well-typed process, some bad behaviour is absent such as the communication mismatch that we alluded in the example above. Thus, typing also ensures the safety of communication, referred to as type-safety.

Paper III adapts the standard binary session types to systems with unreliable and broadcast communication.

3. Summary of Contributions

3.1 Paper I: Modal Logics for Nominal Transition Systems

Often in more advanced process calculi the labelled transition system would contain names that also bind into the derivative. Furthermore, transitions would adhere to a principle that the choice of these names is immaterial if they are sufficiently fresh for the derivative. Examples of such systems include the pi-calculus and psi-calculi that we have introduced in the sections Section 2.1.1 and Section 2.1.2. In Paper I, we have introduced a notion of labelled transition system and logic to canonically reason about them.

In Paper I, we introduce a general notion of a nominal labelled transition system with labels whose names may bind into the derivative. Formally, we generalise the standard labelled transition system (see Section 2.2) to include state predicates and a labelling relation on states and state predicates. We add structure to actions by requiring a function that denotes the *binding names* of an action. Finally, the transition relation is required to be invariant under the consistent renaming of the binding names in the action and the derivative.

We define a notion of bisimulation relation for a nominal transition system. The definition of bisimulation is standard except for the addition of the clause of static implication which ensures that the related states enable the same state predicates, and a refinement of the simulation clause with the requirement that binding names are chosen fresh for the related state (as it is done in the pi-calculus, and psi-calculi).

Finally, we define an infinitary Hennessy-Milner logic (Section 2.4) for nominal transition systems. The difference to the standard HML (Section 2.4) is that the action α in the formula $\langle\alpha\rangle A$ may contain binding names that bind into the formula A and that the formulas are required to be finitely supported (roughly, to have a finite set of free names). We show that this logic is adequate for bisimilarity relation. The novel construct in our logic is the infinitary finitely-supported conjunction. Significantly, it allows us to express formulas with quantification, in particular, we can quantify over names, i.e. $\forall n \in \mathscr{N}.A(n)$ as $\bigwedge_{n\in\mathscr{N}} A(n)$ that was not possible with previous HML logics that required uniformly bounded formulas for each member of the infinitary conjunction.

Adequate variants of our logic are defined for various notions of bisimilarity that have been introduced over the years. In particular, we can capture early bisimilarity, early congruence, late bisimilarity and equivalence, open

bisimilarity and hyperbisimilarity. We also introduce a variant of our logic to characterise weak bisimilarity.

Our logic is expressive: many standard modal logic and HML connectives are definable in our logic. We show that the least fixpoint operator (allowing recursive logical formulas) is definable in our logic. The next-step operator is also definable in our logic. Thus, we can express formulas of standard branching time logics like CTL.

We demonstrate the expressiveness of our logic, by instantiating it to provide an adequate HML for CCS, the pi-calculus, the spi-calculus, the applied pi-calculus, the fusion calculus, the concurrent constraint pi-calculus, and psi-calculi.

The main results have been formalised and proved in the theorem prover Isabelle, giving us significant trust in the correctness of the results. In particular, we have formalised adequacy of the logic, and adequacy of the variant of logic.

3.1.1 Comments on My Participation

I participated in exploring the design space by developing a less minimal, more concrete Hennesy-Milner logic with adequacy results for psi-calculi.

In the paper, I have introduced the encoding of the least fixpoint operator. I have initially introduced the formula valuation in sets of states and proved that the valuation of least fixpoint operator is indeed the least fixpoint in sets of states. Later I collaborated with my coauthors to refine the proofs and definitions, who found problems with the encodings.

I had contributed text on the fixpoint operator and as well some text on the derived operators section.

3.2 Paper II: A Sorted Semantic Framework for Applied Process Calculi

The theory of psi-calculi (Section 2.1.2) is untyped. This becomes cumbersome when expressing more complex data in psi-calculi. Since any name may be used as a variable and the substitution function needs to be a total function, the set of terms need to include terms that result from substituting variables for any other terms, even when the terms are meaningless. All substitutions need to be accounted as substitutions in psi-calculi arise in the input process.

In Paper II, we introduce a sort system for the psi-calculi where we move away from any substitutions to only well-sorted substitutions by giving sorts to the names, terms, and patterns. The sort not only solves the problem but also gives additional expressive power. We also generalise the input process and input rule to allow more general pattern matching that allows arbitrary computation.

We generalise the pattern matching performed by the input rule. Formally, we introduce another parameter set for patterns that we denote by **X**. The input process now is defined to be $M(\lambda x_1,\dots,x_n)X.P$ where M is a term, but now X is a pattern drawn from **X**. We also distinguish names that occur in pattern X between names as data and names that are pattern variables bound by $x_1,\dots,x_n$. We do this by introducing the operation $\textsc{Vars}(X)$ to return a set of sets of pattern variables. The pattern matching is also parametrised with the operation $\textsc{Match}(M,(x_1,\dots,x_n),X)$ to return a set of list of terms that are assigned to the pattern variables $x_1,\dots,x_n$ by matching the pattern X against the term M. The MATCH function is allowed to return more than one possible match, allowing for non-deterministic behaviour in the input.

We found that this fine control of the input process is important for modelling security protocols. For example, $a(\lambda m,k)\mathsf{enc}(m,k).P$ is allowed in psi-calculi where the pattern $\mathsf{enc}(m,k)$ representing the cypher of the message m encrypted with the key k is decrypted by simply pattern matching. However, with fine control over the binding names we can disallow k from being a pattern variable in the pattern $\mathsf{enc}(m,k)$, and thus the only allowed input form is $a(\lambda m)\mathsf{enc}(m,k).P$ where now the meaning subtly changes to the decrypting of a message with the key k since the process must have the knowledge of k as it is free.

The sort system is also parametric. The sorts are given by defining the set of sorts S; the sort assigning function SORT that assigns a sort to terms, patterns, and names; and four capability relations: (1) capability to input a pattern of sort s via a channel of sort s', (2) capability to output a term of sort s via a channel of sort s', (3) capability of substituting a term of sort s for a name of sort s', (4) capability of being able to restrict a name of sort s. The input, output and restriction constructs are are required to respect the capability relations. We call processes that respect capability relations well-formed. The capabilities for output and input together with the MATCH function need also be compatible with the substitution capability relation.

The subject reduction property holds for any instance of our sort system: a well-formed process transitions to a well-formed process. We then go on to re-establish the main results for psi-calculi with the new input process and SOS rule, and the sort system. More specifically, we show that the resulting strong and weak congruences satisfy the usual structural congruence laws. We establish this result in the theorem prover Isabelle for the pattern matching extension, and for the sort system we do this manually for technical limitations of the package that we rely on. The manual check is simplified by reducing the sorted process calculi to a more manageable simpler form.

In order to relate encodings, we introduce a notion of representation of a process calculus by a psi-calculi instance. Representation is a map that we require to be homomorphic with regard to a context, and the transition systems operationally correspond with regard to this map up to a structural congruence. Representation is complete if a map is also surjective up to bisimulation. This

notion of encoding is quite stronger than the standard encoding criteria by Gorla [14].

The extended psi-calculi with generalised pattern matching and sorts is expressive and capable of presenting many well-known process calculi. The extended psi-calculi completely represents both unsorted and sorted polyadic pi-calculus. Also, the subcalculus with inputs of the process calculus LINDA falls out as a special case. We can also represent polyadic synchronisation pi-calculus and value-passing CCS.

3.2.1 Comments on My Participation

I contributed to encodings and representation proofs, and implemented the sort system in the PWB tool.

3.3 Paper III: A Session Type System for Unreliable Broadcast Communication

Session type systems rely on the reliability of communication, that is, no message loss is allowed, to ensure that the process follows the protocol specified by the session type. In Paper III, we forgo reliability of communication. We introduce a process calculus with unreliable synchronous broadcast and equip it with a sound binary session type system, meaning that the subject reduction property is established.

With the broadcast process calculus that we propose we capture many communication features found in ad-hoc and wireless sensor networks. The processes of the calculus are annotated with labels that represent locations, called nodes. Communication in the calculus operates with respect to a connectivity graph where edges denote connections between locations. The graph is arbitrary. However, the graph is static, meaning it does not evolve during the execution of the system. We capture two common operations in such networks that we call scatter and gather. Scatter is simply a broadcast to neighbouring nodes (one-to-many) with regard to the connectivity graph. Notably, the scattered data may be lost: it is not necessarily received by all, or indeed any, of the neighbouring nodes. Gather is an operation that aggregates received data from the neighbouring nodes (many-to-one). Again, not all of the data is received and aggregated due to message loss.

To cope with unreliability, we introduce a recovery process. The recovery mechanism is autonomous, that is, it occurs non-deterministically without any particular trigger. Thus, recovery does not involve communication. Furthermore, the nodes keep track of the session state they are in to ensure that communication can only occur if they are in the same protocol stage. The resulting reduction semantics for the broadcast calculus is quite straightforward.

Our session type system is based on the standard binary session types. In particular, the session endpoints of scatter and input are assigned dual session types, and similarly for gather and output. When typing the session channel we allow multiple copies of a type assignment to a session channel, as there may be multiple nodes following the same protocol. Also, when typing nodes, the context containing sessions is synchronised with the session state in the nodes.

The subject reduction property holds for our system. That is, if a well-typed network reduces, then it is still well-typed. However, a reduced network may make use of fewer session channels due to some nodes recovering. Furthermore, we formalise a type-safety property such that type-safe nodes in the same session state have the dual communication capabilities. In our system, a progression of a network is always guaranteed as we may always lose messages; so in our system, progress occurs via communication. We also demonstrate that we can give a type to a standard data aggregation algorithm in wireless sensor networks.

3.3.1 Comments on My Participation

I am the principal author of this paper. The idea to use session types in systems with unreliable broadcast is mine. I introduced the calculus and typing rules, however, the final form is certainly a product of collaboration. I have done the proofs and wrote most of the paper.

3.4 Paper IV: The Psi-Calculi Workbench: A Generic Tool for Applied Process Calculi

In Paper IV, we present a tool for modelling concurrent systems called the Psi-calculi workbench (PWB). The tool accepts psi-calculi as the modelling language for processes. The tool implements an interactive command interpreter for inputting processes and interacting with various sub-tools. Most notably, we implement a execution for psi-calculi processes and a bisimulation generator for given processes. Both of these are also provided with the weak transition versions.

PWB implements a variant of psi-calculi. It includes both the usual synchronous point-to-point and unreliable synchronous broadcast communication. PWB extends psi-calculi with process constant definitions and process constant invocations for the convenience of developing large models. Furthermore, PWB implements the sorts extension of Paper II. However, there is no pattern matching in PWB, although polyadic communication is allowed. The weak symbolic semantics and bisimulation generation algorithm require the weakening of assertions to hold, that is, conditions that are entailed by an assertion are also entailed by all of the extensions of that assertion.

In the paper, we introduce the symbolic structural operational semantics for psi-calculi that include both point-to-point and unreliable broadcast communication. Symbolic semantics is a way of abstracting infinite behaviour of a process to a finite symbolic version coupled with finite formulas that characterise that behaviour. We show that the symbolic semantics correspond to the standard structural operational semantics of psi-calculi. PWB indeed implements the symbolic semantics.

PWB has a modular and parametric architecture. It is parametrised by the structures for defining the psi-calculi parameters and solvers for solving the symbolic transition formulas (constraints). By providing the appropriate parameters, one obtains a tool, for example, for the pi-calculus, broadcast pi-calculus, spi-calculus.

We show the utility of the tool with examples of instantiating PWB with parameters and showing examples of symbolic execution. We have implemented an instance and provided a model in psi-calculi for the alternating bit protocol. We also, to demonstrate the broadcasting capabilities of a tool, provided an instance with constraint solver and a model in psi-calculi for a simple data aggregation protocol in wireless sensor networks. We also show an example of the utility of psi-calculi assertions to model dynamic connectivity graphs. All of the examples use structured data and channels.

PWB is a useful tool for modelling distributed concurrent systems. It provides tools for experimenting and developing models of those systems. One can use the behavioural equational reasoning for showing properties of the models. The tool can also be used for implementing new verification techniques.

3.4.1 Comments on My Participation

I have implemented PWB with the exception of parts of the broadcasting extension. I have devised and implemented the examples. I contributed text to the section on the tool and examples.

4. Conclusion and Future Work

In this dissertation, we presented contributions in the psi-calculi process calculi framework, Hennessy-Milner logic for nominal transition systems, and behavioural types for systems with unreliable communication.

The psi-calculi framework reduces the effort for defining new process calculi that share common traits such as shared channels, synchronous point-to-point communication, structured data, logical environment, pi-calculus like syntax and semantics. To obtain a new calculus with a bisimulation theory, one needs to instantiate a handful of parameters that must satisfy fairly straightforward requirements. It has been shown that many process calculi are captured by the psi-calculi [5], e.g., the pi-calculus, CCS, the concurrent constraint calculus and others. In this work, we have extended the expressiveness and generality of the psi-calculi even further by equipping the psi-calculi with a simple type system and generalising the pattern matching mechanism in the input rule to arbitrary and non-deterministic computation. This allowed us to directly capture the value passing CCS, the sorted and unsorted polyadic pi-calculus, and the polyadic synchronisation pi-calculus. The correspondence between the psi-calculi and the mentioned calculi are much stronger than the now standard Gorla's expressiveness criteria [14]. The sorts and pattern matching give powerful yet simple tools for developing new process calculi theory.

Furthermore, in this thesis, we developed a tool for the psi-calculi framework called the psi-calculi workbench (PWB). The tool is both a software library and an interactive command line tool. A user is capable of extending the tool with new process calculi by implementing the parameters of psi-calculi and constraint solvers that drive the symbolic execution and bisimulation generation modules. Thus, an implementer of a calculus in PWB can also distribute a derived tool just for a specific process calculus to other users. Therefore, the derived tool ships with an interactive symbolic execution interface of a process and a bisimulation relation generator with far less effort than developing a new tool from scratch. PWB makes it simpler to experiment with new definitions of process calculi and also explore the specification of concurrent systems in those calculi.

Modal logic is a way of specifying properties and verifying systems abstractly. We have developed a Hennessy-Milner modal logic (HML) for transition systems that make use of binding names in their actions, which we call nominal transition systems. The process calculi literature is rich with these kinds of systems, chief among them the pi-calculus and its many derivatives. Our logic is infinitary; namely, the conjunction operator is constructed from

an infinite set of formulas. The innovative feature in our logic is that we do not require the formation of an infinite conjunction to have a uniformly bounded finite support (free names of the formula), but require the set of formulas of infinite conjunction to have a finite support. Crucially, this feature still allows us to have well-defined alpha-conversion of formulas. We obtain, by considering such formulas, a significant expressive power over other Hennessy-Milner logics for transition systems: in our logic, we are able to define quantification over names, and the fresh quantifier found in nominal logics (e.g. [26]). In our logic, we are also capable of encoding fixpoint formulas as derived operators. By having such general HML, we can provide an adequate logic for many systems that no logical system has been considered and capture many that have. For example, our logic is adequate (in the sense of Section 2.4) for the pi-calculus and the psi-calculi framework.

Session type systems are prominent specification language for distributed system protocols. In this thesis, we have adapted a standard binary session type system to a new setting: we have devised a typing relation for process calculus with unreliable broadcasting communication. The system is novel in the sense that up to now there were no session type systems for process calculi that have unreliable communication semantics.

Future work

There is a lot of work that remains to be done.

Sorts are a simple and straightforward way of modelling data invariants in the sorted psi-calculi. However, more advanced typing system should be considered for the psi-calculi, for instance, binary session type system. Hüttel has devised several type systems for psi [19, 20], however, his systems do not consider psi-calculi in full generality and have quite complicated conditions. A major challenge with adapting more advanced type systems for psi-calculi is the non-monotonic logic of the assertion environments. For example, a psi-calculi process can easily disable the session channel connectivity after a transition and thus diverge from the session protocol. One could consider monotonic logics for psi-calculi only; or, treat the session channels specially in the way that they are not affected by the assertions; or, one could also consider some type of dependency of assertion environment in type judgements. Thus, there is a non-trivial design space to explore to arrive at a satisfactory type system for psi-calculi.

The symbolic execution of PWB is based on the symbolic semantics for psi-calculi where the idea is to abstract the values to make the transitions finite. There is an established correspondence between standard psi-calculi semantics and symbolic semantics. However, the semantics differ in significant ways. For one, it does not cover the full psi-calculi. Specifically, it is not obvious how to extend the current symbolic semantics to handle the general pattern matching of the psi-calculi input process of Paper II. Furthermore, the abstracted values and names in processes are conflated. This puts restrictions

on the psi-calculi parameters that are not present in the original. Devising symbolic semantics, in general, can be an arduous and ad-hoc process. Instead, we can consider a general format for SOS rules and derive sound and complete symbolic semantics for such rules. This idea looks promising, and we already started exploring such rule formats (Paper VII).

For the Hennessy-Milner logic for nominal transition systems, we envision a sound and complete proof system. The challenge here is providing proof rules for the infinitary conjunction with finite support and modalities with binding names. A development of a finitary subset of our logic and a model checking algorithm would be beneficial for applications.

The broadcast process calculus and binary session type system are just the first steps towards applying session type to unreliable systems. There are shortcomings to our system. The biggest is the recovery system is too strong. We would like to investigate other forms of recovery, e.g., exceptions. However, this is not entirely obvious how to achieve this without communicating that an exception has occurred to other participants. The system would benefit from an extension to multiparty session types and choreographies. Choreographies, in particular, for the unreliable broadcast systems seem quite removed from standard choreographies. It seems there is a need for the notion of neighbourhood which counter-intuitively is a notion of locality to particular nodes. Communication is synchronous in our system, which may be not fitting certain classes of applications. We would like to reformulate our system to use asynchronous communication.

References

[1] Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. In *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '01, pages 104–115, New York, NY, USA, 2001. ACM.

[2] Martín Abadi and Andrew D. Gordon. Reasoning about cryptographic protocols in the spi calculus. In *Proceedings of the 8th International Conference on Concurrency Theory*, CONCUR '97, pages 59–73, London, UK, 1997. Springer-Verlag.

[3] Jos C. M. Baeten. Process Algebra A brief history of process algebra. *Theoretical Computer Science*, 335(2):131–146, 2005.

[4] Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. Studies in Logic and the Foundations of Mathematics. Elsevier Science, 2013.

[5] Jesper Bengtson, Magnus Johansson, Joachim Parrow, and Björn Victor. Psi-calculi: a framework for mobile processes with nominal data and logic. *Logical Methods in Computer Science*, 7(1):1–44, 2011.

[6] Gerard Berry and Gerard Boudol. The Chemical Abstract Machine. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, pages 81–94, New York, NY, USA, 1990. ACM.

[7] Johannes Borgström, Shuqin Huang, Magnus Johansson, Palle Raabjerg, Björn Victor, Johannes Åman Pohjola, and Joachim Parrow. Broadcast psi-calculi with an application to wireless protocols. *Software and Systems Modeling*, 14(1):201–216, 2013.

[8] Gérard Boudol. Asynchrony and the Pi-calculus. Rapport de recherche RR-1702, INRIA, 1992.

[9] Maria Grazia Buscemi and Ugo Montanari. CC-Pi: A Constraint-Based Language for Specifying Service Level Agreements. In Rocco de Nicola, editor, *Programming Languages and Systems*, volume 4421 of *Lecture Notes in Computer Science*, pages 18–32. Springer Berlin Heidelberg, 2007.

[10] Luca Cardelli and Andrew D. Gordon. Mobile ambients. *Electronic Notes in Theoretical Computer Science*, 10:198–201, 1997.

[11] Christian Ene and Traian Muntean. A Broadcast-based Calculus for Communicating Systems. *Parallel and Distributed Processing Symposium, International*, 3:30149b, 2001.

[12] Marcelo Fiore, Eugenio Moggi, and Davide Sangiorgi. A Fully Abstract Model for the pi-calculus. *Information and Computation*, 179(1):76–117, 2002.

[13] Murdoch Gabbay and Andrew Pitts. A New Approach to Abstract Syntax Involving Binders. In *Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science*, LICS '99, Washington, DC, USA, 1999. IEEE Computer Society.

[14] Daniele Gorla. Towards a unified approach to encodability and separation results for process calculi. *Information and Computation*, 208(9):1031–1053, 2010.
[15] Matthew Hennessy. A fully abstract denotational semantics for the π-calculus. *Theoretical Computer Science*, 278(1–2):53–89, 2002.
[16] Matthew Hennessy and Robin Milner. On observing nondeterminism and concurrency. In Jaco Bakker and Jan Leeuwen, editors, *Automata, Languages and Programming*, volume 85 of *Lecture Notes in Computer Science*, pages 299–309. Springer Berlin Heidelberg, 1980.
[17] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In Pierre America, editor, *ECOOP'91 European Conference on Object-Oriented Programming*, volume 512 of *Lecture Notes in Computer Science*, pages 133–147. Springer Berlin Heidelberg, 1991.
[18] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In Chris Hankin, editor, *Programming Languages and Systems*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer Berlin Heidelberg, 1998.
[19] Hans Hüttel. Typed Psi-calculi. In Joost-Pieter Katoen and Barbara König, editors, *CONCUR 2011 – Concurrency Theory*, volume 6901 of *Lecture Notes in Computer Science*, pages 265–279. Springer Verlag, 2011.
[20] Hans Hüttel. Types for Resources in Psi-calculi. In Martín Abadi and Alberto Lluch Lafuente, editors, *Trustworthy Global Computing*, volume 8358 LNCS of *Lecture Notes in Computer Science*, pages 83–102. Springer International Publishing, 2014.
[21] Robin Milner. *A Calculus of Communicating Systems*. Springer-Verlag, Secaucus, NJ, USA, 1980.
[22] Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes, I. *Information and Computation*, 100(1):1–40, 1992.
[23] Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes, II. *Information and Computation*, 100(1):41–77, 1992.
[24] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How Amazon Web Services Uses Formal Methods. *Communications of the ACM*, 58(4):66–73, 2015.
[25] Joachim Parrow, Johannes Borgström, Palle Raabjerg, and Johannes Åman Pohjola. Higher-order psi-calculi. *Mathematical Structures in Computer Science*, 24(02):1–37, 2013.
[26] Andrew M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 186(2):165–193, 2003.
[27] Andrew M. Pitts. *Nominal Sets: Names and Symmetry in Computer Science*, volume 57 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2013.
[28] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, 1981.
[29] Viggo Stoltenberg-Hansen, Ingrid Lindström, and Edward R. Griffor. *Mathematical Theory of Domains*. Cambridge University Press, New York, NY, USA, 1994.

Acta Universitatis Upsaliensis

Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 1392

Editor: The Dean of the Faculty of Science and Technology

A doctoral dissertation from the Faculty of Science and Technology, Uppsala University, is usually a summary of a number of papers. A few copies of the complete dissertation are kept at major Swedish research libraries, while the summary alone is distributed internationally through the series Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology. (Prior to January, 2005, the series was published under the title "Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology".)

ACTA
UNIVERSITATIS
UPSALIENSIS
UPPSALA
2016

Distribution: publications.uu.se
urn:nbn:se:uu:diva-300029

Paper I

MODAL LOGICS FOR NOMINAL TRANSITION SYSTEMS

JOACHIM PARROW, JOHANNES BORGSTRÖM, LARS-HENRIK ERIKSSON, RAMŪNAS GUTKOVAS, TJARK WEBER

Uppsala University, Sweden

Abstract. We define a uniform semantic substrate for a wide variety of process calculi where states and action labels can be from arbitrary nominal sets. A Hennessy-Milner logic for these systems is introduced, and proved adequate for bisimulation equivalence. A main novelty is the use of finitely supported infinite conjunctions. We show how to treat different bisimulation variants such as early, late, open and weak in a systematic way, and make substantial comparisons with related work. The main definitions and theorems have been formalized in Nominal Isabelle.

1. Introduction

Transition systems. Transition systems are ubiquitous in models of computing, and specifications to say what may and must happen during executions are often formulated in a modal logic. There is a plethora of different versions of both transition systems and logics, including a variety of higher-level constructs such as updatable data structures, new name generation, alias generation, dynamic topologies for parallel components etc. In this paper we formulate a general framework where such aspects can be treated uniformly, and define accompanying modal logics which are adequate for bisimulation. This is related to, but independent of, our earlier work on psi-calculi [BJPV11], which proposes a particular syntax for defining behaviours. The present paper does not depend on any such language, and provides general results for a large class of transition systems.

In any transition system there is a set of *states* $P, Q, \ldots$ representing the configurations a system can reach, and a relation telling how a computation can move between them. Many formalisms, for example all process algebras, define languages for expressing states, but in the present paper we shall make no assumptions about any such syntax.

In systems describing communicating parallel processes the transitions are labelled with *actions* α, β, representing the externally observable effect of the transition. A transition $P \xrightarrow{\alpha} P'$ thus says that in state P the execution can progress to P' while conducting the action α, which is visible to the rest of the world. For example, in CCS these actions are atomic and partitioned into output and input communications. In value-passing calculi the actions can be more complicated, consisting of a channel designation and a value from some data structure to be sent along that channel.

Scope openings. With the advent of the pi-calculus [MPW92] an important aspect of transitions was introduced: that of name generation and scope opening. The main idea is

LOGICAL METHODS IN COMPUTER SCIENCE

DOI:10.2168/LMCS-???

© Parrow, Borgström, Eriksson, Gutkovas, Weber
Creative Commons

that names (i.e., atomic identifiers) can be scoped to represent local resources. They can also be transmitted in actions, to give a parallel entity access to this resource. In the monadic pi-calculus such an action is written $\overline{a}(\nu b)$, to mean that the local name b is exported along the channel a. These names can be subjected to alpha-conversion: if $P \xrightarrow{\overline{a}(\nu b)} P'$ and c is a fresh name then also $P \xrightarrow{\overline{a}(\nu c)} P'\{c/b\}$, where $P'\{c/b\}$ is P' with all bs replaced by cs. Making this idea fully formal is not entirely trivial and many papers gloss over it. In the polyadic pi-calculus several names can be exported in one action, and in psi-calculi arbitrary data structures may contain local names. In this paper we make no assumptions about how actions are expressed, and just assume that for any action α there is a finite set of names $\mathrm{bn}(\alpha)$, the *binding names*, representing exported names. In our formalization we use nominal sets, an attractive theory to reason about objects depending on names on a high level and in a fully rigorous way.

State predicates. The final general components of our transition systems are the *state predicates* ranged over by φ, representing what can be concluded in a given state. For example state predicates can be equality tests of expressions, or connectivity between communication channels. We write $P \vdash \varphi$ to mean that in state P the state predicate φ holds.

A structure with states, transitions, and state predicates as discussed above we call a *nominal transition system*.

Hennessy-Milner Logic. Modal logic has been used since the 1970s to describe how facts evolve through computation. We use the popular and general branching time logic known as Hennessy-Milner Logic [HM85] (HML). Here the idea is that an action modality $\langle\alpha\rangle$ expresses a possibility to perform an action α. If A is a formula then $\langle\alpha\rangle A$ says that it is possible to perform α and reach a state where A holds. With conjunction and negation this gives a powerful logic shown to be *adequate* for bisimulation equivalence: two processes satisfy the same formulas exactly if they are bisimilar. In the general case, conjunction must take an infinite number of operands when the transition systems have states with an infinite number of outgoing transitions. The fully formal treatment of this requires care in ensuring that such infinite conjunctions do not exhaust all names, leaving none available for alpha-conversion. All previous works that have considered this issue properly have used uniformly bounded conjunction, i.e., the set of all names in all conjuncts is finite.

Contributions. Our definition of nominal transition systems is very general since we leave open what the states, transitions and predicates are. The only requirement is that transitions satisfy alpha-conversion. A technically important point is that we do not assume the usual *name preservation principle*, that if $P \xrightarrow{\alpha} P'$ then the names occurring in P' must be a subset of those occurring in P and α. This means that the results are applicable to a wide range of calculi. For example, the pi-calculus represents a trivial instance where there are no state predicates. CCS represent an even more trivial instance where bn always returns the empty set. In the fusion calculus and the applied pi-calculus the state contains an environmental part which tells what expressions are equal to what. In the general framework of psi-calculi the states are processes with assertions describing their environments.

We define a modal logic with the $\langle\alpha\rangle$ operator that binds the names in $\mathrm{bn}(\alpha)$, and contains operators for state predicates. Instead of uniformly bounded conjunction we use the notion of finite support from nominal sets: a conjunction of an infinite set of formulas is

admissible if the set has finite support. This results in greater generality and expressiveness. For example, we can now define quantifiers and the next step modalities as derived operators. The main technical difficulty is to ensure that formulas and their alpha-equivalence classes throughout are finitely supported, i.e., only depend on a finite set of names, even in the presence of infinite conjunction.

We establish that logical equivalence coincides with bisimilarity. Compared to previous such adequacy results our proof takes a new twist. We also show how variants of the logic correspond to late, open and hyperbisimilarity in a uniform way. Such logics have to some extent been proposed before in ad-hoc ways; our contribution here is a systematic framework where one adequacy proof covers all of them.

We obtain an adequacy result for weak bisimulation through the weak modality $\langle\langle\alpha\rangle\rangle$, to represent a sequence of actions with observable content α. Weak bisimulation for nominal transition systems with state predicates is notoriously difficult, and we need an extra logical operator Φ **also** A to express that through unobservable actions we can reach a state satisfying both the potentially infinite but finitely supported set of state predicates Φ and the formula A.

Finally we compare our logic to several other proposed logics for CCS and developments of the pi-calculus. A conclusion is that we can easily represent most of them. The correspondence is not exact because of our slightly different treatment of conjunction, but we certainly gain simplicity and robustness in otherwise complicated logics. We also show how our framework can be applied to obtain logics where none have been suggested previously.

Formalization. Our main definitions and theorems have been formalized in Nominal Isabelle [UK12]. This has required significant new ideas to represent data types with infinitary constructors like infinite conjunction and their alpha-equivalence classes. As a result we corrected several details in our formulations and proofs, and now have very high confidence in their correctness. The formalization effort has been substantial, but certainly less than half of the total effort, and we consider it a very worthwhile investment.

Exposition. In the following section we provide the necessary background on nominal sets. In Section 3 we present our main definitions and results on nominal transition systems and modal logics. In Section 4 we derive useful operators such as quantifiers and fixpoints, and indicate some practical uses. Section 5 shows how to treat variants of bisimilarity such as late and open in a uniform way, and in Section 6 we treat a logic for weak bisimilarity. In Section 7 we compare with related work and demonstrate how our framework can be applied to recover earlier results uniformly. Finally Section 8 concludes with some remarks on the formalization in Nominal Isabelle.

This paper is an extended version of [PBE+15]. The present paper contains a new section on logics for weak bisimilarity as well as more explanations, examples and proofs.

2. Background on nominal sets

Nominal sets [Pit13] is a general theory of objects which depend on names, and in particular formulates the notion of alpha-equivalence when names can be bound. The reader need not know nominal set theory to follow this paper, but some key definitions will make it easier to appreciate our work and we recapitulate them here.

We assume an infinitely countable multi-sorted set of atomic identifiers or *names* $\mathcal{N}$ ranged over by $a, b, \ldots$. A *permutation* is a bijection on names that leaves all but finitely many names invariant. The singleton permutation which swaps names a and b and has no other effect is written $(a\,b)$, and the identity permutation that swaps nothing is written id. Permutations are ranged over by π, π'. The effect of applying a permutation π to an object X is written $\pi \cdot X$. Formally, the permutation action $\cdot$ can be any operation that satisfies $\mathrm{id} \cdot X = X$ and $\pi \cdot (\pi' \cdot X) = (\pi \circ \pi') \cdot X$, but a reader may comfortably think of $\pi \cdot X$ as the object obtained by permuting all names in X according to π.

A set of names N *supports* an object X if for all π that leave all members of N invariant it holds $\pi \cdot X = X$. In other words, if N supports X then names outside N do not matter to X. If a finite set supports X then there is a unique minimal set supporting X, called the *support* of X, written $\mathrm{supp}(X)$, intuitively consisting of exactly the names that matter to X. As an example the set of names textually occurring in a datatype element is the support of that element, and the set of free names is the support of the alpha-equivalence class of the element. Note that in general, the support of a set is not the same as the union of the support of its members. An example is the set of all names; each element has itself as support, but the whole set has empty support since $\pi \cdot \mathcal{N} = \mathcal{N}$ for any π.

We write $a\#X$, pronounced "a is fresh for X", for $a \notin \mathrm{supp}(X)$. The intuition is that if $a\#X$ then X does not depend on a in the sense that a can be replaced with any fresh name without affecting X. If A is a set of names we write $A\#X$ for $\forall a \in A \,.\, a\#X$.

A *nominal set* S is a set with a permutation action such that $X \in S \Rightarrow \pi \cdot X \in S$, and where each member $X \in S$ has finite support. A main point is that then each member has infinitely many fresh names available for alpha-conversion. Similarly, a set of names N supports a function f on a nominal set if for all π that leave N invariant it holds $\pi \cdot f(X) = f(\pi \cdot X)$, and similarly for relations and functions of higher arity. Thus we extend the notion of support to finitely supported functions and relations as the minimal finite support, and can derive general theorems such as $\mathrm{supp}(f(X)) \subseteq \mathrm{supp}(f) \cup \mathrm{supp}(X)$.

An object that has empty support we call *equivariant*. For example, a unary function f is equivariant if $\pi \cdot f(X) = f(\pi \cdot X)$ for all π, X. The intuition is that an equivariant object does not treat any name special.

3. Nominal transition systems and Hennessy-Milner logic

Definition 1. A *nominal transition system* is characterized by the following

- STATES: A nominal set of *states* ranged over by P, Q.
- PRED: A nominal set of *state predicates* ranged over by φ.
- An equivariant binary relation $\vdash$ on STATES and PRED. We write $P \vdash \varphi$ to mean that in state P the state predicate φ holds.
- ACT: A nominal set of *actions* ranged over by α.
- An equivariant function bn from ACT to finite sets of names, which for each α returns a subset of $\mathrm{supp}(\alpha)$, called the *binding names*.
- An equivariant transition relation $\rightarrow$ on states and residuals. A residual is a pair of action and state. For $\rightarrow (P, (\alpha, P'))$ we write $P \xrightarrow{\alpha} P'$. The transition relation must satisfy alpha-conversion of residuals: If $a \in \mathrm{bn}(\alpha)$, $b\#\alpha, P'$ and $P \xrightarrow{\alpha} P'$ then also $P \xrightarrow{(a\,b)\cdot\alpha} (a\,b) \cdot P'$.

As an example, basic CCS from [Mil89] is a trivial nominal transition system. Here the STATES are the CCS agents, ACT the CCS actions, $\text{bn}(\alpha) = \emptyset$ for all actions, and PRED $= \emptyset$. For the pi-calculus, STATES are the pi-calculus agents, and ACT the four kinds of pi-calculus actions (silent, output, input, bound output). In the early semantics bn returns the empty set except for bound outputs where $\text{bn}(\overline{a}(\nu x)) = \{x\}$. In the late semantics there are actions like $a(x)$ where x is a placeholder so also $\text{bn}(a(x)) = \{x\}$. In the polyadic pi-calculus each action may bind a finite set of names.

In the original terminology of these and similar calculi these names are referred to as "bound". We believe a better terminology is "binding", since they bind into the target state. For higher-order calculi this distinction is important. Consider an example where objects transmitted in a communications are processes, and a communicated object contains a bound name:

$$P \xrightarrow{\overline{a}(\nu b)Q} R$$

The action here transmits the process $(\nu b)Q$ along the channel a. The name b is local to Q, so alpha-converting b to some new name affects only Q. Normally agents are considered up to alpha-equivalence, this means that b is not in the support of the action, and we have $\text{bn}(\overline{a}(\nu b)Q) = \emptyset$.

In the same calculus we may also have a different action

$$P \xrightarrow{(\nu b)\overline{a}Q} R$$

Again this transmits a process along the channel a, but the process here is just Q. The name b is shared between Q and R, and is extruded in the action. An alpha conversion of b thus affects both Q and R simultaneously. In the action b is a free name, in the sense that b is in its support, and it cannot be replaced by another name in the action alone. Here we have $\text{bn}((\nu b)\overline{a}Q) = \{b\}$.

In all of the above we have PRED $= \emptyset$ since communication is the only way a process may influence a parallel process, and thus communications are the only things that matter for process equivalence. More general examples come from psi-calculi [BJPV11] where there are so called "conditions" representing what holds in different states; those would then correspond to PRED. Other calculi, e.g. [WG05, BM07], also have mechanisms where processes can influence each other without explicit communication, such as fusions and updates of a constraint store. All of these are straightforward to accommodate as nominal transition systems. Section 8 contains further descriptions of these and other examples.

Definition 2. A *bisimulation* R is a symmetric binary relation on states in a nominal transition system satisfying the following two criteria: $R(P, Q)$ implies

(1) *Static implication*: $P \vdash \varphi$ implies $Q \vdash \varphi$.
(2) *Simulation*: For all α, P' such that $\text{bn}(\alpha)\#Q$ there exist Q' such that if $P \xrightarrow{\alpha} P'$ then $Q \xrightarrow{\alpha} Q'$ and $R(P', Q')$

We write $P \dot{\sim} Q$ to mean that there exists a bisimulation R such that $R(P, Q)$.

Static implication means that bisimilar states must satisfy the same state predicates; this is reasonable since these can be tested by an observer or parallel process. The simulation requirement is familiar from the pi-calculus. Note that this definition corresponds to "early" bisimulation in the pi-calculus. In Section 6 we will consider other variants of bisimilarity.

Proposition 1. $\dot{\sim}$ is an equivariant equivalence relation.

Proof: The proof has been formalized in Isabelle. Equivariance is a simple calculation, based on the observation that if R is a bisimulation, then $\pi \cdot R$ is a bisimulation. To prove reflexivity of $\dot{\sim}$, we note that equality is a bisimulation. Symmetry is immediate from Def. 2. To prove transitivity, we show that the composition of $\dot{\sim}$ with itself is a bisimulation; the simulation requirement is proved by a considering an alpha-variant of $P \xrightarrow{\alpha} P'$ where $\mathrm{bn}(\alpha)$ is fresh for Q. □

The minimal HML for nominal transition systems is the following.

Definition 3. The nominal set of formulas $\mathcal{A}$ ranged over by A is defined by induction as follows:

$$A \quad ::= \quad \bigwedge_{i \in I} A_i \quad | \quad \neg A \quad | \quad \varphi \quad | \quad \langle\alpha\rangle A$$

Support and name permutation are defined as usual (permutation distributes over all formula constructors). In $\bigwedge_{i\in I} A_i$ it is assumed that the indexing set I has bounded cardinality, by which we mean that $|I| \leq \kappa$ for some fixed infinite cardinal κ that is larger than the cardinality of STATES, ACT and PRED. It is also required that the set of conjuncts $\{A_i \mid i \in I\}$ has finite support; this is then the support of the conjunction. Alpha-equivalent formulas are identified; the only binding construct is in $\langle\alpha\rangle A$ where $\mathrm{bn}(\alpha)$ binds into A.

For a simple example related to the pi-calculus with the late semantics, consider the formula $\langle a(x)\rangle\langle\overline{b}x\rangle\top$ where $\top$ is the empty conjunction and thus always true. We have $\mathrm{bn}(a(x))) = \{x\}$ and therefore an alpha-variant of the formula is $\langle a(y)\rangle\langle\overline{b}y\rangle\top$. It says that it must be possible to input something along a and then output it along b. In the early semantics, where the input action contains the object received rather than a placeholder, the corresponding formula is

$$\bigwedge_{x\in\mathcal{N}} \langle ax\rangle\langle\overline{b}x\rangle\top$$

in other words, the conjunction is over the set of formulas $S = \{\langle ax\rangle\langle\overline{b}x\rangle\top \mid x \in \mathcal{N}\}$. This set has finite support, in fact the support is just $\{a, b\}$. The reason is that for $c, d\#\{a, b\}$ we have $(c\,d) \cdot S = S$. Note that this set has no finite common support, i.e., there is no finite set of names that supports all elements, and thus the conjunction is not expressible in the usual logics for the pi-calculus.

This example highlights one of the main novelties in Definition 3, that we use conjunction of a possibly infinite and finitely supported set of conjuncts. In comparison, the earliest HML for CCS, Hennessy and Milner (1985) [HM85], uses finite conjunction, meaning that the logic is adequate only for finite branching transition systems. In his subsequent book (1989) [Mil89] Milner admits arbitrary infinite conjunction, disregarding the danger of running into paradoxes. Abramsky (1991) [Abr91] employs a kind of uniformly bounded conjunction, with a finite set of names that supports all conjuncts, an idea that is also used in the first HML for the pi-calculus (1993) [MPW93]. All subsequent developments follow one of these three approaches. Our main point is that both finite and uniformly bounded conjunction are expressively weak, in that the logic is not adequate for the full range of nominal transition systems, and in that quantifiers over infinite structures are not definable. In contrast, our use of finitely supported sets of conjuncts is adequate for all nominal transition systems (cf. Theorems 1 and 2 below) and admits quantifiers as derived operators (cf. Section 4 below). As in the simple example above, universal quantification over names $\forall x \in \mathcal{N}.A(x)$ is usually defined to mean that $A(n)$ must hold for all $n \in \mathcal{N}$.

We can define this as the (infinite) conjunction of all these $A(n)$. This set of conjuncts is not uniformly bounded if $n \in \mathrm{supp}(A(n))$. But it is supported by $\mathrm{supp}(A)$ since, for any permutation π not affecting $\mathrm{supp}(A)$ we have $\pi \cdot A(n) = A(\pi(n))$ which is also a conjunct; thus the set of conjuncts is unaffected by π.

Another novelty is the use of a nominal set of actions α with binders, and the formal definition of alpha-equivalence.We define it by structural recursion over formulas. Two conjunctions $\bigwedge_{i \in I} A_i$ and $\bigwedge_{i \in I} B_i$ are alpha-equivalent if for every conjunct A_i there is an alpha-equivalent conjunct B_j, and vice versa. The other cases are standard; two formulas $\langle\alpha\rangle A$ and $\langle\beta\rangle B$ are alpha-equivalent if there exists a permutation π, renaming the binding names of α to those of β, such that $\pi \cdot A$ and B are alpha-equivalent, and $\pi \cdot \alpha = \beta$. Moreover, π must leave names that are free in A invariant. The free names in a formula are also defined by structural recursion. Most cases are standard again; a name is free in $\langle\alpha\rangle A$ if it is in $\mathrm{supp}(\alpha)$ or free in A, and not contained in $\mathrm{bn}(\alpha)$. However, the free names in a conjunction are given by the support of its alpha-equivalence class (rather than by the union of free names in all conjuncts). This is analogous to the situation for nominal sets in general, whose support is not necessarily the same as the union of the support of its members. Fortunately, our formalization proves that we need not keep the details of this construction in mind, but can simply identify alpha-equivalent formulas. The notions of free names and support then coincide.

The validity of a formula A for a state P is written $P \models A$ and is defined by recursion over A as follows.

Definition 4.

$$\begin{array}{ll} P \models \bigwedge_{i \in I} A_i & \text{if for all } i \in I \text{ it holds that } P \models A_i \\ P \models \neg A & \text{if not } P \models A \\ P \models \varphi & \text{if } P \vdash \varphi \\ P \models \langle\alpha\rangle A & \text{if there exists } P' \text{ such that } P \xrightarrow{\alpha} P' \text{ and } P' \models A \end{array}$$

In the last clause we assume that $\langle\alpha\rangle A$ is a representative of its alpha-equivalence class such that $\mathrm{bn}(\alpha) \# P$.

Lemma 1. $\models$ is equivariant.

Proof: By the Equivariance Principle in Pitts (2013) [Pit13, page 21]. A more detailed proof that verifies $P \models A \iff \pi \cdot P \models \pi \cdot A$ for any permutation π has been formalized in Isabelle. The proof proceeds by structural induction on A, using equivariance of all involved relations. For the case $\langle\alpha\rangle A$ in particular, we use the fact that if $\langle\alpha'\rangle A' = \langle\alpha\rangle A$, then $\langle\pi \cdot \alpha'\rangle(\pi \cdot A') = \langle\pi \cdot \alpha\rangle(\pi \cdot A)$. □

Definition 5. Two states P and Q are *logically equivalent*, written $P \doteq Q$, if for all A it holds that $P \models A$ iff $Q \models A$

Theorem 1. $P \dot{\sim} Q \implies P \doteq Q$

Proof: The proof has been formalized in Isabelle. Assume $P \dot{\sim} Q$. We show $P \models A \iff Q \models A$ by structural induction on A.

(1) Base case: $A = \varphi$. Then $P \models A \iff P \vdash \varphi \iff Q \vdash \varphi \iff Q \models A$ by static implication and symmetry of $\dot{\sim}$.
(2) Inductive steps $\bigwedge_{i \in I} A_i$ and $\neg A$: immediate by induction.

(3) Inductive step $\langle\alpha\rangle A$: Assume $P \models \langle\alpha\rangle A$. Then for some alpha-variant $\langle\alpha'\rangle A' = \langle\alpha\rangle A$, $\exists P' . P \xrightarrow{\alpha'} P'$ and $P' \models A'$. Without loss of generality we assume also $\mathrm{bn}(\alpha')\#Q$, otherwise just find an alpha-variant of $\langle\alpha'\rangle A'$ where this holds. Then by simulation $\exists Q' . Q \xrightarrow{\alpha'} Q'$ and $P' \mathrel{\dot\sim} Q'$. By induction and $P' \models A'$ we get $Q' \models A'$, whence by definition $Q \models \langle\alpha\rangle A$. The proof of $Q \models \langle\alpha\rangle A \implies P \models \langle\alpha\rangle A$ is symmetric, using the fact that $P \mathrel{\dot\sim} Q$ entails $Q \mathrel{\dot\sim} P$. □

The converse result uses the idea of *distinguishing formulas*.

Definition 6. A *distinguishing formula* for P and Q is a formula A such that $P \models A$ and not $Q \models A$.

The following lemma says that we can find such a formula where, a bit surprisingly, the support does not depend on Q.

Lemma 2. If $P \mathrel{\dot{\not=}} Q$ then there exists a distinguishing formula B for P and Q such that $\mathrm{supp}(B) \subseteq \mathrm{supp}(P)$.

Proof: The proof has been formalized in Isabelle. If $P \mathrel{\dot{\not=}} Q$ then there exists a distinguishing formula A for P and Q, i.e., $P \models A$ and not $Q \models A$. Let $\Pi_P = \{\pi \mid n \in \mathrm{supp}(P) \Rightarrow \pi(n) = n\}$ be the group of name permutations that leave $\mathrm{supp}(P)$ invariant and let $\mathcal{B}$ be the Π_P-orbit of A, i.e.,

$$\mathcal{B} = \{\pi \cdot A \mid \pi \in \Pi_P\}$$

In the terminology of Pitts [Pit13] ch. 5, $\mathcal{B}$ is $\mathrm{hull}_{\mathrm{supp}(P)}A$. Clearly, if $a, b\#P$ and $\pi \in \Pi_P$ then $(a\,b) \circ \pi \in \Pi_P$. Thus $(a\,b) \cdot \mathcal{B} = \mathcal{B}$ and therefore $\mathrm{supp}(\mathcal{B}) \subseteq \mathrm{supp}(P)$. This means that the formula $B = \bigwedge \mathcal{B}$ is well-formed and $\mathrm{supp}(B) \subseteq \mathrm{supp}(P)$. For all $\pi \in \Pi_P$ we have by definition $P = \pi \cdot P$ and by equivariance $\pi \cdot P \models \pi \cdot A$, i.e., $P \models \pi \cdot A$. Therefore $P \models B$. Furthermore, since the identity permutation is in Π_P and not $Q \models A$ we get not $Q \models B$. □

Note that in this proof B uses a conjunction which is not uniformly bounded.

Theorem 2. $P \doteq Q \implies P \mathrel{\dot\sim} Q$

Proof: The proof has been formalized in Isabelle. We establish that $\doteq$ is a bisimulation. Obviously it is symmetric. So assume $P \doteq Q$, we need to prove the two requirements on a bisimulation.

(1) Static implication. $P \vdash \varphi$ iff $P \models \varphi$ iff $Q \models \varphi$ iff $Q \vdash \varphi$.
(2) Simulation. The proof is by contradiction. Assume that $\doteq$ does not satisfy the simulation requirement. Then there exist P, Q, P', α with $\mathrm{bn}(\alpha)\#Q$ such that $P \doteq Q$ and $P \xrightarrow{\alpha} P'$ and, letting $\mathcal{Q} = \{Q' \mid Q \xrightarrow{\alpha} Q'\}$, for all $Q' \in \mathcal{Q}$ it holds that $P' \mathrel{\dot{\not=}} Q'$. Assume $\mathrm{bn}(\alpha)\#P$, otherwise just find an alpha-variant of the transition satisfying this. By $P' \mathrel{\dot{\not=}} Q'$, for all $Q' \in \mathcal{Q}$ there exists a distinguishing formula for P' and Q'. The formula may depend on Q', and by Lemma 2 we can find such a distinguishing formula $B_{Q'}$ for P' and Q' with $\mathrm{supp}(B_{Q'}) \subseteq \mathrm{supp}(P')$. Therefore the formula

$$B = \bigwedge_{Q' \in \mathcal{Q}} B_{Q'}$$

is well-formed with support included in $\mathrm{supp}(P')$. We thus get that $P \models \langle\alpha\rangle B$ but not $Q \models \langle\alpha\rangle B$, contradicting $P \doteq Q$. □

This proof of the simulation property is different from other such proofs in the literature. For finite branching transition systems, $\mathcal{Q}$ is finite so finite conjunction is enough to define B. For transition systems with the name preservation property, i.e., that if $P \xrightarrow{\alpha} P'$ then $\mathrm{supp}(P') \subseteq \mathrm{supp}(P) \cup \mathrm{supp}(\alpha)$, uniformly bounded conjunction suffices with common support $\mathrm{supp}(P) \cup \mathrm{supp}(Q) \cup \mathrm{supp}(\alpha)$. Without the name preservation property, we here use a not uniformly bounded conjunction in Lemma 2.

4. Derived formulas

Dual connectives. We define logical disjunction $\bigvee_{i \in I} A_i$ in the usual way as $\neg \bigwedge_{i \in I} \neg A_i$, when the indexing set I has bounded cardinality and $\{A_i \mid i \in I\}$ has finite support. A special case is $I = \{1, 2\}$: we then write $A_1 \wedge A_2$ instead of $\bigwedge_{i \in I} A_i$, and dually for $A_1 \vee A_2$. We write $\top$ for the empty conjunction $\bigwedge_{i \in \emptyset}$, and $\bot$ for $\neg\top$. We also write $A \implies B$ for $B \vee \neg A$. The must modality $[\alpha]A$ is defined as $\neg\langle\alpha\rangle\neg A$, and requires A to hold after every possible α-labelled transition from the current state. Note that $\mathrm{bn}(\alpha)$ bind into A. For example, $[\alpha](A \wedge B)$ is equivalent to $[\alpha]A \wedge [\alpha]B$, and dually $\langle\alpha\rangle(A \vee B)$ is equivalent to $\langle\alpha\rangle A \vee \langle\alpha\rangle B$.

Quantifiers. Let S be any finitely supported set of bounded cardinality and use v to range over members of S. Write $A\{{}^{v}\!/\!{}_{x}\}$ for the substitution of v for x in A, and assume this substitution function is equivariant. Then we define $\forall x \in S.A$ as $\bigwedge_{v \in S} A\{{}^{v}\!/\!{}_{x}\}$. There is not necessarily a common finite support for the formulas $A\{{}^{v}\!/\!{}_{x}\}$, for example if S is some term algebra over names, but the set $\{A\{{}^{v}\!/\!{}_{x}\} \mid v \in S\}$ has finite support bounded by $\{x\} \cup \mathrm{supp}(S) \cup \mathrm{supp}(A)$. In our examples in Section 8, substitution is defined inductively on the structure of formulas, based on primitive substitution functions for actions and state predicates, avoiding capture and preserving the binding names of actions.

Existential quantification $\exists x \in S.A$ is defined as the dual $\neg\forall x \in S.\neg A$. When X is a metavariable used to range over a nominal set $\mathcal{X}$, we simply write X for "$X \in \mathcal{X}$". As an example, $\forall a.A$ means that the formula $A\{{}^{n}\!/\!{}_{a}\}$ holds for all names $n \in \mathcal{N}$.

New name quantifier. The new name quantifier $\text{И}x.A$ intuitively states that $P \models A\{{}^{n}\!/\!{}_{x}\}$ holds where n is a fresh name for P. For example, suppose we have actions of the form $a\,b$ for input, and $\overline{a}\,b$ for output where a and b are free names, then the formula $\text{И}x.[a\,x]\langle\overline{b}\,x\rangle\top$ expresses that whenever a process inputs a fresh name x on channel a, it has to be able to output that name on channel b. If the name received is not fresh (i.e., already present in P) then P is not required to do anything. Therefore this formula is weaker than $\forall x.[a\,x]\langle\overline{b}\,x\rangle\top$.

Since A and P have finite support, if $P \models A\{{}^{n}\!/\!{}_{x}\}$ holds for some n fresh for P, by equivariance it also holds for almost all n, i.e., all but finitely many n. Conversely, if it holds for almost all n, it must hold for some $n \# \mathrm{supp}(P)$. Therefore $\text{И}x$ is often pronounced "for almost all x". In other words, $P \models \text{И}x.A$ holds if $\{x \mid P \models A(x)\}$ is a cofinite set of names [Pit13, Definition 3.8].

To avoid the need for a substitution function, we here define the new name quantifier using name swapping $(a\ n)$. Compared to using the standard notion of substitution for nominal term algebras our definition yields the same result, but is applicable without assuming any particular structure on actions or predicates. Letting $\mathrm{COF} = \{S \subseteq \mathcal{N} \mid \mathcal{N} \setminus S \text{ is finite}\}$ we thus encode $\text{И}x.A$ as $\bigvee_{S \in \mathrm{COF}} \bigwedge_{n \in S} (x\,n) \cdot A$. This formula states there is a cofinite set of names such that for all of them A holds. The support of $\bigwedge_{n \in S} (x\,n) \cdot A$ is bounded by

$(\mathcal{N} \setminus S) \cup \text{supp}(A)$ where $S \in \text{COF}$, and the support of the encoding $\bigvee_{S \in \text{COF}} \bigwedge_{n \in S} (x\,n) \cdot A$ is bounded by $\text{supp}(A)$.

Next step. We can generalise the action modality to sets of actions: if T is a finitely supported set of actions, we write $\langle T \rangle A$ for $\bigvee_{\alpha \in T} \langle \alpha \rangle A$. The support of $\{\langle \alpha \rangle A \mid \alpha \in T\}$ is bounded by $\text{supp}(T) \cup \text{supp}(A)$ and thus finite. Dually, we write $[T]A$ for $\neg \langle T \rangle \neg A$, denoting that A holds after all transitions with actions in T. Note that binding names in actions in T bind into A, and so $\langle \alpha \rangle A$ is equivalent to $\langle \{\alpha\} \rangle A$ for any α.

To encode the next-step modality, let $\text{ACT}_A = \{\alpha \mid \text{bn}(\alpha) \# A\}$. Note that $\text{supp}(\text{ACT}_A) \subseteq \text{supp}(A)$ is finite. We write $\langle\,\rangle A$ for $\langle \text{ACT}_A \rangle A$, meaning that we can make some (non-capturing) transition to a state where A holds. As an example, $\langle\,\rangle \top$ means that the current state is not deadlocked. The dual modality $[\,]A = \neg \langle\,\rangle \neg A$ means that A holds after every transition from the current state. Larsen [Lar88] uses the same approach to define next-step operators in HML, though his version is less expressive since he uses a finite action set to define the next-step modality.

5. Fixpoint operators

Fixpoint operators are a way to introduce recursion into a logic. For example, they can be used to concisely express safety and liveness properties of a transition system, where by safety we mean that some invariant holds for all reachable states, and by liveness that some property will eventually hold. Kozen [Koz83] introduced the least ($\mu X.A$) and the greatest ($\nu X.A$) fixpoints operators in modal logic. Intuitively, the least fixpoint operator states a property that holds for states of a finite path, while the greatest holds for states of an infinite path.

By combining the fixpoints and next-step operators, we can encode the temporal logic CTL [CE82], following Emerson [Eme96]. The CTL formula $\mathsf{AG}\,A$, which states that A holds along all paths, is defined as $\nu X.A \wedge [\,]X$. Dually the formula $\mathsf{EF}\,A$, stating the there is some path where A eventually holds, is defined $\mu X.A \vee \langle\,\rangle X$. These are special cases of more general formulas: the formula $\mathsf{A}[A \,\mathsf{U}\, B]$ states that for all paths A holds until B holds, and dually $\mathsf{E}[A \,\mathsf{U}\, B]$ states that there is a path along which A holds until B. They are encoded as $\nu X.B \vee ([\,]X \wedge A)$ and $\mu X.B \vee (\langle\,\rangle X \wedge A)$, respectively. For example, deadlock freedom is given by the CTL formula $\mathsf{AG}\,\langle\,\rangle \top$ expressing that every reachable state has a transition. The encoding of this formula is $\nu X.\langle\,\rangle \top \wedge [\,]X$.

We extend the logic of Definition 3 with the least fixpoint operator and give meaning to formulas as sets of satisfying states. We show that the meaning of the fixpoint operator is indeed a fixpoint. We then proceed to show that the least fixpoint operator can be directly encoded in the logic.

5.1. Logic with the least fixpoint operator.

Definition 7. We extend the nominal set of formulas with the least fixpoint operator:

$$A \quad ::= \quad \bigwedge_{i \in I} A_i \quad | \quad \neg A \quad | \quad \varphi \quad | \quad \langle \alpha \rangle A \quad | \quad X \quad | \quad \mu X.A$$

where X ranges over a countable set of equivariant variables. We require that all occurrences of a variable X in a formula $\mu X.A$ are in the scope of an even number of negations.

An occurrence of a variable X in A is said to be free if it is not a subterm of some $\mu X.B$. We say that a formula A is closed if for every variable X, none of its occurrences in A are free. We use a capture-avoiding substitution function $[A/X]$ on formulas that substitutes A for free occurrences of the variable X. In particular, $(\langle\alpha\rangle B)[A/X] = \langle\alpha\rangle(B[A/X])$ when $\mathrm{bn}(\alpha)$ is fresh for A.

We give a semantics to formulas containing fixpoint modalities as sets of satisfying states.

Definition 8. A *valuation function* ε is a finitely supported map from variables to (finitely supported) sets of states. We write $\varepsilon[X \mapsto S]$ for the valuation function that maps X to S, and any variable $X' \neq X$ to $\varepsilon(X')$.
We define the *interpretation* of formula A under valuation ε by structural induction as the set of states $[\![A]\!]_\varepsilon$:

$$\begin{array}{rcl}
[\![\bigwedge_{i\in I} A_i]\!]_\varepsilon & = & \bigcap_{i\in I}[\![A_i]\!]_\varepsilon \\
[\![\neg A]\!]_\varepsilon & = & \textsc{states} - [\![A]\!]_\varepsilon \\
[\![\varphi]\!]_\varepsilon & = & \{P \mid P \vdash \varphi\} \\
[\![\langle\alpha\rangle A]\!]_\varepsilon & = & \left\{P \mid \exists \alpha'\, A'\, P' \,.\, \langle\alpha\rangle A = \langle\alpha'\rangle A' \wedge \mathrm{bn}(\alpha')\#P,\varepsilon \wedge P \xrightarrow{\alpha'} P' \wedge P' \in [\![A']\!]_\varepsilon\right\} \\
[\![X]\!]_\varepsilon & = & \varepsilon(X) \\
[\![\mu X.A]\!]_\varepsilon & = & \bigcap\left\{S \in \mathcal{P}_{\mathrm{fs}}(\textsc{states}) \mid [\![A]\!]_{\varepsilon[X\mapsto S]} \subseteq S\right\}
\end{array}$$

We write $[\![\ ]\!]$ for the function $(A,\varepsilon) \mapsto [\![A]\!]_\varepsilon$.

Lemma 3. $[\![\ ]\!]$ is equivariant.

Proof. By the Equivariance Principle [Pit13, page 21]. □

Lemma 4. For any formula A and valuation function ε, $[\![A]\!]_\varepsilon \in \mathcal{P}_{\mathrm{fs}}(\textsc{states})$.

Proof. By equivariance of $[\![\ ]\!]$, $\mathrm{supp}([\![A]\!]_\varepsilon) \subseteq \mathrm{supp}(A) \cup \mathrm{supp}(\varepsilon)$. □

Temporal operators such as "eventually" can be encoded using the least fixpoint operator: the formula $\mu X.\langle\alpha\rangle X \vee A$ states that eventually A holds along some path labelled with α. We define the greatest fixpoint operator $\nu X.A$ in terms of the least as $\neg\mu X.\neg A[\neg X/X]$. Using the greatest fixpoint operator we can state global invariants: $\nu X.[\alpha]X \wedge A$ expresses that A holds along all paths labelled with α. We can freely mix the fixpoint operators to obtain formulas like $\nu X.[\alpha]X \wedge (\mu Y.\langle\beta\rangle Y \vee A)$, which means that for each state along any path labelled with α, a state where A holds is reachable along a path labelled with β.

As sanity checks for our definition, we prove that the interpretation of formulas without fixpoint modalities is unchanged, and that the interpretation of the formula $\mu X.A$ is indeed the least fixpoint of the function $F^\varepsilon_A : S \mapsto [\![A]\!]_{\varepsilon[X\mapsto S]}$.

Proposition 2. Let A be a formula as in Definition 3. Then for any valuation function ε and state P, $P \models A$ if and only if $P \in [\![A]\!]_\varepsilon$.

Proof. By structural induction on A. The clauses for X and $\mu X.A'$ in Definition 8 are not used. The interesting case is

Case $\langle\alpha\rangle A'$: Assume $P \models \langle\alpha\rangle A'$. Without loss of generality assume also $\mathrm{bn}(\alpha)\#P,\varepsilon$, otherwise just find an alpha-variant of $\langle\alpha\rangle A'$ where this holds. From Definition 4 we obtain P' such that $P \xrightarrow{\alpha} P'$ and $P' \models A'$. Then $P' \in [\![A']\!]_\varepsilon$ by the induction hypothesis, hence $P \in [\![\langle\alpha\rangle A']\!]_\varepsilon$ by Definition 8.

Next, assume $P \in [\![\langle\alpha\rangle A']\!]_\varepsilon$. From Definition 8 we obtain an alpha-variant $\langle\alpha'\rangle A'' = \langle\alpha\rangle A'$ and P' such that $\mathrm{bn}(\alpha')\#P$, $P \xrightarrow{\alpha'} P'$ and $P' \in [\![A'']\!]_\varepsilon$. Then $P' \models A''$ by the induction hypothesis. Hence $P \models \langle\alpha'\rangle A'' = \langle\alpha\rangle A'$ by Definition 4.

□

Lemma 5. For any formula A and valuation function ε, F_A^ε has finite support.

Proof. By equivariance of $[\![\ ]\!]$, $\mathrm{supp}(F_A^\varepsilon) \subseteq \mathrm{supp}(A) \cup \mathrm{supp}(\varepsilon)$. □

Lemma 6. For any formula $\mu X.A$ and valuation function ε, the function $F_A^\varepsilon \colon \mathcal{P}_{\mathrm{fs}}(\textsc{states}) \to \mathcal{P}_{\mathrm{fs}}(\textsc{states})$ is monotonic with respect to subset inclusion.

Proof. By structural induction on A, for arbitrary ε. Let $S, T \in \mathcal{P}_{\mathrm{fs}}(\textsc{states})$ such that $S \subseteq T$. We prove a more general statement: if all occurrences of X in A are positive (i.e., within the scope of an even number of negations), $F_A^\varepsilon(S) \subseteq F_A^\varepsilon(T)$, and if all occurrences of X in A are negative (i.e., within the scope of an odd number of negations), $F_A^\varepsilon(T) \subseteq F_A^\varepsilon(S)$. The interesting case is

Case $\mu X'.A'$: If $X = X'$, note that for any $U, V \in \mathcal{P}_{\mathrm{fs}}(\textsc{states})$, $\varepsilon[X \mapsto U][X \mapsto V] = \varepsilon[X \mapsto V]$. Therefore, $[\![\mu X'.A']\!]_{\varepsilon[X\mapsto S]} = [\![\mu X'.A']\!]_{\varepsilon[X\mapsto T]}$ is immediate from Definition 8.

Otherwise, $X \neq X'$. Suppose that all occurrences of X in A are positive. Then all occurrences of X in A' are positive, and for any $V \in \mathcal{P}_{\mathrm{fs}}(\textsc{states})$, $[\![A']\!]_{\varepsilon[X'\mapsto V][X\mapsto S]} \subseteq [\![A']\!]_{\varepsilon[X'\mapsto V][X\mapsto T]}$ by the induction hypothesis applied to A' and $\varepsilon[X' \mapsto V]$. Since $X \neq X'$, for any $U, V \in \mathcal{P}_{\mathrm{fs}}(\textsc{states})$, $\varepsilon[X \mapsto U][X' \mapsto V] = \varepsilon[X' \mapsto V][X \mapsto U]$. Thus,

$$[\![\mu X'.A']\!]_{\varepsilon[X\mapsto S]} = \bigcap \left\{ S' \in \mathcal{P}_{\mathrm{fs}}(\textsc{states}) \mid [\![A']\!]_{\varepsilon[X\mapsto S][X'\mapsto S']} \subseteq S' \right\} \subseteq \bigcap \left\{ S' \in \mathcal{P}_{\mathrm{fs}}(\textsc{states}) \mid [\![A']\!]_{\varepsilon[X\mapsto T][X'\mapsto S']} \subseteq S' \right\} = [\![\mu X'.A']\!]_{\varepsilon[X\mapsto T]}.$$

The case where all occurrences of X in A are negative is similar.

□

We use a nominal version of Tarski's fixpoint theorem [Tar55] to show existence, uniqueness, and the construction of the least fixpoint of F_A^ε. Note that the usual Tarski fixpoint theorem does not apply, since the lattice $\mathcal{P}_{\mathrm{fs}}(\textsc{states})$ is not necessarily complete.

Theorem 3. Suppose X is a nominal set, and $f \colon \mathcal{P}_{\mathrm{fs}}(X) \to \mathcal{P}_{\mathrm{fs}}(X)$ is finitely supported and monotonic with respect to subset inclusion. Then f has a least fixpoint $\mathrm{lfp}\, f$, and

$$\mathrm{lfp}\, f = \bigcap \{ S \in \mathcal{P}_{\mathrm{fs}}(\textsc{states}) \mid f(S) \subseteq S \}.$$

Proof. (Due to Pitts [Pit15])
Since f is finitely supported and $\bigcap$ is equivariant, also $\bigcap\{S \in \mathcal{P}_{\mathrm{fs}}(\textsc{states}) \mid f(S) \subseteq S\}$ is finitely supported (with support bounded by $\mathrm{supp}(f)$). It then follows by a replay of the usual Tarski argument that $\bigcap\{S \in \mathcal{P}_{\mathrm{fs}}(\textsc{states}) \mid f(S) \subseteq S\}$ is the least fixpoint of f. □

Finally, we can show that the interpretation of a fixpoint formula $\mu X.A$ is the least fixpoint of the function F_A^ε.

Proposition 3. For any formula $\mu X.A$ and valuation function ε, $[\![\mu X.A]\!]_\varepsilon = \mathrm{lfp}\, F_A^\varepsilon$.

Proof. Using Lemmas 5 and 6, the proposition is immediate from Theorem 3. □

5.2. **Encoding the least fixpoint operator.** The least fixpoint operator can be encoded in our logic of Section 3. The idea here is simple: we translate the fixpoint modality into a transfinite disjunction that at each step α unrolls the formula α times. This then semantically corresponds to a limit of an increasing chain generated by a monotonic function, i.e., a least fixpoint.

Definition 9. We define the formula $\mathsf{unroll}_\alpha(\mu X.A)$ where α is an ordinal bounded by κ (Definition 3) by transfinite induction.

$$\begin{array}{rcl} \mathsf{unroll}_0(\mu X.A) & = & \bot \\ \mathsf{unroll}_{\alpha+1}(\mu X.A) & = & A[\mathsf{unroll}_\alpha(\mu X.A)/X] \\ \mathsf{unroll}_\lambda(\mu X.A) & = & \bigvee_{\alpha<\lambda} \mathsf{unroll}_\alpha(\mu X.A) \text{ when } \lambda \text{ is a limit ordinal} \end{array}$$

We define the formula $\overline{A}$ homomorphically on the structure of A. The encoding $\overline{\mu X.A}$ of a fixpoint is its unrolling up to κ.

$$\begin{array}{rcl} \overline{\bigwedge_{i\in I} A_i} & = & \bigwedge_{i\in I} \overline{A_i} \\ \overline{\neg A} & = & \neg \overline{A} \\ \overline{\varphi} & = & \varphi \\ \overline{\langle\alpha\rangle A} & = & \langle\alpha\rangle \overline{A} \\ \overline{X} & = & X \\ \overline{\mu X.A} & = & \mathsf{unroll}_\kappa(\mu X.\overline{A}) \end{array}$$

Since unroll_α is equivariant for all $\alpha \leq \kappa$, and moreover $A \mapsto \overline{A}$ is equivariant, the encoding preserves the finite support property for conjunctions in A, and the disjunction in the fixpoint case has finite support bounded by $\mathrm{supp}(A)$. Clearly, $\overline{A}$ does not contain any fixpoint operators. Moreover, if A is closed, then $\overline{A}$ does not contain any variables, and is therefore a formula in the sense of Definition 3.

To show that the interpretation of $\mathsf{unroll}_\kappa(\mu X.\overline{A})$ indeed is the least fixpoint of F^ε_A, we use a nominal version of a chain fixpoint theorem for sets by Kuratowski (1922), augmented with a bound on the depth of the unrolling.

Theorem 4. Suppose X is a nominal set, and $f\colon \mathcal{P}_{\mathrm{fs}}(X) \to \mathcal{P}_{\mathrm{fs}}(X)$ is finitely supported and monotonic with respect to subset inclusion. Set $f^0 = \emptyset$, $f^{\alpha+1} = f(f^\alpha)$, and $f^\lambda = \bigcup_{\alpha<\lambda} f^\alpha$ for limit ordinals λ. Then f has a least fixpoint $\mathrm{lfp}\, f$, and for any cardinal κ with $\kappa > |X|$ we have

$$\mathrm{lfp}\, f = f^\kappa.$$

Proof. First, f^κ is finitely supported, since $\mathrm{supp}(f^\alpha) \subseteq \mathrm{supp}(f)$ for all ordinals α by transfinite induction. Also, using monotonicity of f, we have $f^\alpha \subseteq f^\beta$ for all $\alpha \leq \beta$.

We then show that f has a fixpoint f^α with $\alpha < \kappa$, by contradiction. Otherwise, for each $\alpha < \kappa$ there is $x_\alpha \in f^{\alpha+1} \setminus f^\alpha$. This yields an injective function $g : \kappa \to X$ with $g(\alpha) = x_\alpha$, which is a contradiction since $\kappa > |X|$.

We then have $f^\beta = f^\alpha$ for all $\beta \geq \alpha$, by transfinite induction. Since $\kappa > \alpha$, we have that $f^\kappa = f^\alpha$ is a fixpoint of f.

Let y be any fixpoint of f. For every ordinal β, $f^\beta \subseteq y$ by transfinite induction, so in particular $f^\kappa \subseteq y$. Thus f^κ is the least fixpoint of f. □

Lemma 7. For any formulas A, B and valuation function ε, if A does not contain any fixpoint operators, then $[\![A[B/X]]\!]_\varepsilon = [\![A]\!]_{\varepsilon[X\mapsto[\![B]\!]_\varepsilon]}$.

Proof. By structural induction on A. The clause for $\mu X.A'$ in Definition 8 is not used. □

Theorem 5. For any formula A and valuation function ε, $[\![\overline{A}]\!]_\varepsilon = [\![A]\!]_\varepsilon$.

Proof. By structural induction on A, for arbitrary ε. The interesting case is

Case $\mu X.A'$**:** We need to show that $[\![\overline{\mu X.A'}]\!]_\varepsilon = [\![\mu X.A']\!]_\varepsilon$. First, we compute the left-hand side to $[\![\overline{\mu X.A'}]\!]_\varepsilon = [\![\mathsf{unroll}_\kappa(\mu X.\overline{A'})]\!]_\varepsilon$. For the right-hand side, we have $[\![\mu X.A']\!]_\varepsilon = \mathrm{lfp}\, F^\varepsilon_{A'} = (F^\varepsilon_{A'})^\kappa$ by Proposition 3 and Theorem 4, whose assumptions follow from Lemmas 5 and 6. It therefore suffices to show $[\![\mathsf{unroll}_\alpha(\mu X.\overline{A'})]\!]_\varepsilon = (F^\varepsilon_{A'})^\alpha$ for all $\alpha \leq \kappa$. We proceed by transfinite induction on α.

(1) Base case: $[\![\mathsf{unroll}_0(\mu X.\overline{A'})]\!]_\varepsilon = [\![\bot]\!]_\varepsilon = \emptyset = (F^\varepsilon_{A'})^0$ by definition.

(2) Inductive step:

$$
\begin{aligned}
[\![\mathsf{unroll}_{\alpha+1}(\mu X.\overline{A'})]\!]_\varepsilon &= [\![\overline{A'}[\mathsf{unroll}_\alpha(\mu X.\overline{A'})/X]]\!]_\varepsilon \\
&\overset{(1)}{=} [\![\overline{A'}]\!]_{\varepsilon[X\mapsto[\![\mathsf{unroll}_\alpha(\mu X.\overline{A'})]\!]_\varepsilon]} \\
&\overset{(2)}{=} [\![\overline{A'}]\!]_{\varepsilon[X\mapsto(F^\varepsilon_{A'})^\alpha]} \\
&\overset{(3)}{=} [\![A']\!]_{\varepsilon[X\mapsto(F^\varepsilon_{A'})^\alpha]} \\
&= F^\varepsilon_{A'}((F^\varepsilon_{A'})^\alpha) \\
&= (F^\varepsilon_{A'})^{\alpha+1}
\end{aligned}
$$

as required. Above, equality (1) follows from Lemma 7, equality (2) follows from the induction hypothesis for α, and equality (3) follows from the outer induction hypothesis applied to A' and $\varepsilon[X \mapsto (F^\varepsilon_{A'})^\alpha]$.

(3) Limit case: The limit case is straightforward. We have

$$
\begin{aligned}
[\![\mathsf{unroll}_\lambda(\mu X.\overline{A'})]\!]_\varepsilon &= [\![\bigvee_{\alpha<\lambda} \mathsf{unroll}_\alpha(\mu X.\overline{A'})]\!]_\varepsilon \\
&= \bigcup_{\alpha<\lambda} [\![\mathsf{unroll}_\alpha(\mu X.\overline{A'})]\!]_\varepsilon \\
&\overset{(1)}{=} \bigcup_{\alpha<\lambda} (F^\varepsilon_{A'})^\alpha \\
&= (F^\varepsilon_{A'})^\lambda
\end{aligned}
$$

where equality (1) follows from the induction hypothesis for all $\alpha < \lambda$.

□

Every closed formula containing fixpoint operators can be translated into an equivalent formula without fixpoint operators.

Corollary 1. For any ε, P and closed formula A, we have $P \models \overline{A}$ iff $P \in [\![A]\!]_\varepsilon$.

Proof. By Theorem 5 and Proposition 2. □

6. Logics for variants of bisimilarity

The bisimilarity of Section 3 is of the early kind: any substitutive effect of an input (typically replacing a variable with the value received) must have manifested already in the action corresponding to the input, since we apply no substitution to the target state. Alternative treatments of substitutions include late-, open- and hyperbisimilarity, where the input action instead contains the variable to be replaced, and there are different ways to make sure that bisimulations are preserved by relevant substitutions.

In our definition of nominal transition systems there are no particular input variables in the states or in the actions, and thus no a priori concept of "substitution". We therefore choose to formulate the alternatives using so-called effect functions. An *effect* is simply a finitely supported function from states to states. For example, in the monadic pi-calculus the effects would be the functions replacing one name by another. In a value-passing calculus the effects would be substitutions of values for variables. In the psi-calculi framework the effects would be sequences of parallel substitutions. Our definitions and results are applicable to any of these; our only requirement is that the effects form a nominal set which we designate by $\mathcal{F}$. Variants of bisimilarity then correspond to requiring continuation after various effects. For example, if the action contains an input variable x then the effects appropriate for late bisimilarity would be substitutions for x.

We will formulate these variants as *F/L-bisimilarity*, where F (for *first*) represents the set of effects that must be observed before following a transition, and L (for *later*) is a function that represents how this set F changes depending on the action of a transition, i.e., $L(\alpha, F)$ is the set of effects that must follow the action α if the previous effect set was F. In the following let $\mathcal{P}_{\text{fs}}(\mathcal{F})$ ranged over by F be the finitely supported subsets of $\mathcal{F}$, and L range over equivariant functions from actions and $\mathcal{P}_{\text{fs}}(\mathcal{F})$ to $\mathcal{P}_{\text{fs}}(\mathcal{F})$.

Definition 10. An *L-bisimulation* where $L : \textsc{act} \times \mathcal{P}_{\text{fs}}(\mathcal{F}) \to \mathcal{P}_{\text{fs}}(\mathcal{F})$ is a $\mathcal{P}_{\text{fs}}(\mathcal{F})$-indexed family of symmetric binary relations on states satisfying the following:

If $R_F(P, Q)$ then:

1. *Static implication*: for all $f \in F$ it holds that $f(P) \vdash \varphi$ implies $f(Q) \vdash \varphi$.
2. *Simulation*: For all $f \in F$ and α, P' such that $\text{bn}(\alpha) \# f(Q), F$ there exist Q' such that

$$\text{if } f(P) \xrightarrow{\alpha} P' \text{ then } f(Q) \xrightarrow{\alpha} Q' \text{ and } R_{L(\alpha,F)}(P', Q')$$

We write $P \overset{F/L}{\sim} Q$, called F/L-bisimilarity, to mean that there exists an L-bisimulation R such that $R_F(P, Q)$.

Most strong bisimulation varieties can be formulated as F/L-bismilarity. Write $\text{id}_{\text{STATES}}$ for the identity function on states, ID for the singleton set $\{\text{id}_{\text{STATES}}\}$ and all_{ID} for the constant function $\lambda(\alpha, F).\text{ID}$.

- *Early bisimilarity*, precisely as defined in Definition 2, is $\text{ID} / \text{all}_{\text{ID}}$-bisimilarity.
- *Early equivalence*, i.e., early bisimilarity under all possible effects, is $\mathcal{F} / \text{all}_{\text{ID}}$-bisimilarity.
- *Late bisimilarity* is ID / L-bisimilarity, where $L(\alpha, F)$ yields the effects that represent substitutions for variables in input actions α (and ID for other actions).
- *Late equivalence* is similarly $\mathcal{F}$ / L-bisimilarity.
- *Open bisimilarity* is $\mathcal{F}$ / L-bisimilarity where $L(\alpha, F)$ is the set F minus all effects that change bound output names in α.
- *Hyperbisimilarity* is $\mathcal{F} / \lambda(\alpha, F).\mathcal{F}$-bisimilarity.

All of the above are generalizations of known and well-studied definitions. The original value-passing variant of CCS [Mil89] uses early bisimilarity. The original bisimilarity for the pi-calculus is of the late kind [MPW92], where it also was noted that late equivalence is the corresponding congruence. Early bisimilarity and equivalence and open bisimilarity for the pi-calculus were introduced in 1993 [MPW93, San93], and hyperbisimilarity for the fusion calculus in 1998 [PV98].

In view of this we only need to provide a modal logic adequate for F/L-bisimilarity; it can then immediately be specialized to all of the above variants. For this we introduce a new kind of logical operator as follows.

Definition 11. For each $f \in \mathcal{F}$ the logical unary *effect consequence* operator $\langle f \rangle$ has the definition

$$P \models \langle f \rangle A \quad \text{if} \quad f(P) \models A$$

Thus the formula $\langle f \rangle A$ means that A holds if the effect f is applied to the state. Note that by definition this distributes over conjunction and negation, e.g. $P \models \neg\langle f \rangle A$ iff $P \models \langle f \rangle \neg A$ iff not $f(P) \models A$ etc. The effect consequence operator is similar in spirit to the action modalities: both $\langle f \rangle A$ and $\langle \alpha \rangle A$ assert that something (an effect or action) must be possible and that A holds afterwards. Indeed, effects can be viewed as a special case of transitions (as formalised in Definition 13 below) which is why we give the operators a common syntactic appearance.

Now define the formulas that can directly use effects from F and after actions use effects according to L, ranged over by $A^{F/L}$, in the following way:

Definition 12. Given L as in Definition 10, for all $F \in \mathcal{P}_{\mathrm{fs}}(\mathcal{F})$ define $\mathcal{A}^{F/L}$ as the set of formulas given by the mutually recursive definitions:

$$A^{F/L} \quad ::= \quad \bigwedge_{i \in I} A_i^{F/L} \quad | \quad \neg A^{F/L} \quad | \quad \langle f \rangle \varphi \quad | \quad \langle f \rangle \langle \alpha \rangle A^{L(\alpha,F)/L}$$

where we require $f \in F$ and that the conjunction has bounded cardinality and finite support. Validity of a formula for a state P is defined as in Definitions 4 and 11, where in the last clause we assume that the formula is a representative of its alpha-equivalence class such that $\mathrm{bn}(\alpha) \# P, F$.

Lemma 8. If $A \in \mathcal{A}^{F/L}$ is a distinguishing formula for P and Q, then there exists a distinguishing formula $B \in \mathcal{A}^{F/L}$ for P and Q such that $\mathrm{supp}(B) \subseteq \mathrm{supp}(P, F)$.

Proof: The proof has been formalized in Isabelle. It is an easy generalisation of the proof of Lemma 2, just replace $\mathcal{A}$ by $\mathcal{A}^{F/L}$ and $\mathrm{supp}(P)$ by $\mathrm{supp}(P, F)$ everywhere. We additionally need to prove that $B \in \mathcal{A}^{F/L}$. Since L is equivariant we have $\mathrm{supp}(\mathcal{A}^{F/L}) \subseteq \mathrm{supp}(F)$, which means that $\pi \cdot A \in \mathcal{A}^{F/L}$ for all $\pi \in \Pi_P$, this establishes $B \in \mathcal{A}^{F/L}$. □

Let $P \overset{F/L}{=} Q$ mean that P and Q satisfy the same formulas in $\mathcal{A}^{F/L}$.

Theorem 6. $P \overset{F/L}{\sim} Q \quad \Leftrightarrow \quad P \overset{F/L}{=} Q$

Proof: The proof has been formalised in Isabelle. Direction $\Rightarrow$ is a generalization of Theorem 1.

(1) Base case: $A = \langle f \rangle \varphi$ and $f \in F$. Then $f(P) \vdash \varphi$. By static implication $f(Q) \vdash \varphi$, which means $Q \models A$.

(2) Inductive step $\langle f\rangle\langle\alpha\rangle A$ where $A \in \mathcal{A}^{F/L}$: Assume $P \models \langle f\rangle\langle\alpha\rangle A$. Then $\exists P' . f(P) \xrightarrow{\alpha} P'$ and $P' \models A$. Without loss of generality we assume also $\mathrm{bn}(\alpha)\#f(Q)$, otherwise just find an alpha-variant of the transition where this holds. Then by simulation $\exists Q' . f(Q) \xrightarrow{\alpha} Q'$ and $P' \overset{L(\alpha,F)/L}{\sim} Q'$. By induction and $P' \models A$ and $A \in \mathcal{A}^{F/L}$ we get $Q' \models A$, whence by definition $Q \models \langle f\rangle\langle\alpha\rangle A$.

The direction $\Leftarrow$ is a generalization of Theorem 2: we prove that $\overset{F/L}{=}$ is an F/L-bisimulation. The modified clauses are:

(1) Static implication. Assume $f \in F$, then $f(P) \vdash \varphi$ iff $P \models \langle f\rangle\varphi$ iff $Q \models \langle f\rangle\varphi$ iff $f(Q) \vdash \varphi$.
(2) Simulation. The proof is by contradiction. Assume that $\overset{F/L}{=}$ does not satisfy the simulation requirement. Then there exist $f \in F, P, Q, P', \alpha$ such that $P \overset{F/L}{=} Q$ and $f(P) \xrightarrow{\alpha} P'$ and, letting $\mathcal{Q} = \{Q' \mid f(Q) \xrightarrow{\alpha} Q'\}$, for all $Q' \in \mathcal{Q}$ it holds not $P' \overset{L(\alpha,F)/L}{=} Q'$. Choose $\mathrm{bn}(\alpha)\#f(P)$. Thus, for all $Q' \in \mathcal{Q}$ there exists a distinguishing formula in $\mathcal{A}^{L(\alpha,F)/L}$ for P' and Q'. The formula may depend on Q', and by Lemma 8 we can find such a distinguishing formula $B_{Q'} \in \mathcal{A}^{L(\alpha,F)/L}$ for P' and Q' with $\mathrm{supp}(B_{Q'}) \subseteq \mathrm{supp}(P', L(\alpha, F))$. Therefore the formula
$$B = \bigwedge_{Q' \in \mathcal{Q}} B_{Q'}$$
is well formed in $\mathcal{A}^{L(\alpha,F)/L}$ with support included in $\mathrm{supp}(P', L(\alpha, F))$. We thus get that $P \models \langle f\rangle\langle\alpha\rangle B$ but not $Q \models \langle f\rangle\langle\alpha\rangle B$, contradicting $P \overset{F/L}{=} Q$.

□

An alternative to the effect consequence operators is to transform the transition system such that standard (early) bisimulation on the transforms coincides with F/L-bisimilarity. The idea is to let the effect function be part of the transition relation, thus $f(P) = P'$ becomes $P \xrightarrow{f} P'$.

Definition 13. Assume $\mathcal{F}$ and L as above. The *L-transform* of a nominal transition system $\mathbf{T}$ is a nominal transition system where:

- The states are of the form $\textsc{ac}(F, f(P))$ and $\textsc{ef}(F, P)$, for $f \in F \in \mathcal{P}_{\mathrm{fs}}(\mathcal{F})$ and states P of $\mathbf{T}$. The intuition is that states of kind $\textsc{ac}$ can perform ordinary actions, and states of kind $\textsc{ef}$ can commit effects.
- The state predicates are those of $\mathbf{T}$.
- $\textsc{ac}(F, P) \vdash \varphi$ if in $\mathbf{T}$ it holds $P \vdash \varphi$, and $\textsc{ef}(F, P) \vdash \varphi$ never holds.
- The actions are the actions of $\mathbf{T}$ and the effects in $\mathcal{F}$.
- bn is as in $\mathbf{T}$, and additionally $\mathrm{bn}(f) = \emptyset$ for $f \in \mathcal{F}$.
- The transitions are of two kinds. If in $\mathbf{T}$ it holds $P \xrightarrow{\alpha} P'$, then there is a transition $\textsc{ac}(F, P) \xrightarrow{\alpha} \textsc{ef}(L(\alpha, F), P')$. And for each $f \in F$ it holds $\textsc{ef}(F, P) \xrightarrow{f} \textsc{ac}(F, f(P))$.

Theorem 7. $P \overset{F/L}{\sim} Q$ in $\mathbf{T}$ if and only if $\textsc{ef}(F, P) \dot{\sim} \textsc{ef}(F, Q)$ in the L-transform of $\mathbf{T}$.

Proof: For the direction $\Rightarrow$, assume that R is an L-bisimulation. Define R' on the L-transform by including $(\textsc{ef}(F, P), \textsc{ef}(F, Q))$ and $(\textsc{ac}(F, f(P)), \textsc{ac}(F, f(Q)))$ for all P, Q, f, F such that $f \in F$ and $R_F(P, Q)$. We now prove R' to be a simulation. Assume $R'(S, T)$.

(1) Static implication: Assume $S \vdash \varphi$. Then $S = \text{AC}(F, f(P))$ for some F, $f \in F$ and P and $f(P) \vdash \varphi$ holds in $\mathbf{T}$, and $T = \text{AC}(F, f(Q))$ with $R_F(P, Q)$. Thus $f(Q) \vdash \varphi$ whence $T \vdash \varphi$.
(2) Simulation: Assume $S \xrightarrow{\alpha} S'$. There are two cases:
   - $S = \text{EF}(F, P) \xrightarrow{f} \text{AC}(F, f(P)) = S'$ and $f \in F$. Then $T = \text{EF}(F, Q)$ where $R_F(P, Q)$. We get $T \xrightarrow{f} \text{AC}(F, f(Q)) = T'$ and $R'(S', T')$. Note here and below that $\text{bn}(f) = \emptyset$.
   - $S = \text{AC}(F, f(P)) \xrightarrow{\alpha} \text{EF}(L(\alpha, F), P') = S'$ and $f(P) \xrightarrow{\alpha} P'$, with $\text{bn}(\alpha)\#\text{AC}(F, f(Q))$. Then also $\text{bn}(\alpha)\#f(Q)$. We get $T = \text{AC}(F, f(Q))$ where $R_F(P, Q)$, so also $f(Q) \xrightarrow{\alpha} Q'$ with $R_{L(\alpha,F)}(P', Q')$. Thus $T \xrightarrow{\alpha} \text{EF}(L(\alpha, F), Q') = T'$, and $R'(S', T')$ as required.

For the direction $\Leftarrow$, assume that R' is a bisimulation in the L-transform of $\mathbf{T}$. Define R_F by $R_F(P, Q)$ if $R'(\text{EF}(F, P), \text{EF}(F, Q))$. We prove R an L-bisimulation. Assume $R_F(P, Q)$.

(1) Static implication: Let $f \in F$ and assume $f(P) \vdash \varphi$. Then $\text{EF}(F, P) \xrightarrow{f} \text{AC}(F, f(P))$. Since R' is a bisimulation we get $\text{EF}(F, Q) \xrightarrow{f} \text{AC}(F, f(Q))$. Now $f(P) \vdash \varphi$ means $\text{AC}(F, f(P)) \vdash \varphi$, and again since R' is a bisimulation $\text{AC}(F, f(Q)) \vdash \varphi$, which means $f(Q) \vdash \varphi$ as required.
(2) Simulation: Assume $f \in F$ and $f(P) \xrightarrow{\alpha} P'$ with $\text{bn}(\alpha)\#f(Q)$. Without loss of generality additionally assume the transition is represented by an alpha-variant such that $\text{bn}(\alpha)\#F$. We get the transitions

$$\text{EF}(F, P) \xrightarrow{f} \text{AC}(F, f(P)) \xrightarrow{\alpha} \text{EF}(L(\alpha, F), P')$$

Since R' is a bisimulation and $\text{bn}(\alpha)\#F, f(Q)$ we get a simulating sequence

$$\text{EF}(F, Q) \xrightarrow{f} \text{AC}(F, f(Q)) \xrightarrow{\alpha} \text{EF}(L(\alpha, F), Q')$$

This means that $f(Q) \xrightarrow{\alpha} Q'$ with $R_{L(\alpha,F)}(P', Q')$ as required. □

A direct consequence is that $P \overset{F/L}{\sim} Q$ iff $\text{EF}(F, P) \doteq \text{EF}(F, Q)$ in the L-transform of $\mathbf{T}$. Here the actions in the logic would include effects $f \in \mathcal{F}$.

7. Logics and bisimulations with unobservable transitions

The logics and bisimulations considered so far are all of the strong variety, in the sense that all transitions are regarded as equally significant. In many models of concurrent computation there is a special action which is *unobservable* in the sense that in a bisimulation, and also in the definition of the action modalities, the presence of extra such transitions does not matter. This leads to notions of weak bisimulation and accompanying weak modal logics. For example, a process that has no transitions is weakly bisimilar with any process that has only unobservable transitions, and these satisfy the same weak modal logic formulas. We shall here introduce these ideas into the nominal transition systems. The main source of complication over similar treatments in process algebras turns out to be the presence of state predicates. In general weak bisimulation for nominal transition systems is quite intricate and for details we refer to [JBPV10], where we explore psi-calculi, a particular class of nominal transition systems, and investigate different versions of weak bisimilarity.

To cater for unobservable transitions in our nominal transition systems, assume a special action τ with empty support. As usual, define $P \Rightarrow P'$ to mean that there exist $P_0, \ldots, P_n$ with $P = P_0$ and $P' = P_n$ such that for all $i \in [0, n-1]$ it holds $P_i \xrightarrow{\tau} P_{i+1}$, where we allow the case $n = 0$ and $P = P'$. Intuitively, $P \Rightarrow P'$ means that P can evolve to P' without an observer noticing. Also let $P \stackrel{\alpha}{\Rightarrow} P'$ mean that there exist P'', P''' such that $P \Rightarrow P''$ and $P'' \xrightarrow{\alpha} P'''$ and $P''' \Rightarrow P'$. Finally let $P \stackrel{\hat{\alpha}}{\Rightarrow} P'$ mean $P \Rightarrow P'$ if $\alpha = \tau$ and $P \stackrel{\alpha}{\Rightarrow} P'$ otherwise. Intuitively $P \stackrel{\hat{\alpha}}{\Rightarrow} P'$ means that P can evolve to P' through transitions with the only observable content α.

We shall here only pursue the special case where state predicates are *persistent* in the sense that if $P \vdash \varphi$ and $P \xrightarrow{\tau} P'$ then also $P' \vdash \varphi$. Persistent models are common enough that they merit interest; examples include most versions of the pi-calculus and models of constraint stores without retracts. Briefly put, without this assumption the weak bisimulation in Definition 16 below would not be preserved by natural forms of parallel composition of transition systems. Although we shall not go into details of parallel composition or other operators on transition systems in this paper, we do not want to exclude the possibility of studying them later. Without persistence the definition would have to become much more complicated, and we leave that for further work.

The normal way to define weak bisimilarity is to replace $Q \xrightarrow{\alpha} Q'$ with $Q \stackrel{\hat{\alpha}}{\Rightarrow} Q'$ in the simulation requirement. This results in the weak simulation criterion:

Definition 14. A binary relation R on STATES is a *weak simulation* if $R(P, Q)$ implies that for all α, P' such that $\mathrm{bn}(\alpha)\#Q$ there exist Q' such that

$$\text{if } P \xrightarrow{\alpha} P' \text{ then } Q \stackrel{\hat{\alpha}}{\Rightarrow} Q' \text{ and } R(P', Q')$$

However, just replacing the simulation requirement with weak simulation will not suffice. The reason is that through static implication an observer can still observe the state predicates directly, and thus distinguish between a state that satisfies φ and a state that can silently evolve to something that satisfies φ. Therefore we need a weak counterpart of static implication where τ transitions are treated specially. A first guess could be that if $P \vdash \varphi$ then $Q \Rightarrow Q' \vdash \varphi$, i.e., that for Q to be weakly bisimilar it is enough that Q can unobservably evolve to a state that satisfies φ, but also this turns out to be insufficient. In the follwing we give some examples to illustrate the difficulties. The first is that we need to consider sets of simultaneously satisfied predicates, since we may have an observer that can test for such sets.

Example 1.

Let there be an enumerable set $\{\varphi_i\}_{i \in \mathbb{N}}$ of state predicates, a state P satisfying all of them, and for each $i \in \mathbb{N}$ a state Q_i such that $Q_i \vdash \varphi_j$ iff $j < i$. Let there be transitions $Q_i \xrightarrow{\tau} Q_{i+1}$ for all $i \in \mathbb{N}$:

$$\underset{\varphi_0, \varphi_1, \ldots}{P} \qquad Q_0 \xrightarrow{\ \tau\ } \underset{\varphi_0}{Q_1} \xrightarrow{\ \tau\ } \underset{\varphi_0, \varphi_1}{Q_2} \xrightarrow{\ \tau\ } \cdots$$

Adopting the first guess above, that $Q \Rightarrow Q' \vdash \varphi$ is enough, we would have that P is weakly bisimilar to all Q_i. The reason is that for each φ_j there is a Q' such that $Q_i \Rightarrow Q' \vdash \varphi_j$.

However, an observer could observe all φ_i simultaneously in P but not in Q_i, and thus distinguish them. For example, in a model of concurrent computation with a parallel operator and conditionals we can admit a tester of kind "if $\forall i \in \mathbb{N}$. φ_i then ..." . We therefore need weak static implication to consider simultaneously satisfied sets of predicates. The idea is to strengthen

$$\forall \varphi. \text{ if } P \vdash \varphi \text{ then } \exists Q'.\ Q \Rightarrow Q' \vdash \varphi$$

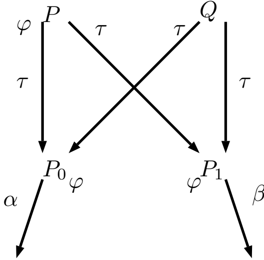
to

$$\exists Q'.\ Q \Rightarrow Q' \wedge \forall \varphi. \text{ if } P \vdash \varphi \text{ then } Q' \vdash \varphi$$

In other words, a single state Q' must simultaneously satisfy all state predicates true of P.

The second difficulty has to do with the interplay between state predicates and transitions.

Example 2. Consider states P, P_0, P_1 such that $P \xrightarrow{\tau} P_i$ and $P_i \vdash \varphi$, and where P_0 has a transition α and P_1 has another transition β. Let also $Q \xrightarrow{\tau} P_i$, and the only difference between P and Q be that $P \vdash \varphi$ but not $Q \vdash \varphi$:



Here we do not want to regard P and Q as weakly equivalent. Intuitively, an observer of Q that determines that φ holds must already have chosen one of P_0 and P_1, whereas this choice can be unresolved for the corresponding situation with P. For example, P in parallel with a process of kind "if φ then γ" can perform γ and retain the possibility of both α and β. For that reason we need weak bisimilarity to recur in the definition of weak static implication.

This leads to the following, where we use the notation $P \leq Q$ to mean that for all φ it holds $P \vdash \varphi$ implies $Q \vdash \varphi$.

Definition 15. A binary relation R on states is a *weak static implication* if $R(P, Q)$ implies that there exists Q' such that $Q \Rightarrow Q'$ and $R(P, Q')$ and $P \leq Q'$.

Note that the weak static implication is a quite strong requirement: Q must be able to evolve to some Q' which satisfies all state predicates of P and at the same time remains in R.

Definition 16. A *weak bisimulation* is a symmetric binary relation on states satisfying both weak simulation and weak static implication. We write $P \dot{\approx} Q$ to mean that there exists a weak bisimulation R such that $R(P, Q)$.

Lemma 9. If $P \dot{\approx} Q$ and $P \stackrel{\hat{\alpha}}{\Rightarrow} P'$ then for some Q' it holds $P' \dot{\approx} Q'$ and $Q \stackrel{\hat{\alpha}}{\Rightarrow} Q'$.

The proof is just by applying the weak simulation property repeatedly.

Lemma 10. $\dot{\approx}$ is an equivariant equivalence.

Proof: Equivariance follows from the equivariance principle. Reflexivity and symmetry are immediate. For transitivity assume R_1 and R_2 are weak bisimulations, we prove that $R = R_1 \circ R_2$ is a weak bisimulation. Symmetry is immediate, and weak simulation is easy using Lemma 9. For weak static implication, assume (1) $R_1(P_1, Q)$ and (2) $R_2(Q, P_2)$. By (1) and weak static implication, for some Q' we have $Q \Rightarrow Q'$ and $R_1(P_1, Q')$ and $P \leq Q'$. By (2) and Lemma 9 there is P_2' such that $P_2 \Rightarrow P_2'$ and (3) $R_2(Q', P_2')$. By (3) and weak static implication we get P_2'' such that $P_2' \Rightarrow P_2''$ and $R_2(Q', P_2'')$ and $Q' \leq P_2''$. So we have $P_2 \Rightarrow P_2''$ and since $\leq$ is transitive also $P_1 \leq P_2''$, and by definition $R(P_1, P_2'')$, as required. □

We shall now define a modal logic adequate for weak bisimilarity. As expected the logic has a weak action modality $\langle\langle\alpha\rangle\rangle$ to talk about a sequence of transitions with observable content α. The main difference from the logic in Definition 3 is that there is an operator to check the simultaneous satisfaction of both a finitely supported set of state predicates and a modal logic formula. In the following we let Φ range over finitely supported sets of state predicates.

Definition 17. The set of *weak formulas* ranged over by W is defined inductively as

$$W \quad ::= \quad \bigwedge_{i \in I} W_i \quad | \quad \neg W \quad | \quad \Phi \textbf{ also } W \quad | \quad \langle\langle\alpha\rangle\rangle W$$

As in Definition 3, in the conjunction we require that $\{W_i\}_{i \in I}$ is of bounded cardinality and finitely supported. We introduce the abbreviation $P \vdash \Phi$ to mean $\forall \varphi \in \Phi \,.\, P \vdash \varphi$.

Definition 18.

$$\begin{array}{ll} P \models \bigwedge_{i \in I} W_i & \text{if for all } i \in I \text{ it holds that } P \models W_i \\ P \models \neg W & \text{if not } P \models W \\ P \models \Phi \textbf{ also } W & \text{if } \exists P' \,.\, P \Rightarrow P' \text{ and } P' \vdash \Phi \text{ and } P' \models W \\ P \models \langle\langle\alpha\rangle\rangle W & \text{if } \exists P' \,.\, P \overset{\hat{\alpha}}{\Rightarrow} P' \text{ and } P' \models W \end{array}$$

Note that Φ **also** W is stronger than requiring both Φ and W separately, since it is required that a single P' satisfies both.

We write $\top$ for the empty conjunction and abbreviate Φ **also** $\top$ to Φ when there is no risk of misunderstanding. As an example consider the formula $\{\varphi_i\}_{i \in \mathbb{N}}$. This says that it is possible to reach a state through unobservable transitions where all φ_i hold. In Example 1 above it holds for P but not for any Q_i. In constrast the formula $\bigwedge_{i \in \mathbb{N}} \{\varphi_i\}$ holds for both P and all Q_i. For Example 2 consider the formula

$$\{\varphi\} \textbf{ also } (\langle\langle\alpha\rangle\rangle\top \wedge \langle\langle\beta\rangle\rangle\top)$$

saying that there must be a similarly reachable state where both φ holds and actions with observable content α and β respectively must be possible. This holds for P but not for Q. In contrast the formula $\{\varphi\} \wedge \langle\langle\alpha\rangle\rangle\top \wedge \langle\langle\beta\rangle\rangle\top$ holds for both P and Q.

Proposition 4. $\models$ is equivariant.

Proof: By the equivariance principle. □

Definition 19. Two states P and Q are *weakly logically equivalent*, written $P \doteqdot Q$, if

$$\text{for all } W \text{ it holds that } P \models W \text{ iff } Q \models W$$

To prove that weak logical equivalence coincides with weak bisimilarity we first introduce Φ_P as the set of state predicates satisfied by P.

Definition 20. $\Phi_P = \{\varphi \mid P \vdash \varphi\}$

Note that $Q \vdash \Phi_P$ iff $P \leq Q$.

Lemma 11. $\mathrm{supp}(\Phi_P) \subseteq \mathrm{supp}(P)$

The proof is by standard nominal reasoning using equivariance of $\vdash$.

Theorem 8.

$$P \dot{\approx} Q \quad \Leftrightarrow \quad P \doteqdot Q$$

The proof of direction $\Rightarrow$ is as for Theorem 1 using structural induction on formulas. Note that the empty conjunction is now the only base case. The significant new case is the inductive step Φ **also** W. For this inductive step, assume $P \models \Phi$ **also** W. Then for some P' it holds $P \Rightarrow P'$ with $P' \vdash \Phi$ and $P' \models W$. By $P \dot{\approx} Q$ and Lemma 9 we get Q' such that $Q \Rightarrow Q'$ and $P' \dot{\approx} Q'$. By weak static implication and $P' \vdash \Phi$ there must exist Q'' such that $Q' \Rightarrow Q''$ with $Q'' \vdash \Phi$ and $P' \dot{\approx} Q''$, the latter by induction implies $Q'' \models W$. With $Q \Rightarrow Q' \Rightarrow Q''$ this gives $Q \models \Phi$ **also** W as required.

The other inductive steps are similar to Theorem 1. For the case $\langle\langle\alpha\rangle\rangle W$ Lemma 9 is again needed.

For direction $\Leftarrow$ we establish that $\doteqdot$ is a weak bisimulation. So assume $P \doteqdot Q$. We need to prove the requirements on a weak bisimulation. For weak static implication the proof is by contradiction. Assume that $\doteqdot$ does not satisfy the weak static implication requirement. Then there exist P, Q such that for all Q' such that $Q \Rightarrow Q'$ and $Q' \vdash \Phi_P$ there exists a distinguishing formula $B_{Q'}$ such that $P \models B_{Q'}$ and not $Q' \models B_{Q'}$. By a variant of Lemma 2 for weak formulas (that lemma only depends on infinitary conjunction in the logic and works regardless of all other operators) $\mathrm{supp}(B_{Q'}) \subseteq \mathrm{supp}(P)$ which means that the infinite conjunction B of all these $B_{Q'}$ is well formed. By Lemma 11 Φ_P has finite support. We thus have that Φ_P **also** B is a well formed distinguishing formula for P and Q, contradicting $P \doteqdot Q$.

That $\doteqdot$ is a weak simulation is proven just as in Theorem 2. □

8. Related work and examples

In this first part of this section we discuss other modal logics for process calculi, with a focus on how their constructors can be captured by finitely supported conjunction in our HML. This comparison is by necessity somewhat informal: formal correspondence fails to hold due to differences in the conjunction operator of the logic (finite, uniformly bounded or unbounded vs. bounded support). In the later part of this section, we obtain novel, adequate HMLs for more recent process calculi.

8.1. **Existing Hennessy-Milner Logics.**

HML for CCS. The first published HML is Hennessy and Milner (1985) [HM85]. They use finite (binary) conjunction with the assumption of image-finiteness for ordinary CCS. The same goes for the value-passing calculus and logic by Hennessy and Liu (1995) [HL95], where image-finiteness is due to a late semantics and the logic contains quantification over data values. A similar idea and argument is in a logic for LOTOS by Calder et al. (2002) [CMS02], though that only considers stratified bisimilarity up to ω.

Hennessy and Liu's value-passing calculus is based on ordinary CCS. In this calculus, a receiving process $a(x).P$ can participate in a synchronization on a, becoming an abstraction $(x)P$ where v is a bound variable. Dually, a sending process $\overline{a}\,v.Q$ becomes a concretion (v, Q) where v is a value. The abstraction and concretion above react as part of the synchronization on a, yielding $P\{^v/_x\} \mid Q$. To capture the operations of abstractions and concretions in our framework, we add effects $\mathrm{id}_{\mathrm{STATES}}$ and $?v$, with $?v((x)P) = P\{^v/_x\}$, and transitions $(v, P) \xrightarrow{!v} P$. Letting $L(a?, _) = \{?v \mid v \in \text{values}\}$ and $L(\alpha, _) = \{\mathrm{id}_{\mathrm{STATES}}\}$ otherwise, late bisimilarity is $\{\mathrm{id}_{\mathrm{STATES}}\}/L$-bisimilarity as defined in Section 6. We can then encode their universal quantifier $\forall x.A$ as $\bigwedge_v \langle ?v\rangle A\{^v/_x\}$, which has support $\mathrm{supp}(A) \setminus \{x\}$, and their output modality $\langle c!x\rangle A$ as $\langle c!\rangle \bigvee_v \langle !v\rangle A\{^v/_x\}$, with support $\{c\} \cup (\mathrm{supp}(A) \setminus \{x\})$.

An infinitary HML for CCS is discussed in Milner's book (1989) [Mil89], where also the process syntax contains infinite summation. There are no restrictions on the indexing sets and no discussion about how this can exhaust all names. The adequacy theorem is proved by stratifying bisimilarity and using transfinite induction over all ordinals, where the successor step basically is the contraposition of the argument in Theorem 2, though without any consideration of finite support. A more rigorous treatment of the same ideas is by Abramsky (1991) [Abr91] where uniformly bounded conjunction is used throughout.

Pi-calculus. The first HML for the pi-calculus is by Milner et al. (1993) [MPW93], where infinite conjunction is used in the early semantics and conjunctions are restricted to use a finite set of free names. The adequacy proof is of the same structure as in this paper. The logic of Section 3, applied to the pi-calculus transition system from which bound input actions $x(y)$ have been removed, contains the logic $\mathcal{F}$ of Milner et al., or the equipotent logic $\mathcal{FM}$ if we take the set of name matchings $[a = b]$ as state predicates.

Spi Calculus. Frendrup et al. (2002) [FHNJ02] provide three Hennessy-Milner logics for the spi calculus [AG99]. All three logics use infinite quantification without any consideration of finite support. The transition system used is a variant of the one by Boreale et al. (2001) [BDNP01], where a state is a pair $\sigma \triangleright P$ of a process P and its environment σ: a substitution that maps environment variables to public names and messages received from the process. This version of the spi calculus has expressions ξ, that are terms constructed from names and environment variables using encryption and decryption operators, and messages M, that only contain names and encryption. Substitution $\xi\sigma$ replaces environment variables in ξ with their values in σ, and evaluation $\mathbf{e}(\xi)$ is a partial function that attempts to perform the decryptions in ξ, yielding a message M if all decryptions are successful.

As usual for the spi calculus, the bisimulation (and logic) is defined in terms of the environment actions, rather than the process actions. In Frendrup's version of Boreale's environment-sensitive transition system, the transition labels are related to the process actions in the following way: when a process P receives message M on channel a, the label

of a corresponding environment-sensitive transition $\sigma \triangleright P \xrightarrow{a\,\xi} \sigma' \triangleright P'$ describes how the environment σ computed the message $M = \mathbf{e}(\xi\sigma)$. For process output of message M on channel a, the corresponding environment-sensitive transition is simply $\sigma \triangleright P \xrightarrow{\overline{a}} \sigma' \triangleright P'$; the message M can be recovered from the updated environment σ'.

The logics of Frendrup et al. include a matching modality $[M = N]A$ that is defined using implication: $\sigma \triangleright P \models [M = N]A$ iff $\mathbf{e}(M\sigma) = \mathbf{e}(N\sigma)$ implies $\sigma \triangleright P \models A$. This is equipotent to having matching as a state predicate, since we can rewrite all non-trivial guards by $[M = N]A \iff A \vee \neg[M = N]\top$.

The logic of Section 3, applied to the nominal transition system with the environment labels τ, $\overline{a}$ and $a\,\xi$ above has the same modalities as the logic $\mathcal{F}$ of Frendrup et al. The logic $\mathcal{EM}$ by Frendrup et al. replaces the simple input modality by an early input modality $\langle \underline{a}\,(x)\rangle^E A$, which (after a minor manipulation of the input labels) can be encoded as the conjunction $\bigwedge_\xi \langle a\,\xi\rangle A\{^\xi/_x\}$ with support $\mathrm{supp}(A) \setminus \{x\}$. We do not consider their logic $\mathcal{LM}$ that uses a late input modality, since its application relies on sets that do not have finite support [FHNJ02, Theorem 6.12], which are not meaningful in nominal logic.

Frendrup et al. claim to "characterize early and late versions of the environment sensitive bisimilarity of [BDNP01]", but this claim only holds with some modification. First the definition of static equivalence [FNJ01, Definition 22] that is used in the adequacy proofs is strictly stronger than the one that appears in the published paper [FHNJ02, Definition 3.4]. Thus, the adequacy results [FHNJ02, Theorems 6.3, 6.4, and 6.14] are false as stated, but can be repaired by substituting the stronger notion of static equivalence. Then Frendrup's logics and bisimilarities become sound, but not complete, with respect to the bisimilarity of [BDNP01], since the latter uses the weaker notion of static equivalence (Definition 3.4).

Applied Pi-calculus. A more recent work is a logic by Hüttel and Pedersen (2007) [HP07] for the applied pi-calculus by Abadi and Fournet (2001) [AF01], where the adequacy theorem uses image-finiteness of the semantics in the contradiction argument. Similarly to the spi calculus, there is a requirement that terms M received by a process P can be computed from the current knowledge available to an observer of the process, which we here write $M \in \mathcal{S}(P)$.

The logic contains atomic formulae for term equality (indistinguishability) in the frame of a process, corresponding to our state predicates. However, Hüttel and Pedersen use a notion of equality (and thus static equivalence) that is stronger than the corresponding relation by Abadi and Fournet, and that is not well-defined modulo alpha-renaming [Ped06, p. 20]. Since our framework is based on nominal sets, we must identify alpha-equivalent processes, and instead use Abadi and Fournet's notion of term equality.

Hüttel and Pedersen's logic includes an early input modality and an existential quantifier. The early input modality $\langle \underline{a}\,(x)\rangle A$ can be straightforwardly encoded as the conjunction $\bigwedge_M \langle \underline{a}\, M\rangle A\{^M/_x\}$, with support $\{a\} \cup (\mathrm{supp}(A) \setminus \{x\})$. The definition of the existential quantifier takes the observer knowledge into account: P satisfies $\exists x.A$ if $x\#P$ and there is $M \in \mathcal{S}(P)$ such that $\{^M/_x\} \mid P$ satisfies A. The condition $M \in \mathcal{S}(P)$ makes the quantifier difficult to encode using effects, since there is no corresponding state predicate (for good reason: the main property modelled by cryptographic process calculi is that different cipher texts $\mathrm{E}(M, k)$ and $\mathrm{E}(N, k)$ are indistinguishable unless the key k is known). We instead add an action (x) with $\mathrm{bn}((x)) = x$ and transitions $P \xrightarrow{(x)} \{^M/_x\} \mid P$ if $M \in \mathcal{S}(P)$ and $x\#P$. We can then encode $\exists x.A$ as $\langle (x)\rangle A$.

Fusion calculus. In an HML for the fusion calculus by Haugstad et al. (2006) [HTV06] the fusions (i.e., equality relations on names) are action labels φ. The corresponding modal operator $\langle\varphi\rangle A$ has the semantics that the formula A must be satisfied for all substitutive effects of φ (intuitively, substitutions that map each name to a fixed representative for its equivalence class). In order to represent fusion actions in the logics in this paper, we add substitution effects σ such that $\sigma(P') = P'\sigma$. The fusion modality $\langle\varphi\rangle A$ can then be encoded in our framework as $\langle\varphi\rangle \bigvee_\sigma \langle\sigma\rangle A\sigma$, where the parameter σ of the disjunction ranges over the (finite set of) substitutive effects of φ. Their adequacy theorem uses the contradiction argument with infinite conjunction, with no argument about finiteness of names for the distinguishing formula.

Nominal transition systems. De Nicola and Loreti (2008) [DL08] define a general format for multiple-labelled transition systems with labels for name revelation and resource management, and an associated modal logic with name equality predicates, name quantification ($\exists$ and И), and a fixed-point modality. In contrast, we seek a small and expressive HML for general nominal transition systems. Indeed, the logic of De Nicola and Loreto can be seen as a special case of ours: their different transition systems can be merged into a single one, and we can encode their quantifiers and fixpoint operator as described in Section 4. Nominal SOS of Cimini et al. (2012) [CMRG12] is also a special case of our nominal transition systems.

8.2. **Recent process calculi.** In each of the final two examples below, no HML has to our knowledge yet been proposed, and we immediately obtain one by instantiating the logic in the present paper.

Concurrent constraint pi calculus. The concurrent constraint pi calculus (CC-pi) by Buscemi and Montanari (2007) [BM07] extends the explicit fusion calculus [WG05] with a more general notion of constraint stores c. Using the labelled transition system of CC-pi and the associated bisimulation (Definition 2), we immediately get an adequate modal logic.

The reference equivalence for CC-pi is open bisimulation [BM08] (closely corresponding to hyperbisimulation in the fusion calculus [PV98]), which differs from labelled bisimulation in two ways: First, two equivalent processes must be equivalent under all store extensions. To encode this, we let the effects $\mathcal{F}$ be the set of constraint stores c different from 0, and let $c(P) = c \mid P$. Second, when simulating a labelled transition $P \xrightarrow{\alpha} P'$, the simulating process Q can use any transition $Q \xrightarrow{\beta} Q'$ with an equivalent label, as given by a state predicate $\alpha = \beta$. As an example, if $\alpha = \overline{a}\langle x\rangle$ is a free output label then $P \vdash \alpha = \beta$ iff $\beta = \overline{b}\langle y\rangle$ where $P \vdash a = b$ and $P \vdash x = y$. To encode this, we transform the labels of the transition system by replacing them with their equivalence classes, i.e., $P \xrightarrow{\alpha} P'$ becomes $P \xrightarrow{[\alpha]_P} P'$ where $\beta \in [\alpha]_P$ iff $P \vdash \beta = \alpha$. Hyperbisimilarity (Definition 10) on this transition system then corresponds to open bisimilarity, and the modal logic defined in Section 6 is adequate.

Psi-calculi. In psi-calculi by Bengtson et al. (2011) [BJPV11], the labelled transitions take the form $\Psi \rhd P \xrightarrow{\alpha} P'$, where the assertion environment Ψ is unchanged after the step. We model this as a nominal transition system by letting the set of states be pairs (Ψ, P) of assertion environments and processes, and define the transition relation by $(\Psi, P) \xrightarrow{\alpha} (\Psi, P')$ if $\Psi \rhd P \xrightarrow{\alpha} P'$. The notion of bisimulation used with psi-calculi also uses an assertion

environment and is required to be closed under environment extension, i.e., if $\Psi \triangleright P \sim Q$, then $\Psi \otimes \Psi' \triangleright P \sim Q$ for all Ψ'. We let the effects $\mathcal{F}$ be the set of assertions, and define $\Psi((\Psi', P)) = (\Psi \otimes \Psi', P)$. Hyperbisimilarity on this transition system then subsumes the standard psi-calculi bisimilarity, and the modal logic defined in Section 6 is adequate.

9. Conclusion

We have given a general account of transition systems and Hennessy-Milner Logic using nominal sets. The advantage of our approach is that it is more expressive than previous work. We allow infinite conjunctions that are not uniformly bounded, meaning that we can encode e.g. quantifiers and the next-step operator. We have given ample examples of how the definition captures different variants of bisimilarity and how it relates to many different versions of HML in the literature.

We have formalized the results of Section 3, including Theorems 1 and 2, using Nominal Isabelle [UK12].[1] Nominal Isabelle is an implementation of nominal logic in Isabelle/HOL [NPW02], a popular interactive proof assistant for higher-order logic. It adds convenient specification mechanisms for, and automation to reason about, datatypes with binders.

However, Nominal Isabelle does not directly support infinitely branching datatypes. Therefore, the mechanization of formulas (Definition 3) was challenging. We construct formulas from first principles in higher-order logic, by defining an inductive datatype of *raw* formulas (where alpha-equivalent raw formulas are *not* identified). The datatype constructor for conjunction recurses through sets of raw formulas of bounded cardinality, a feature made possible only by a recent re-implementation of Isabelle/HOL's datatype package [BHL$^+$14].

We then define alpha-equivalence of raw formulas. For finitely branching datatypes, alpha-equivalence is based on a notion of free variables. Here, to obtain the correct notion of free variables of a conjunction, we define alpha-equivalence and free variables via mutual recursion. This necessitates a fairly involved termination proof. (All recursive functions in Isabelle/HOL must be terminating.) To obtain formulas, we quotient raw formulas by alpha-equivalence, and finally carve out the subtype of all terms that can be constructed from finitely supported ones. We then prove important lemmas; for instance, a strong induction principle for formulas that allows the bound names in $\langle\alpha\rangle A$ to be chosen fresh for any finitely supported context.

Our development, which in total consists of about 2700 lines of Isabelle definitions and proofs, generalizes the constructions that Nominal Isabelle performs for finitely branching datatypes to a type with infinite branching. To our knowledge, this is the first mechanization of an infinitely branching nominal datatype in a proof assistant.

Acknowledgements

We thank Andrew Pitts for enlightening discussions on nominal datatypes with infinitary constructors, and Dmitriy Traytel for providing an Isabelle/HOL formalization of cardinality-bounded sets.

[1] Our Isabelle theories are available at `https://github.com/tjark/ML-for-NTS`.

References

[Abr91] Samson Abramsky. A domain equation for bisimulation. *Journal of Information and Computation*, 92(2):161–218, 1991.

[AF01] Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. In *Proceedings of POPL '01*, pages 104–115. ACM, January 2001.

[AG99] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The Spi calculus. *Information and Computation*, 148(1):1–70, 1999.

[BDNP01] Michele Boreale, Rocco De Nicola, and Rosario Pugliese. Proof techniques for cryptographic processes. *SIAM Journal on Computing*, 31(3):947–986, 2001.

[BHL+14] Jasmin Christian Blanchette, Johannes Hölzl, Andreas Lochbihler, Lorenz Panny, Andrei Popescu, and Dmitriy Traytel. Truly modular (co)datatypes for Isabelle/HOL. In Gerwin Klein and Ruben Gamboa, editors, *Proceedings of ITP 2014*, volume 8558 of *LNCS*, pages 93–110. Springer, 2014.

[BJPV11] Jesper Bengtson, Magnus Johansson, Joachim Parrow, and Björn Victor. Psi-calculi: a framework for mobile processes with nominal data and logic. *Logical Methods in Computer Science*, 7(1), 2011.

[BM07] Maria Grazia Buscemi and Ugo Montanari. CC-Pi: A constraint-based language for specifying service level agreements. In Rocco De Nicola, editor, *Proceedings of ESOP 2007*, volume 4421 of *LNCS*, pages 18–32. Springer, 2007.

[BM08] Maria Grazia Buscemi and Ugo Montanari. Open bisimulation for the concurrent constraint pi-calculus. In Sophia Drossopoulou, editor, *Proceedings of ESOP 2008*, volume 4960 of *LNCS*, pages 254–268. Springer, 2008.

[CE82] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In Dexter Kozen, editor, *Logic of Programs, Workshop*, volume 131 of *LNCS*, pages 52–71. Springer, 1982.

[CMRG12] Matteo Cimini, Mohammad Reza Mousavi, Michel A. Reniers, and Murdoch J. Gabbay. Nominal SOS. *Electronic Notes in Theoretical Computer Science*, 286:103–116, September 2012. Proceedings of the 28th Conference on the Mathematical Foundations of Programming Semantics (MFPS XXVIII).

[CMS02] Muffy Calder, Savi Maharaj, and Carron Shankland. A modal logic for full LOTOS based on symbolic transition systems. *The Computer Journal*, 45(1):55–61, 2002.

[DL08] Rocco De Nicola and Michele Loreti. Multiple-labelled transition systems for nominal calculi and their logics. *Mathematical Structures in Computer Science*, 18(1):107–143, 2008.

[Eme96] E. Allen Emerson. Model checking and the Mu-calculus. In Neil Immerman and Phokion G. Kolaitis, editors, *Descriptive Complexity and Finite Models, Proceedings of a DIMACS Workshop*, volume 31 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 185–214. American Mathematical Society, 1996.

[FHNJ02] Ulrik Frendrup, Hans Hüttel, and Jesper Nyholm Jensen. Modal logics for cryptographic processes. *Electronic Notes in Theoretical Computer Science*, 68(2):124–141, 2002. Proceedings of EXPRESS'02, 9th International Workshop on Expressiveness in Concurrency.

[FNJ01] Ulrik Frendrup and Jesper Nyholm Jensen. Bisimilarity in the spi-calculus. Master's thesis, Aalborg University, 2001. http://projekter.aau.dk/projekter/en/studentthesis/bisimilarity-in-the-spicalculus(f1636334-8677-4fdf-aaa8-66f9d8e0fbf7).html.

[HL95] Matthew Hennessy and Xinxin Liu. A modal logic for message passing processes. *Acta Informatica*, 32(4):375–393, 1995.

[HM85] Matthew Hennessy and Robin Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32(1):137–161, 1985.

[HP07] Hans Hüttel and Michael D. Pedersen. A logical characterisation of static equivalence. *Electronic Notes in Theoretical Computer Science*, 173:139–157, 2007. Proceedings of the 23rd Conference on the Mathematical Foundations of Programming Semantics (MFPS XXIII).

[HTV06] Arild Martin Møller Haugstad, Anders Franz Terkelsen, and Thomas Vindum. A modal logic for the fusion calculus. Unpublished, Aalborg University, http://vbn.aau.dk/ws/files/61067487/1149104946.pdf, 2006.

[JBPV10] Magnus Johansson, Jesper Bengtson, Joachim Parrow, and Björn Victor. Weak equivalences in psi-calculi. In *Proceedings of the 25th Annual IEEE Symposium on Logic in Computer Science, LICS 2010, 11-14 July 2010, Edinburgh, United Kingdom*, pages 322–331, 2010.

[Koz83] Dexter Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27(3):333 – 354, 1983. Special Issue Ninth International Colloquium on Automata, Languages and Programming (ICALP) Aarhus, Summer 1982.

[Lar88] Kim G. Larsen. Proof systems for Hennessy-Milner logic with recursion. In M. Dauchet and M. Nivat, editors, *Proceedings of CAAP '88*, volume 299 of *LNCS*, pages 215–230. Springer, 1988.

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I. *Information and Computation*, 100(1):1–40, 1992.

[MPW93] Robin Milner, Joachim Parrow, and David Walker. Modal logics for mobile processes. *Theoretical Computer Science*, 114(1):149 – 171, 1993.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[PBE+15] Joachim Parrow, Johannes Borgström, Lars-Henrik Eriksson, Ramunas Gutkovas, and Tjark Weber. Modal logics for nominal transition systems. In Luca Aceto and David de Frutos-Escrig, editors, *26th International Conference on Concurrency Theory, CONCUR 2015, Madrid, Spain, September 1.4, 2015*, volume 42 of *LIPIcs*, pages 198–211. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015.

[Ped06] Michael Pedersen. Logics for the applied pi calculus. Master's thesis, Aalborg University, 2006. BRICS RS-06-19.

[Pit13] Andrew M. Pitts. *Nominal Sets*. Cambridge University Press, 2013. Cambridge Books Online.

[Pit15] Andy Pitts, 2015. Personal communication.

[PV98] Joachim Parrow and Björn Victor. The fusion calculus: Expressiveness and symmetry in mobile processes. In *Proceedings of LICS 1998*, pages 176–185. IEEE Computer Society Press, 1998.

[San93] Davide Sangiorgi. A theory of bisimulation for the π-calculus. In Eike Best, editor, *Proceedings of CONCUR '93*, volume 715 of *LNCS*, pages 127–142. Springer, 1993.

[Tar55] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.

[UK12] Christian Urban and Cezary Kaliszyk. General bindings and alpha-equivalence in Nominal Isabelle. *Logical Methods in Computer Science*, 8(2), 2012.

[WG05] Lucian Wischik and Philippa Gardner. Explicit fusions. *Theoretical Computer Science*, 304(3):606–630, 2005.

Paper II

Logical Methods in Computer Science
Vol. 12(1:8)2016, pp. 1–49
www.lmcs-online.org

Submitted Oct. 20, 2014
Published Mar. 31, 2016

A SORTED SEMANTIC FRAMEWORK FOR APPLIED PROCESS CALCULI

JOHANNES BORGSTRÖM, RAMŪNAS GUTKOVAS, JOACHIM PARROW, BJÖRN VICTOR, AND JOHANNES ÅMAN POHJOLA

Computing Science Division Department of Information Technology Uppsala University
e-mail address: {johannes.borgstrom, ramunas.gutkovas, Joachim.Parrow, Bjorn.Victor, johannes.aman-pohjola}@it.uu.se

Abstract. Applied process calculi include advanced programming constructs such as type systems, communication with pattern matching, encryption primitives, concurrent constraints, nondeterminism, process creation, and dynamic connection topologies. Several such formalisms, e.g. the applied pi calculus, are extensions of the the pi-calculus; a growing number is geared towards particular applications or computational paradigms.

Our goal is a unified framework to represent different process calculi and notions of computation. To this end, we extend our previous work on psi-calculi with novel abstract patterns and pattern matching, and add sorts to the data term language, giving sufficient criteria for subject reduction to hold. Our framework can directly represent several existing process calculi; the resulting transition systems are isomorphic to the originals up to strong bisimulation. We also demonstrate different notions of computation on data terms, including cryptographic primitives and a lambda-calculus with erratic choice. Finally, we prove standard congruence and structural properties of bisimulation; the proof has been machine-checked using Nominal Isabelle in the case of a single name sort.

1. Introduction

There is today a growing number of high-level constructs in the area of concurrency. Examples include type systems, communication with pattern matching, encryption primitives, concurrent constraints, nondeterminism, and dynamic connection topologies. Combinations of such constructs are included in a variety of application oriented process calculi. For each such calculus its internal consistency, in terms of congruence results and algebraic laws, must be established independently. Our aim is a framework where many such calculi fit and where such results are derived once and for all, eliminating the need for individual proofs about each calculus.

2012 ACM CCS: [**Theory of computation**]: Semantics and Reasoning—Program Semantics—Operational semantics; [**Software and its engineering**]: Software Notations and Tools—System Description Languages—System modeling languages.

Key words and phrases: Expressiveness, Pattern matching, Type systems, Theorem proving, pi-calculus, Nominal sets.

This project is financially supported by the Swedish Foundation for Strategic Research.

LOGICAL METHODS IN COMPUTER SCIENCE DOI:10.2168/LMCS-12(1:8)2016
© J. Borgström, R. Gutkovas, J. Parrow, B. Victor, and J. Åman Pohjola
Creative Commons

Our effort in this direction is the framework of psi-calculi [BJPV11], which provides machine-checked proofs that important meta-theoretical properties, such as compositionality of bisimulation, hold in all instances of the framework. We claim that the theoretical development is more robust than that of other calculi of comparable complexity, since we use a structural operational semantics given by a single inductive definition, and since we have checked most results in the interactive theorem prover Nominal Isabelle [Urb08].

In this paper we introduce a novel generalization of pattern matching, decoupled from the definition of substitution, and add sorts for data terms and names. The generalized pattern matching is a new contribution that holds general interest; here it allows us to directly capture computation on data in advanced process calculi, without elaborate encodings.

We evaluate our framework by providing instances that correspond to standard calculi, and instances that use several different notions of computation. We define strong criteria for a psi-calculus to *represent* another process calculus, meaning that they are for all practical purposes one and the same. Representation is stronger than the standard *encoding* correspondences e.g. by Gorla [Gor10], which define criteria for one language to encode the behaviour of another. The representations that we provide of other standard calculi advance our previous work, where we had to resort to nontrivial encodings with an unclear formal correspondence to the source calculus.

An extended abstract [BGP$^+$14] of the present paper has previously been published.

1.1. **Background: Psi-calculi.** In the following we assume the reader to be acquainted with the basic ideas of process algebras based on the pi-calculus, and explain psi-calculi by a few simple examples. Full definitions can be found in the references above, and for a reader not acquainted with our work we recommend the first few sections of [BJPV11] for an introduction.

A psi-calculus has a notion of data terms, ranged over by K, L, M, N, and we write $\overline{M}\ N\,.\,P$ to represent an agent sending the term N along the channel M (which is also a data term), continuing as the agent P. We write $\underline{K}(\lambda\widetilde{x})X\,.\,Q$ to represent an agent that can input along the channel K, receiving some object matching the pattern X, where $\widetilde{x}$ are the variables bound by the prefix. These two agents can interact under two conditions. First, the two channels must be *channel equivalent*, as defined by the channel equivalence predicate $M \dot{\leftrightarrow} K$. Second, N must match the pattern X.

Formally, a *transition* is of kind $\Psi \rhd P \xrightarrow{\alpha} P'$, meaning that in an environment represented by the *assertion* Ψ the agent P can do an action α to become P'. An assertion embodies a collection of facts used to infer *conditions* such as the channel equivalence predicate $\dot{\leftrightarrow}$. To continue the example, if $N = X[\widetilde{x} := \widetilde{L}]$ we will have $\Psi \rhd \overline{M}\ N\,.\,P \mid \underline{K}(\lambda\widetilde{x})X\,.\,Q \xrightarrow{\tau} P \mid Q[\widetilde{x} := \widetilde{L}]$ when additionally $\Psi \vdash M \dot{\leftrightarrow} K$, i.e. when the assertion Ψ entails that M and K represent the same channel. In this way we may introduce a parametrised equational theory over a data structure for channels. Conditions, ranged over by φ, can be tested in the **if** construct: we have that $\Psi \rhd \textbf{if}\ \varphi\ \textbf{then}\ P \xrightarrow{\alpha} P'$ when $\Psi \vdash \varphi$ and $\Psi \rhd P \xrightarrow{\alpha} P'$. In order to represent concurrent constraints and local knowledge, assertions can be used as agents: $(\!|\Psi|\!)$ stands for an agent that asserts Ψ to its environment. Assertions may contain names and these can be scoped; for example, in $P \mid (\nu a)((\!|\Psi|\!) \mid Q)$ the agent Q uses all entailments provided by Ψ, while P only uses those that do not contain the name a.

Assertions and conditions can, in general, form any logical theory. Also the data terms can be drawn from an arbitrary set. One of our major contributions has been to pinpoint the precise requirements on the data terms and logic for a calculus to be useful in the sense that the natural formulation of bisimulation satisfies the expected algebraic laws (see Section 2). It turns out that it is necessary to view the terms and logics as *nominal* [Pit03]. This means that there is a distinguished set of names, and for each term a well defined notion of *support*, intuitively corresponding to the names occurring in the term. Functions and relations must be *equivariant*, meaning that they treat all names equally. In addition, we impose straight-forward requirements on the combination of assertions, on channel equivalence, and on substitution. Our requirements are quite general, and therefore our framework accommodates a wide variety of applied process calculi.

1.2. **Extension: Generalized pattern matching.** In our original definition of psi-calculi ([BJPV11], called "the original psi-calculi" below), patterns are just terms and pattern matching is defined by substitution in the usual way: the output object N matches the pattern X with binders $\widetilde{x}$ iff $N = X[\widetilde{x} := \widetilde{L}]$. In order to increase the generality we now introduce a function MATCH which takes a term N, a sequence of names $\widetilde{x}$ and a pattern X, returning a set of sequences of terms; the intuition is that if $\widetilde{L}$ is in $\text{MATCH}(N, \widetilde{x}, X)$ then the term N matches the pattern X by instantiating $\widetilde{x}$ to $\widetilde{L}$. The receiving agent $\underline{K}(\lambda\widetilde{x})X\,.\,Q$ then continues as $Q[\widetilde{x} := \widetilde{L}]$.

As an example we consider a term algebra with two function symbols: $\texttt{enc}$ of arity three and $\texttt{dec}$ of arity two. Here $\texttt{enc}(N, n, k)$ means encrypting N with the key k and a random nonce n and and $\texttt{dec}(N, k)$ represents symmetric key decryption, discarding the nonce. Suppose an agent sends an encryption, as in $\overline{M}\ \texttt{enc}(N, n, k)\,.\,P$. If we allow all terms to act as patterns, a receiving agent can use $\texttt{enc}(x, y, z)$ as a pattern, as in $\underline{c}(\lambda x, y, z)\texttt{enc}(x, y, z)\,.\,Q$, and in this way decompose the encryption and extract the message and key. Using the encryption function as a destructor in this way is clearly not the intention of a cryptographic model. With the new general form of pattern matching, we can simply limit the patterns to not bind names in terms at key position. Together with the separation between patterns and terms, this allows to directly represent dialects of the spi-calculus as in Sections 5.2 and 5.3.

Moreover, the generalization makes it possible to safely use rewrite rules such as $\texttt{dec}(\texttt{enc}(M, N, K), K) \to M$. In the psi-calculi framework such evaluation is not a primitive concept, but it can be part of the substitution function, with the idea that with each substitution all data terms are normalized according to rewrite rules. Such evaluating substitutions are dangerous for two reasons. First, in the original psi-calculi they can introduce ill-formed input prefixes. The input prefix $\underline{M}(\lambda\widetilde{x})N$ is well-formed when $\widetilde{x} \subseteq \mathrm{n}(N)$, i.e. the names $\widetilde{x}$ must all occur in N; a rewrite of the well-formed $\underline{M}(\lambda y)\texttt{dec}(\texttt{enc}(N, y, k), k)\,.\,P$ to $\underline{M}(\lambda y)N\,.\,P$ yields an ill-formed agent when y does not appear in N. Such ill-formed agents could also arise from input transitions in some original psi-calculi; with the current generalization preservation of well-formedness is guaranteed.

Second, in the original psi-calculi there is a requirement that substituting $\widetilde{L}$ for $\widetilde{x}$ in M must yield a term containing all names in $\widetilde{L}$ whenever $\widetilde{x} \subseteq \mathrm{n}(M)$. The reason is explained at length in [BJPV11]; briefly put, without this requirement the scope extension law is unsound. If rewrites such as $\texttt{dec}(\texttt{enc}(M, N, K), K) \to M$ are performed by substitutions this requirement is not fulfilled, since a substitution may then erase the names in N and K.

However, a closer examination reveals that this requirement is only necessary for some uses of substitution. In the transition

$$\underline{M}(\lambda\widetilde{x})N.P \xrightarrow{\underline{K}\ N[\widetilde{x}:=\widetilde{L}]} P[\widetilde{x} := \widetilde{L}]$$

the non-erasing criterion is important for the substitution above the arrow ($N[\widetilde{x} := \widetilde{L}]$) but unimportant for the substitution after the arrow ($P[\widetilde{x} := \widetilde{L}]$). In the present paper, we replace the former of these uses by the MATCH function, where a similar non-erasing criterion applies. All other substitutions may safely use arbitrary rewrites, even erasing ones.

In this paper, we address these three issues by introducing explicit notions of patterns, pattern variables and matching. This allows us to control precisely which parts of messages can be bound by pattern-matching and how messages can be deconstructed, admit computations such as $\mathtt{dec}(\mathtt{enc}(M, N, K), K) \to M$. We obtain criteria that ensure that well-formedness is preserved by transitions, and apply these to the original psi-calculi [BJPV11] (Theorem 2.7) and to pattern-matching spi calculus [HJ06] (Lemma 5.3).

1.3. **Extension: Sorting.** Applied process calculi often make use of a sort system. The applied pi-calculus [AF01] has a name sort and a data sort; terms of name sort must not appear as subterms of terms of data sort. It also makes a distinction between input-bound variables (which may be substituted) and restriction-bound names (which may not). The pattern-matching spi-calculus [HJ06] uses a sort of patterns and a sort of implementable terms; every implementable term can also be used as a pattern.

To represent such calculi, we admit a user-defined sort system on names, terms and patterns. Substitutions are only well-defined if they conform to the sorting discipline. To specify which terms can be used as channels, and which values can be received on them, we use compatibility predicates on the sorts of the subject and the object in input and output prefixes. The conditions for preservation of sorting by transitions (subject reduction) are very weak, allowing for great flexibility when defining instances.

The restriction to well-sorted substitution also allows to avoid "junk": terms that exist solely to make substitutions total. A prime example is representing the polyadic pi-calculus as a psi-calculus. The terms that can be transmitted between agents are tuples of names. Since a tuple is a term it can be substituted for a name, even if that name is already part of a tuple. The result is that the terms must admit nested tuples of names, which do not occur in the original calculus. Such anomalies disappear when introducing an appropriate sort system; cf. Section 4.1.

1.4. **Related work.** Pattern-matching is in common use in functional programming languages. Scala admits pattern-matching of objects [EOW07] using a method `unapply` that turns the receiving object into a matchable value (e.g. a tuple). F# admits the definition of pattern cases independently of the type that they should match [SNM07], facilitating interaction with third-party and foreign-language code. Turning to message-passing systems, LINDA [Gel85] uses pattern-matching when receiving from a tuple space. Similarly, in Erlang, message reception from a mailbox is guarded by a pattern.

These notions of patterns, with or without computation, are easily supported by the MATCH construct. The standard first-match policy can be encoded by extending the pattern language with mismatching and conjunction [Kri09].

Pattern matching in process calculi. The pattern-matching spi-calculus [HJ06] limits which variables may be binding in a pattern in order to match encrypted messages without binding unknown keys (cf. Section 5.3). The Kell calculus [SS05] also uses pattern languages equipped with a match function. However, in the Kell calculus the channels are single names and appear as part of the pattern in the input prefix, patterns may match multiple communications simultaneously (à la join calculus), and first-order pattern variables only match names (not composite messages) which reduces expressiveness [Giv14].

The applied pi-calculus [AF01] models deterministic computation by using for data language a term algebra modulo an equational logic. ProVerif [Bla11] is a specialised tool for security protocol verification in an extension of applied pi, including a pattern matching construct. Its implementation allows pattern matching of tagged tuples modulo a user-defined rewrite system; this is strictly less general than the psi-calculus pattern matching described in this paper (cf. Section 5.1).

Other tools for process calculi extended with datatypes include mCRL2 [CGK+13] for ACP, which allows higher order sorted term algebras and equational logic, and PAT3 [LSD11] which includes a CSP♯ [SLDC09] module where actions built over types like booleans and integers are extended with C♯-like programs. In all these cases, the pattern matching is defined by substitution in the usual way.

Sort systems for mobile processes. Sorts for the pi-calculus were first described by Milner [Mil93], and were developed in order to remove nonsensical processes using polyadic communication, similar to the motivation for the present work.

In contrast, Hüttel's dependently typed psi-calculi [Hüt11, Hüt14] is intended for a more fine-grained control of the behaviour of processes, and is capable of capturing a wide range of earlier type systems for pi-like calculi formulated as instances of psi-calculi. In Hüttel's typed psi-caluli the term language is a free term algebra (without name binders), using the standard notions of substitution and matching, and not admitting any computation on terms.

In contrast, in our sorted psi-calculi terms and substitution are general. A given term always has a fixed sort, not dependent on any term or value and independent of its context. We also have important meta-theoretical results, with machine-checked proofs for the case of a single name sort, including congruence results and structural equivalence laws for well-sorted bisimulation, and the preservation of well-sortedness under structural equivalence; no such results exist for Hüttel's typed psi-calculi. Indeed, our sorted psi-calculi can be seen as a foundation for Hüttel's typed psi-calculi: we give a formal account of the separation between variables and names used in Hüttel's typed psi-calculi, and substantiate Hüttel's claim that "the set of well-[sorted] terms is closed under well-[sorted] substitutions, which suffices" (Theorem 3.19).

The state-of-the art report [HV13] of WG1 of the BETTY project (EU COST Action IC1201) is a comprehensive guide to behavioural types for process calculi.

Fournet et al. [FGM05] add type-checking for a general authentication logic to a process calculus with destructor matching; there the authentication logic is only used to specify program correctness, and does not influence the operational semantics in any way.

1.5. **Results and outline.** In Section 2 we define psi-calculi with the above extensions and prove preservation of well-formedness. In Section 3 we prove the usual algebraic properties

of bisimilarity. The proof is in two steps: a machine-checked proof for calculi with a single name sort, followed by manual proof based on the translation of a multi-sorted psi calculus instance to a corresponding single-sorted instance. We demonstrate the expressiveness of our generalization in Section 4 where we directly represent standard calculi, and in Section 5 where we give examples of calculi with advanced data structures and computations on them, even nondeterministic reductions.

2. Definitions

Psi-calculi are based on nominal data types. A nominal data type is similar to a traditional data type, but can also contain binders and identify alpha-variants of terms. Formally, the only requirements are related to the treatment of the atomic symbols called names as explained below. In this paper, we consider sorted nominal datatypes, where names and members of the data type may have different sorts.

We assume a set of sorts $\mathcal{S}$. Given a countable set of sorts for names $\mathcal{S}_{\mathcal{N}} \subseteq \mathcal{S}$, we assume countably infinite pair-wise disjoint sets of atomic *names* $\mathcal{N}_s$, where $s \in \mathcal{S}_{\mathcal{N}}$. The set of all names, $\mathcal{N} = \cup_s \mathcal{N}_s$, is ranged over by $a, b, \ldots, x, y, z$. We write $\widetilde{x}$ for a tuple of names $x_1, \ldots, x_n$ and similarly for other tuples, and $\widetilde{x}$ also stands for the set of names $\{x_1, \ldots, x_n\}$ if used where a set is expected. We let π range over permutations of tuples of names: $\pi \cdot \widetilde{x}$ is a tuple of names of the same length as $\widetilde{x}$, containing the same names with the same multiplicities.

A sorted *nominal set* [Pit03, GP01] is a set equipped with *name swapping* functions written $(a\ b)$, for any sort s and names $a, b \in \mathcal{N}_s$, i.e. name swappings must respect sorting. An intuition is that for any member T of a nominal set we have that $(a\ b) \cdot T$ is T with a replaced by b and b replaced by a. The support of a term, written $\mathrm{n}(T)$, is intuitively the set of names that can be be affected by name swappings on T. This definition of support coincides with the usual definition of free names for abstract syntax trees that may contain binders. We write $a\#T$ for $a \notin \mathrm{n}(T)$, and extend this to finite sets and tuples by conjunction. A function f is *equivariant* if $(a\ b) \cdot (f(T)) = f((a\ b) \cdot T)$ always holds; a relation $\mathcal{R}$ is equivariant if $x\ \mathcal{R}\ y$ implies that $(a\ b) \cdot x\ \mathcal{R}\ (a\ b) \cdot y$ holds; and a constant symbol C is equivariant if $(a\ b) \cdot C = C$. In particular, we require that all sorts $s \in \mathcal{S}$ are equivariant. A *nominal data type* is a nominal set together with some equivariant functions on it, for instance a substitution function.

2.1. Original Psi-calculi Parameters.

Sorted psi-calculi is an extension of the original psi-calculi framework [BJPV11], which are given by three nominal datatypes (data terms, conditions and assertions) as discussed in the introduction.

Definition 2.1 (Original psi-calculus parameters)**.** The psi-calculus parameters from the original psi-calculus are the following nominal data types: (data) terms $M, N \in \mathbf{T}$, conditions $\varphi \in \mathbf{C}$, and assertions $\Psi \in \mathbf{A}$; equipped with the following four equivariant operators: channel equivalence $\dot{\leftrightarrow} : \mathbf{T} \times \mathbf{T} \to \mathbf{C}$, assertion composition $\otimes : \mathbf{A} \times \mathbf{A} \to \mathbf{A}$, the unit assertion $\mathbf{1} \in \mathbf{A}$, and the entailment relation $\vdash\ \subseteq \mathbf{A} \times \mathbf{C}$.

The binary functions $\dot{\leftrightarrow}$ and $\otimes$ and the relation $\vdash$ above will be used in infix form. Two assertions are said to be equivalent, written $\Psi \simeq \Psi'$, if they entail the same conditions, i.e. for all φ we have that $\Psi \vdash \varphi \Leftrightarrow \Psi' \vdash \varphi$.

We impose certain requisites on the sets and operators. In brief, channel equivalence must be symmetric and transitive modulo entailment, the assertions with $(\otimes, \mathbf{1})$ must form an abelian monoid modulo $\simeq$, and $\otimes$ must be compositional w.r.t. $\simeq$ (i.e. $\Psi_1 \simeq \Psi_2 \implies \Psi \otimes \Psi_1 \simeq \Psi \otimes \Psi_2$). (For details see [BJPV11], and for examples of machine-checked valid instantiations of the parameters see [ÅP10].) In examples in this paper, we usually consider the trivial assertion monoid $\mathbf{A} = \{\mathbf{1}\}$, and let channel equivalence be term equality (i.e. $\mathbf{1} \vdash M \leftrightarrow N$ iff $M = N$).

2.2. **New parameters for generalized pattern-matching.** To the parameters of the original psi-calculi we add patterns X, Y, that are used in input prefixes; a function VARS which yields the possible combinations of binding names in the pattern, and a pattern-matching function MATCH, which is used when the input takes place. Intuitively, an input pattern $(\lambda\tilde{x})X$ matches a message N if there are $\tilde{L} \in \textsc{match}(N, \tilde{x}, X)$; the receiving agent then continues after substituting $\tilde{L}$ for $\tilde{x}$. If $\textsc{match}(N, \tilde{x}, X) = \emptyset$ then $(\lambda\tilde{x})X$ does not match N; if $|\textsc{match}(N, \tilde{x}, X)| > 1$ then one of the matches will be non-deterministically chosen. Below, we use "variable" for names that can be bound in a pattern.

Definition 2.2 (Psi-calculus parameters for pattern-matching)**.** The psi-calculus parameters for pattern-matching include the nominal data type $\mathbf{X}$ of (input) patterns, ranged over by X, Y, and the two equivariant operators

$$\begin{array}{rcll} \textsc{match} & : & \mathbf{T} \times \mathcal{N}^* \times \mathbf{X} \to \mathcal{P}_{\text{fin}}(\mathbf{T}^*) & \text{Pattern matching} \\ \textsc{vars} & : & \mathbf{X} \to \mathcal{P}_{\text{fin}}(\mathcal{P}_{\text{fin}}(\mathcal{N})) & \text{Pattern variables} \end{array}$$

The VARS operator gives the possible (finite) sets of names in a pattern which are bound by an input prefix. For example, we may want an input prefix with a pairing pattern $\langle x, y\rangle$ to be able to bind both x and y, only one of them, or none, and so we define $\textsc{vars}(\langle x, y\rangle) = \{\{x, y\}, \{x\}, \{y\}, \{\}\}$. This way, we can let the input prefix $\underline{c}(\lambda x)\langle x, y\rangle$ only match pairs where the second argument is the name y. To model a calculus where input patterns cannot be selective in this way, we may instead define $\textsc{vars}(\langle x, y\rangle) = \{\{x, y\}\}$. This ensures that input prefixes that use the pattern $\langle x, y\rangle$ must be of the form $\underline{M}(\lambda x, y)\langle x, y\rangle$, where both x and y are bound. Another use for VARS is to exclude the binding of terms in certain positions, such as the keys of cryptographic messages (cf. Section 5.3).

Requisites on VARS and MATCH are given below in Definition 2.5. Note that the four data types $\mathbf{T}$, $\mathbf{C}$, $\mathbf{A}$ and $\mathbf{X}$ are not required to be disjoint. In most of the examples in this paper the patterns $\mathbf{X}$ is a subset of the terms $\mathbf{T}$.

2.3. **New parameters for sorting.** To the parameters defined above we add a sorting function and four sort compatibility predicates.

Definition 2.3 (Psi-calculus parameters for sorting)**.** The psi-calculus parameters for sorting include the equivariant sorting function $\textsc{sort} : \mathcal{N} \uplus \mathbf{T} \uplus \mathbf{X} \to \mathcal{S}$, and the four compatibility predicates

$$\begin{array}{rcll} \underline{\propto} & \subseteq & \mathcal{S} \times \mathcal{S} & \text{can be used to receive,} \\ \overline{\propto} & \subseteq & \mathcal{S} \times \mathcal{S} & \text{can be used to send,} \\ \prec & \subseteq & \mathcal{S} \times \mathcal{S} & \text{can be substituted by,} \\ \mathcal{S}_\nu & \subseteq & \mathcal{S}_\mathcal{N} & \text{can be bound by name restriction.} \end{array}$$

The SORT operator gives the sort of a name, term or pattern; on names we require that $\text{SORT}(a) = s$ iff $a \in \mathcal{N}_s$. This is similar to Church-style lambda-calculi, where each well-formed term has a unique type.

The sort compatibility predicates are used to restrict where terms and names of certain sorts may appear in processes. Terms of sort s can be used to send values of sort t if $s \overline{\propto} t$. Dually, a term of sort s can be used to receive with a pattern of sort t if $s \underline{\propto} t$. A name a can be used in a restriction (νa) if $\text{SORT}(a) \in \mathcal{S}_\nu$. If $\text{SORT}(a) \prec \text{SORT}(M)$ we can substitute the term M for the name a. In most of our examples, $\prec$ is a subset of the equality relation. These predicates can be chosen freely, although the set of well-formed substitutions depends on $\prec$, as detailed in Definition 2.4 below.

2.4. **Substitution and Matching.** We require that each datatype is equipped with an equivariant substitution function, which intuitively substitutes terms for names. The requisites on substitution differ from the original psi-calculi as indicated in the Introduction. Substitutions must preserve or refine sorts, and bound pattern variables must not be removed by substitutions.

We define two usage preorders $\leq_{\mathbf{T}}$ and $\leq_{\mathbf{X}}$ on $\mathcal{S}$. Intuitively, $s_1 \leq_{\mathbf{T}} s_2$ if terms of sort s_1 can be used as a channel or message whenever s_2 can be, and $s_1 \leq_{\mathbf{X}} s_2$ if patterns of sort s_1 can be used whenever s_2 can be. Formally $s_1 \leq_{\mathbf{T}} s_2$ iff $\forall t \in \mathcal{S}.(s_2 \underline{\propto} t \Rightarrow s_1 \underline{\propto} t) \wedge (s_2 \overline{\propto} t \Rightarrow s_1 \overline{\propto} t) \wedge (t \overline{\propto} s_2 \Rightarrow t \overline{\propto} s_1)$. Similarly, we define $s_1 \leq_{\mathbf{X}} s_2$ iff $\forall t \in \mathcal{S}.(t \underline{\propto} s_2 \Rightarrow t \underline{\propto} s_1)$.

Intuitively, substitutions must map every term of sort s to a term of some sort s' with $s' \leq_{\mathbf{T}} s$ and similarly for patterns, or else a sort compatibility predicate may be violated. The usage preorders compare the sorts of terms (resp. patterns), and so do not have any formal relationship to $\prec$ (which relates the sort of a name to the sort of a term). In particular, $\prec$ is not used in the definition of usage preorders.

Definition 2.4 (Requisites on substitution)**.** If $\widetilde{a}$ is a sequence of distinct names and $\widetilde{N}$ is an equally long sequence of terms such that $\text{SORT}(a_i) \prec \text{SORT}(N_i)$ for all i, we say that $[\widetilde{a} := \widetilde{N}]$ is a *substitution*. Substitutions are ranged over by σ.

For each data type among $\mathbf{T}, \mathbf{A}, \mathbf{C}$ we define an equivariant substitution operation on members T of that data type as follows: we require that $T\sigma$ is an member of the same data type, and that if $(\widetilde{a}\ \widetilde{b})$ is a (bijective) name swapping such that $\widetilde{b}\#T, \widetilde{a}$ then $T[\widetilde{a} := \widetilde{N}] = ((\widetilde{a}\ \widetilde{b}) \cdot T)[\widetilde{b} := \widetilde{N}]$ (alpha-renaming of substituted variables). For terms we additionally require that $\text{SORT}(M\sigma) \leq_{\mathbf{T}} \text{SORT}(M)$.

For patterns $X \in \mathbf{X}$, we require that substitution is equivariant, that $X\sigma \in \mathbf{X}$, and that if $\widetilde{x} \in \text{VARS}(X)$ and $\widetilde{x}\#\sigma$ then $\text{SORT}(X\sigma) \leq_{\mathbf{X}} \text{SORT}(X)$ and $\widetilde{x} \in \text{VARS}(X\sigma)$ and alpha-renaming of substituted variables (as above) holds for σ and X.

Intuitively, the requirements on substitutions on patterns ensure that a substitution on a pattern with binders $((\lambda\widetilde{x})X)\sigma$ with $\widetilde{x} \in \text{VARS}(X)$ and $\widetilde{x}\#\sigma$ yields a pattern $(\lambda\widetilde{x})Y$ with $\widetilde{x} \in \text{VARS}(Y)$. As an example, consider the pair patterns discussed above with $\mathbf{X} = \{\langle x, y\rangle : x \neq y\}$ and $\text{VARS}(\langle x, y\rangle) = \{\{x, y\}\}$. We can let $\langle x, y\rangle\sigma = \langle x, y\rangle$ when $x, y\#\sigma$. Since $\text{VARS}(\langle x, y\rangle) = \{\{x, y\}\}$ the pattern $\langle x, y\rangle$ in a well-formed agent will always occur directly under the binder $(\lambda x, y)$, i.e. as $(\lambda x, y)\langle x, y\rangle$, and here a substitution for x or y will have no effect. It therefore does not matter what e.g. $\langle x, y\rangle[x := M]$ is, since it will never occur in derivations of transitions of well-formed agents. We could think of substitutions as partial functions which are undefined in such cases; formally, since substitutions are total, the result of this substitution can be assigned an arbitrary value.

In the original psi-calculi there is no requirement that substitution preserves names that are used as input variables (i.e., $\mathrm{n}(N\sigma) \supseteq \mathrm{n}(N) \setminus \mathrm{n}(\sigma)$). As seen in the introduction, this means that the original psi semantics does not always preserve the well-formedness of agents (an input prefix $\underline{M}(\lambda\widetilde{x})N \, . \, P$ is well-formed when $\widetilde{x} \subseteq \mathrm{n}(N)$) although this is assumed by the operational semantics [BJPV11]. In pattern-matching psi-calculi, substitution on patterns is required to preserve variables, and the operational semantics does preserve well-formedness as shown below in Theorem 2.11.

Matching must be invariant under renaming of pattern variables, and the substitution resulting from a match can only mention names that are from the matched term or the pattern.

Definition 2.5 (Requisites on pattern matching)**.** For the function MATCH we require that if $\widetilde{x} \in \textsc{vars}(X)$ are distinct and $\widetilde{N} \in \textsc{match}(M, \widetilde{x}, X)$ then it must hold that $[\widetilde{x} := \widetilde{N}]$ is a substitution, that $\mathrm{n}(\widetilde{N}) \subseteq \mathrm{n}(M) \cup (\mathrm{n}(X) \setminus \widetilde{x})$, and that for all name swappings $(\widetilde{x}\ \widetilde{y})$ with $\widetilde{y}\#X$ we have $\widetilde{N} \in \textsc{match}(M, \widetilde{y}, (\widetilde{x}\ \widetilde{y}) \cdot X)$ (alpha-renaming of matching).

In many process calculi, and also in the symbolic semantics of psi [JVP12], the input construct binds a single variable. This is a trivial instance of pattern matching where the pattern is a single bound variable, matching any term.

Example 2.6. Given values for the other requisites, we can take $\mathbf{X} = \mathcal{N}$ with $\textsc{vars}(a) = \{a\}$, meaning that the pattern variable must always occur bound, and $\textsc{match}(M, a, a) = \{M\}$ if $\textsc{sort}(a) \prec \textsc{sort}(M)$. On patterns we define substitution as $a\sigma = a$.

When all substitutions on terms preserve names, we can recover the pattern matching of the original psi-calculi. Such psi-calculi also enjoy well-formedness preservation (Theorem 2.11).

Theorem 2.7. *Suppose* $(\mathbf{T}, \mathbf{C}, \mathbf{A})$ *is an original psi-calculus* [BJPV11] *where* $\mathrm{n}(N\sigma) \supseteq \mathrm{n}(N) \setminus \mathrm{n}(\sigma)$ *for all* N, σ*. Let* $\mathbf{X} = \mathbf{T}$ *and* $\textsc{vars}(X) = \mathcal{P}(\mathrm{n}(X))$ *and* $\textsc{match}(M, \widetilde{x}, X) = \{\widetilde{L} : M = X[\widetilde{x} := \widetilde{L}]\}$ *and* $\mathcal{S} = \mathcal{S}_{\mathcal{N}} = \mathcal{S}_{\nu} = \{s\}$ *and* $\underline{\propto} = \overline{\propto} = \prec = \{(s, s)\}$ *and* $\textsc{sort} : \mathcal{N} \uplus \mathbf{T} \uplus \mathbf{X} \to \{s\}$*; then* $(\mathbf{T}, \mathbf{X}, \mathbf{C}, \mathbf{A})$ *is a sorted psi-calculus.*

Proof. Straightforward; this result has been checked in Isabelle. □

2.5. **Agents.**

Definition 2.8 (Agents)**.** The *agents*, ranged over by $P, Q, \ldots$, are of the following forms.

$\overline{M}\ N.P$	Output
$\underline{M}(\lambda\widetilde{x})X.P$	Input
$\mathbf{case}\ \varphi_1 : P_1 \ [\!]\ \cdots\ [\!]\ \varphi_n : P_n$	Case
$(\nu a)P$	Restriction
$P \mid Q$	Parallel
$!P$	Replication
$(\!\vert \Psi \vert\!)$	Assertion

In the Input all names in $\widetilde{x}$ bind their occurrences in both X and P, and in the Restriction a binds in P. Substitution on agents is defined inductively on their structure, using the substitution function of each datatype based on syntactic position, avoiding name capture.

The output prefix $\overline{M}\ N.P$ sends N on a channel that is equivalent to M. Dually, $\underline{M}(\lambda\widetilde{x})X.P$ receives a message matching the pattern X from a channel equivalent to M. A non-deterministic case statement **case** $\varphi_1 : P_1 \ [\!] \cdots [\!] \ \varphi_n : P_n$ executes one of the branches P_i where the corresponding condition φ_i holds, discarding the other branches. Restriction $(\nu a)P$ scopes the name a in P; the scope of a may be extruded if P communicates a data term containing a. A parallel composition $P \mid Q$ denotes P and Q running in parallel; they may proceed independently or communicate. A replication $!P$ models an unbounded number of copies of the process P. The assertion $(\!|\Psi|\!)$ contributes Ψ to its environment. We often write **if** φ **then** P for **case** $\varphi : P$, and nothing or $\mathbf{0}$ for the empty case statement **case**.

In comparison to [BJPV11] we additionally restrict the syntax of well-formed agents by imposing requirements on sorts: the subjects and objects of prefixes must have compatible sorts, and restrictions may only bind names of a sort in $\mathcal{S}_\nu$.

Definition 2.9. An occurrence of an assertion is *unguarded* if it is not a subterm of an Input or Output. An agent is *well-formed* if, for all its subterms,

(1) in a replication $!P$ there are no unguarded assertions in P; and
(2) in **case** $\varphi_1 : P_1 \ [\!] \cdots [\!] \ \varphi_n : P_n$ there is no unguarded assertion in any P_i; and
(3) in an Output $\overline{M}\ N.P$ we require that $\textsc{sort}(M) \ \overline{\propto}\ \textsc{sort}(N)$; and
(4) in an Input $\underline{M}(\lambda\widetilde{x})X.P$ we require that
  (a) $\widetilde{x} \in \textsc{vars}(X)$ is a tuple of distinct names and
  (b) $\textsc{sort}(M) \ \underline{\propto}\ \textsc{sort}(X)$; and
(5) in a Restriction $(\nu a)P$ we require that $\textsc{sort}(a) \in \mathcal{S}_\nu$.

Requirements 3, 4b and 5 are new for sorted psi-calculi.

2.6. **Frames and transitions.** Each agent affects other agents that are in parallel with it via its frame, which may be thought of as the collection of all top-level assertions of the agent. A *frame* F is an assertion with local names, written $(\nu\widetilde{b})\Psi$ where $\widetilde{b}$ is a sequence of names that bind into the assertion Ψ. We use F, G to range over frames, and identify alpha-equivalent frames. We overload $\otimes$ to frame composition defined by $(\nu\widetilde{b_1})\Psi_1 \otimes (\nu\widetilde{b_2})\Psi_2 = (\nu\widetilde{b_1}\widetilde{b_2})(\Psi_1 \otimes \Psi_2)$ where $\widetilde{b_1} \# \widetilde{b_2}, \Psi_2$ and vice versa. We write $\Psi \otimes F$ to mean $(\nu\epsilon)\Psi \otimes F$, and $(\nu c)((\nu\widetilde{b})\Psi)$ for $(\nu c\widetilde{b})\Psi$.

Intuitively a condition is entailed by a frame if it is entailed by the assertion and does not contain any names bound by the frame, and two frames are equivalent if they entail the same conditions. Formally, we define $F \vdash \varphi$ to mean that there exists an alpha variant $(\nu\widetilde{b})\Psi$ of F such that $\widetilde{b} \# \varphi$ and $\Psi \vdash \varphi$. We also define $F \simeq G$ to mean that for all φ it holds that $F \vdash \varphi$ iff $G \vdash \varphi$.

Definition 2.10 (Frames and Transitions)**.** The *frame* $\mathcal{F}(P)$ *of an agent* P is defined inductively as follows:

$$\mathcal{F}((\!|\Psi|\!)) = (\nu\epsilon)\Psi \qquad \mathcal{F}(P \mid Q) = \mathcal{F}(P) \otimes \mathcal{F}(Q) \qquad \mathcal{F}((\nu b)P) = (\nu b)\mathcal{F}(P)$$

$$\mathcal{F}(\underline{M}(\lambda\widetilde{x})N\,.\,P) = \mathcal{F}(\overline{M}\ N\,.\,P) = \mathcal{F}(\mathbf{case}\ \widetilde{\varphi} : \widetilde{P}) = \mathcal{F}(!P) = \mathbf{1}$$

The *actions* ranged over by α, β are of the following three kinds: Output $\overline{M}\ (\nu\widetilde{a})\ N$ where $\tilde{a} \subseteq \mathrm{n}(N)$, Input $\underline{M}\ N$, and Silent τ. Here we refer to M as the *subject* and N as the *object*. We define $\mathrm{bn}(\overline{M}\ (\nu\tilde{a})\ N) = \tilde{a}$, and $\mathrm{bn}(\alpha) = \emptyset$ if α is an input or τ. We also define $\mathrm{n}(\tau) = \emptyset$ and $\mathrm{n}(\alpha) = \mathrm{n}(M) \cup \mathrm{n}(N)$ for the input and output actions. We write $\overline{M}\langle N\rangle$ for $\overline{M}\ (\nu\varepsilon)\ N$.

$$\textsc{In}\ \frac{\Psi \vdash M \dot{\leftrightarrow} K \quad \widetilde{L} \in \textsc{match}(N, \widetilde{y}, X)}{\Psi \rhd \underline{M}(\lambda \widetilde{y})X.P \xrightarrow{K\,N} P[\widetilde{y} := \widetilde{L}]} \qquad \textsc{Out}\ \frac{\Psi \vdash M \dot{\leftrightarrow} K}{\Psi \rhd \overline{M}\,N.P \xrightarrow{\overline{K}\langle N\rangle} P}$$

$$\textsc{Com}\ \frac{\Psi_Q \otimes \Psi \rhd P \xrightarrow{\overline{M}\,(\nu\widetilde{a})\,N} P' \qquad \Psi_P \otimes \Psi \rhd Q \xrightarrow{K\,N} Q' \qquad \Psi \otimes \Psi_P \otimes \Psi_Q \vdash M \dot{\leftrightarrow} K}{\Psi \rhd P \mid Q \xrightarrow{\tau} (\nu\widetilde{a})(P' \mid Q')}\ \widetilde{a}\#Q$$

$$\textsc{Par}\ \frac{\Psi_Q \otimes \Psi \rhd P \xrightarrow{\alpha} P'}{\Psi \rhd P \mid Q \xrightarrow{\alpha} P' \mid Q}\ \mathrm{bn}(\alpha)\#Q \qquad \textsc{Case}\ \frac{\Psi \rhd P_i \xrightarrow{\alpha} P' \qquad \Psi \vdash \varphi_i}{\Psi \rhd \mathbf{case}\ \widetilde{\varphi} : \widetilde{P} \xrightarrow{\alpha} P'}$$

$$\textsc{Rep}\ \frac{\Psi \rhd P \mid !P \xrightarrow{\alpha} P'}{\Psi \rhd !P \xrightarrow{\alpha} P'} \qquad \textsc{Scope}\ \frac{\Psi \rhd P \xrightarrow{\alpha} P'}{\Psi \rhd (\nu b)P \xrightarrow{\alpha} (\nu b)P'}\ b\#\alpha, \Psi$$

$$\textsc{Open}\ \frac{\Psi \rhd P \xrightarrow{\overline{M}\,(\nu\widetilde{a})\,N} P'}{\Psi \rhd (\nu b)P \xrightarrow{\overline{M}\,(\nu\widetilde{a}\cup\{b\})\,N} P'}\ \begin{array}{l} b\#\widetilde{a}, \Psi, M \\ b \in \mathrm{n}(N) \end{array}$$

Symmetric versions of Com and Par are elided. In the rule Com we assume that $\mathcal{F}(P) = (\nu\widetilde{b_P})\Psi_P$ and $\mathcal{F}(Q) = (\nu\widetilde{b_Q})\Psi_Q$ where $\widetilde{b_P}$ is fresh for all of $\Psi, \widetilde{b_Q}, Q, M$ and P, and that $\widetilde{b_Q}$ is correspondingly fresh. In the rule Par we assume that $\mathcal{F}(Q) = (\nu\widetilde{b_Q})\Psi_Q$ where $\widetilde{b_Q}$ is fresh for Ψ, P and α. In Open the expression $\nu\tilde{a} \cup \{b\}$ means the sequence $\tilde{a}$ with b inserted anywhere.

Table 1: Operational semantics.

A *transition* is written $\Psi \rhd P \xrightarrow{\alpha} P'$, meaning that in the environment Ψ the well-formed agent P can do an α to become P'. The transitions are defined inductively in Table 1. We write $P \xrightarrow{\alpha} P'$ without an assertion to mean $\mathbf{1} \rhd P \xrightarrow{\alpha} P'$.

The operational semantics, defined in Table 1, is the same as for the original psi-calculi, except for the use of MATCH in rule In. We identify alpha-equivalent agents and transitions (see [BJPV11] for details). In a transition the names in $\mathrm{bn}(\alpha)$ bind into both the action object and the derivative, therefore $\mathrm{bn}(\alpha)$ is in the support of α but not in the support of the transition. This means that the bound names can be chosen fresh, substituting each occurrence in both the action and the derivative.

As shown in the introduction, well-formedness is not preserved by transitions in the original psi-calculi. However, in sorted psi-calculi the usual well-formedness preservation result holds.

Theorem 2.11 (Preservation of well-formedness)**.** *If P is well-formed, then*

(1) *$P\sigma$ is well-formed; and*

(2) *if $\Psi \rhd P \xrightarrow{\alpha} P'$ then P' is well-formed.*

Proof. The first part is by induction on P. The output prefix case uses the sort preservation property of substitution on terms (Definition 2.4). The interesting case is input prefix $\underline{M}(\lambda\widetilde{x})X.Q$: assume that Q is well-formed, that $\widetilde{x} \in \textsc{vars}(X)$, that $\textsc{sort}(M)\ \underline{\propto}\ \textsc{sort}(X)$

and that $\widetilde{x}\#\sigma$. By induction $Q\sigma$ is well-formed. By sort preservation we get $\textsc{sort}(M\sigma) \leq \textsc{sort}(M)$, so $\textsc{sort}(M\sigma) \underline{\propto} \textsc{sort}(X)$. By preservation of patterns by non-capturing substitutions we have that $\widetilde{x} \in \textsc{vars}(X\sigma)$ and $\textsc{sort}(X\sigma) \leq \textsc{sort}(X)$, so $\textsc{sort}(M\sigma) \underline{\propto} \textsc{sort}(X\sigma)$.

The second part is by induction on the transition rules, using part 1 in the In rule. □

Since well-formedness is preserved by transitions and substitutions, from this point on we only consider well-formed agents.

3. Meta-theory

As usual, the labelled operational semantics gives rise to notions of labelled bisimilarity. Similarly to the applied pi-calculus [AF01], the standard definition of bisimilarity needs to be adapted to take assertions into account. In this section, we show that both strong and weak bisimilarity satisfy the expected structural congruence laws and the standard congruence properties of name-passing process calculi. We first prove these results for calculi with a single name sort (Theorem 3.12) supported by Nominal Isabelle. We then extend the results to all sorted psi-caluli (Theorems 3.19, 3.20, and 3.21) by manual proofs.

3.1. Recollection.

We start by recollecting the required definitions, beginning with the definition of strong labelled bisimulation on well-formed agents by Bengtson et al. [BJPV11], to which we refer for examples and more intuitions.

Definition 3.1 (Strong bisimulation)**.** A *strong bisimulation* $\mathcal{R}$ is a ternary relation on assertions and pairs of agents such that $\mathcal{R}(\Psi, P, Q)$ implies the following four statements.

(1) Static equivalence: $\Psi \otimes \mathcal{F}(P) \simeq \Psi \otimes \mathcal{F}(Q)$.
(2) Symmetry: $\mathcal{R}(\Psi, Q, P)$.
(3) Extension with arbitrary assertion: for all Ψ' it holds that $\mathcal{R}(\Psi \otimes \Psi', P, Q)$.
(4) Simulation: for all α, P' such that $\mathrm{bn}(\alpha)\#\Psi, Q$ and $\Psi \rhd P \xrightarrow{\alpha} P'$,
there exists Q' such that $\Psi \rhd Q \xrightarrow{\alpha} Q'$ and $\mathcal{R}(\Psi, P', Q')$.

We define *bisimilarity* $P \mathrel{\dot{\sim}_\Psi} Q$ to mean that there is a bisimulation $\mathcal{R}$ such that $\mathcal{R}(\Psi, P, Q)$, and write $\dot{\sim}$ for $\dot{\sim}_{\mathbf{1}}$.

Above, (1) corresponds to the capability of a parallel observer to test the truth of a condition using **case**, while (3) models an observer taking a step and adding a new assertion Ψ' to the current environment.

We close strong bisimulation under substitutions to obtain a congruence.

Definition 3.2 (Strong bisimulation congruence)**.** $P \sim_\Psi Q$ means that for all sequences $\widetilde{\sigma}$ of substitutions it holds that $P\widetilde{\sigma} \mathrel{\dot{\sim}_\Psi} Q\widetilde{\sigma}$. We write $P \sim Q$ for $P \sim_{\mathbf{1}} Q$.

To illustrate the definitions of bisimulation and bisimulation congruence, we here prove a result about the **case** statement, to be used in Section 4.

Lemma 3.3 (Flatten Case)**.** *Suppose that there exists a condition $\top \in \mathbf{C}$ such that $\Psi \vdash \top\widetilde{\sigma}$ for all Ψ and substitution sequences $\widetilde{\sigma}$. Let $R = \mathbf{case}\ \top : (\mathbf{case}\ \widetilde{\varphi} : \widetilde{P})\ [\!]\ \widetilde{\phi} : \widetilde{Q}$ and $R' = \mathbf{case}\ \widetilde{\varphi} : \widetilde{P}\ [\!]\ \widetilde{\phi} : \widetilde{Q}$; then $R \sim R'$.*

Proof. We let $\mathcal{I} := \bigcup_{\Psi,P}\{(\Psi, P, P)\}$ be the identity relation, and

$$\mathcal{S} := \bigcup_{\Psi,\widetilde{P},\widetilde{Q},\widetilde{\phi},\widetilde{\varphi}} \{(\Psi, \mathbf{case}\ \varphi_\top : (\mathbf{case}\ \widetilde{\varphi} : \widetilde{P}) \ [\!]\ \widetilde{\phi} : \widetilde{Q}, \mathbf{case}\ \widetilde{\varphi} : \widetilde{P} \ [\!]\ \widetilde{\phi} : \widetilde{Q}) : \varphi_\top \in \mathbf{C} \wedge \forall \Psi' \in \mathbf{A}.\ \Psi' \vdash \varphi_\top\}.$$

We prove that $\mathcal{T} := \mathcal{S} \cup \mathcal{S}^{-1} \cup \mathcal{I}$ is a bisimulation, where $\mathcal{S}^{-1} := \{(\Psi, Q, P) : (\Psi, P, Q) \in \mathcal{S}\}$. Then, $\mathcal{T}(\mathbf{1}, R\widetilde{\sigma}, R'\widetilde{\sigma})$ for all $\widetilde{\sigma}$, so $R \sim R'$ by the definition of $\sim$. The proof that $\mathcal{T}$ is a bisimulation is straightforward:

Static equivalence: The frame of a **case** agent is always $\mathbf{1}$, hence static equivalence follows by reflexivity of $\simeq$.

Symmetry: Follows by definition of $\mathcal{T}$.

Extension with arbitrary assertion: Trivial by the choice of candidate relation, since the Ψ in $\mathcal{S}$ and $\mathcal{I}$ are universally quantified.

Simulation: Trivially, any process P simulates itself. Fix $(\Psi, R, R') \in \mathcal{S}$, such that $R = \mathbf{case}\ \varphi_\top : (\mathbf{case}\ \widetilde{\varphi} : \widetilde{P}) \ [\!]\ \widetilde{\phi} : \widetilde{Q}$ and $R' = \mathbf{case}\ \widetilde{\varphi} : \widetilde{P} \ [\!]\ \widetilde{\phi} : \widetilde{Q}$. Here $\Psi \vdash \varphi_\top$ follows by definition of $\mathcal{S}$. Since $\mathcal{T}$ includes both $\mathcal{S}$ and $\mathcal{S}^{-1}$, we must follow transitions from both R and R'.

- A transition from R via P_i can be derived as follows:

$$\textsc{Case}\ \dfrac{\textsc{Case}\ \dfrac{\Psi \rhd P_i \xrightarrow{\alpha} P_i' \qquad \Psi \vdash \varphi_i}{\Psi \rhd \mathbf{case}\ \widetilde{\varphi} : \widetilde{P} \xrightarrow{\alpha} P_i' \qquad \Psi \vdash \varphi_\top}}{\Psi \rhd \mathbf{case}\ \varphi_\top : (\mathbf{case}\ \widetilde{\varphi} : \widetilde{P}) \ [\!]\ \widetilde{\phi} : \widetilde{Q} \xrightarrow{\alpha} P_i'}$$

  Then R' can simulate this with the following derivation:

$$\textsc{Case}\ \dfrac{\Psi \rhd P_i \xrightarrow{\alpha} P_i' \qquad \Psi \vdash \varphi_i}{\Psi \rhd \mathbf{case}\ \widetilde{\varphi} : \widetilde{P} \ [\!]\ \widetilde{\phi} : \widetilde{Q} \xrightarrow{\alpha} P_i'}$$

  Since $\mathcal{I}(\Psi, P_i', P_i')$ and $\mathcal{I} \subseteq \mathcal{T}$ we have $\mathcal{T}(\Psi, P_i', P_i')$.
- A transition from R' via Q_i can be derived as follows:

$$\textsc{Case}\ \dfrac{\Psi \rhd Q_i \xrightarrow{\alpha} Q_i' \qquad \Psi \vdash \phi_i}{\Psi \rhd \mathbf{case}\ \widetilde{\varphi} : \widetilde{P} \ [\!]\ \widetilde{\phi} : \widetilde{Q} \xrightarrow{\alpha} Q_i'}$$

  The process R can simulate this with the following derivation:

$$\textsc{Case}\ \dfrac{\Psi \rhd Q_i \xrightarrow{\alpha} Q_i' \qquad \Psi \vdash \phi_i}{\Psi \rhd \mathbf{case}\ \varphi_\top : (\mathbf{case}\ \widetilde{\varphi} : \widetilde{P}) \ [\!]\ \widetilde{\phi} : \widetilde{Q} \xrightarrow{\alpha} Q_i'}$$

  Since $\mathcal{I}(\Psi, Q_i', Q_i')$ and $\mathcal{I} \subseteq \mathcal{T}$ we have $\mathcal{T}(\Psi, Q_i', Q_i')$.
- Symmetrically, R' can simulate transitions derived from R via Q_i, and R can simulate transitions derived from R' via P_i. □

Psi-calculi are also equipped with a notion of weak bisimilarity ($\dot{\approx}$) where τ-transitions cannot be observed, introduced by Bengtson et al. [JBPV10]. We here restate its definition, but refer to the original publication for examples and more motivation.

The definition of weak transitions is standard.

Definition 3.4 (Weak transitions)**.** $\Psi \rhd P \Longrightarrow P'$ is defined inductively by the rules:

(1) $\Psi \rhd P \Longrightarrow P$

(2) If $\Psi \rhd P \xrightarrow{\tau} P''$ and $\Psi \rhd P'' \Longrightarrow P'$, then $\Psi \rhd P \Longrightarrow P'$

For weak bisimulation we use static implication (rather than static equivalence) to compare the frames of the process pair under consideration.

Definition 3.5 (Static implication)**.** *P statically implies Q* in the environmental assertion Ψ, written $P \leq_\Psi Q$, if

$$\forall \varphi.\, \Psi \otimes \mathcal{F}(P) \vdash \varphi \;\Rightarrow\; \Psi \otimes \mathcal{F}(Q) \vdash \varphi$$

Definition 3.6 (Weak bisimulation)**.** A *weak bisimulation* $\mathcal{R}$ is a ternary relation between assertions and pairs of agents such that $\mathcal{R}(\Psi, P, Q)$ implies all of

(1) Weak static implication: for all Ψ' there exist Q', Q'' such that

$$\Psi \rhd Q \Longrightarrow Q' \quad \wedge \quad \Psi \otimes \Psi' \rhd Q' \Longrightarrow Q'' \quad \wedge \quad P \leq_\Psi Q' \quad \wedge \quad \mathcal{R}(\Psi \otimes \Psi', P, Q'')$$

(2) Symmetry: $\mathcal{R}(\Psi, Q, P)$
(3) Extension of arbitrary assertion: for all Ψ' it holds that $\mathcal{R}(\Psi \otimes \Psi', P, Q)$
(4) Weak simulation: for all P', if $\Psi \rhd P \xrightarrow{\alpha} P'$ then
  (a) if $\alpha = \tau$ then $\exists Q'.\, \Psi \rhd Q \Longrightarrow Q' \wedge \mathcal{R}(\Psi, P', Q')$; and
  (b) if $\alpha \neq \tau$ and $\mathrm{bn}(\alpha) \# \Psi, Q$, then there exists Q', Q'', Q''' such that

$$\begin{array}{l} \Psi \rhd Q \Longrightarrow Q' \quad \wedge \quad \Psi \rhd Q' \xrightarrow{\alpha} Q'' \quad \wedge \quad \Psi \otimes \Psi' \rhd Q'' \Longrightarrow Q''' \\ \wedge \quad P \leq_\Psi Q' \quad \wedge \quad \mathcal{R}(\Psi \otimes \Psi', P', Q''') \end{array}$$

We define $P \dot{\approx} Q$ to mean that there exists a weak bisimulation $\mathcal{R}$ such that $\mathcal{R}(\mathbf{1}, P, Q)$ and we write $P \dot{\approx}_\Psi Q$ when there exists a weak bisimulation $\mathcal{R}$ such that $\mathcal{R}(\Psi, P, Q)$.

Above, (1) allows Q to take τ-transitions before and after enabling at least those conditions that hold in the frame of P, as per Definition 3.5. Moreover, when testing these conditions, the observer may also add an assertion Ψ' to the environment. In (4b), the observer may test the validity of conditions when matching a visible transition, and may also add an assertion as above.

To obtain a congruence from weak bisimulation, we must require that every τ-transition is simulated by a weak transition containing at least one τ-transition.

Definition 3.7. A *weak τ-bisimulation* $\mathcal{R}$ is a ternary relation between assertions and pairs of agents such that $\mathcal{R}(\Psi, P, Q)$ implies all conditions of a weak bisimulation (Definition 3.6) with 4a replaced by

$$(4a') \text{ if } \alpha = \tau \text{ then } \exists Q', Q''.\, \Psi \rhd Q \xrightarrow{\tau} Q' \wedge \Psi \rhd Q' \Longrightarrow Q'' \wedge P' \dot{\approx}_\Psi Q''.$$

We then let $P \approx_\Psi Q$ mean that for all sequences $\widetilde{\sigma}$ of substitutions there is a weak τ-bisimulation $\mathcal{R}$ such that $\mathcal{R}(\Psi, P\widetilde{\sigma}, Q\widetilde{\sigma})$. We write $P \approx Q$ for $P \approx_{\mathbf{1}} Q$.

Lemma 3.8 (Comparing bisimulations)**.** *For all relations $\mathcal{R} \subseteq \mathbf{A} \times \mathbf{P} \times \mathbf{P}$,*

- *if $\mathcal{R}$ is a strong bisimulation then $\mathcal{R}$ is a weak τ-bisimulation.*
- *if $\mathcal{R}$ is a weak τ-bisimulation then $\mathcal{R}$ is a weak bisimulation.*

Corollary 3.9 (Comparing congruences)**.** *If $P \sim_\Psi Q$ then $P \approx_\Psi Q$.*

We seek to establish the following standard congruence and structural properties properties of strong and weak bisimulation:

Definition 3.10 (Congruence relation)**.** A relation $\mathcal{R} \subseteq \mathbf{A} \times \mathbf{P} \times \mathbf{P}$, where $(\Psi, P, Q) \in \mathcal{R}$ is written $P \ \mathcal{R}_\Psi \ Q$, is a *congruence* iff for all Ψ, $\mathcal{R}_\Psi$ is an equivalence relation, and the following implications hold.

$$
\begin{array}{llcl}
\textsc{CPar} & P \mathrel{\mathcal{R}_\Psi} Q & \Longrightarrow & (P \mid R) \mathrel{\mathcal{R}_\Psi} (Q \mid R) \\
\textsc{CRes} & a\#\Psi \wedge P \mathrel{\mathcal{R}_\Psi} Q & \Longrightarrow & (\nu a)P \mathrel{\mathcal{R}_\Psi} (\nu a)Q \\
\textsc{CBang} & P \mathrel{\mathcal{R}_\Psi} Q & \Longrightarrow & !P \mathrel{\mathcal{R}_\Psi} !Q \\
\textsc{CCase} & \forall i. P_i \mathrel{\mathcal{R}_\Psi} Q_i & \Longrightarrow & \mathbf{case} \ [] \ \widetilde{\varphi} : \widetilde{P} \ \mathrel{\mathcal{R}_\Psi} \mathbf{case} \ [] \ \widetilde{\varphi} : \widetilde{Q} \\
\textsc{COut} & P \mathrel{\mathcal{R}_\Psi} Q & \Longrightarrow & \overline{M}\, N \,.\, P \mathrel{\mathcal{R}_\Psi} \overline{M}\, N \,.\, Q \\
\textsc{CIn} & P \mathrel{\mathcal{R}_\Psi} Q & \Longrightarrow & \underline{M}(\lambda\widetilde{x})X \,.\, P \mathrel{\mathcal{R}_\Psi} \underline{M}(\lambda\widetilde{x})X \,.\, Q
\end{array}
$$

A CCase-*pseudo-congruence* is defined like a congruence, except that CIn is substituted by the following rule CIn-2.

$$
\textsc{CIn-2} \quad (\forall \widetilde{L}.\ P[\widetilde{x} := \widetilde{L}] \mathrel{\mathcal{R}_\Psi} Q[\widetilde{x} := \widetilde{L}]) \quad \Longrightarrow \quad \underline{M}(\lambda\widetilde{x})X \,.\, P \mathrel{\mathcal{R}_\Psi} \underline{M}(\lambda\widetilde{x})X \,.\, Q
$$

A *pseudo-congruence* is defined like a CCase-pseudo-congruence, but without rule CCase.

Definition 3.11 (Structural congruence)**.** *Structural congruence*, denoted $\equiv \in \mathbf{P} \times \mathbf{P}$, is the smallest relation such that $\{(\mathbf{1}, P, Q) \ : \ P \equiv Q\}$ is a congruence relation, and that satisfies the following clauses whenever $a\#Q, \widetilde{x}, M, N, X, \widetilde{\varphi}$.

$$
\begin{array}{rcl@{\qquad\qquad}rcl}
\mathbf{case} \ [] \ \widetilde{\varphi} : (\nu a)\widetilde{P} & \equiv & (\nu a)\mathbf{case} \ [] \ \widetilde{\varphi} : \widetilde{P} & !P & \equiv & P \mid !P \\
\underline{M}(\lambda\widetilde{x})X \,.\, (\nu a)P & \equiv & (\nu a)\underline{M}(\lambda\widetilde{x})X \,.\, P & P \mid (Q \mid R) & \equiv & (P \mid Q) \mid R \\
\overline{M}\, N \,.\, (\nu a)P & \equiv & (\nu a)\overline{M}\, N \,.\, P & P \mid Q & \equiv & Q \mid P \\
Q \mid (\nu a)P & \equiv & (\nu a)(Q \mid P) & P & \equiv & P \mid \mathbf{0} \\
(\nu b)(\nu a)P & \equiv & (\nu a)(\nu b)P & (\nu a)\mathbf{0} & \equiv & \mathbf{0}
\end{array}
$$

A relation $\mathcal{R} \subseteq \mathbf{P} \times \mathbf{P}$ is *complete with respect to structual congruence* if $\equiv \subseteq \mathcal{R}$.

Our goal is to establish that for all Ψ the relations $\dot{\sim}_\Psi$, $\sim_\Psi$, $\dot{\approx}_\Psi$ and $\approx_\Psi$ are complete with respect to structural congruence; that $\dot{\sim}$ is a CCase-pseudo-congruence; that $\sim$ is a congruence; that $\dot{\approx}$ is a pseudo-congruence; and that $\approx$ is a congruence.

3.2. **Psi-calculi with a single name sort.** To prove the desired algebraic properties of strong and weak bisimilarity and their induced congruences, we first adapt the Isabelle proofs for the original psi-calculi to sorted psi-calculi with a single name sort, and then manually lift the results to arbitrary sorted psi-calculi. The reason for this approach is the lack of support in Nominal Isabelle for data types that are parametric in the sorts of names.

Theorem 3.12. *If $|\mathcal{S}_\mathcal{N}| = |\mathcal{S}_\nu| = 1$, then $\dot{\sim}_\Psi$, $\sim_\Psi$, $\dot{\approx}_\Psi$ and $\approx_\Psi$ are complete wrt. structural congruence for all Ψ, $\dot{\sim}$ is a* CCase*-pseudo-congruence, $\sim$ is a congruence, $\dot{\approx}$ is a pseudo-congruence, and $\approx$ is a congruence.*

These results have all been machine-checked in Isabelle [ÅP15]. The proof scripts are adapted from Bengtson's formalisation of psi calculi [Ben10]. The same technical lemmas hold and the proof scripts are essentially identical, save for the input cases of inductive proofs, a more detailed treatment of structural congruence, and the addition of sorts and compatibility relations. We have also machine-checked Theorem 2.7 (relationship to original psi-calculi) and Theorem 2.11 (preservation of well-formedness) in this setting. These developments comprise 31909 lines of Isabelle code; Bengtson's code is 28414 lines. This represents no more than four days of work, with the bulk of the effort going towards proving

a crucial technical lemma stating that transitions do not invent new names with the new matching construct.

Isabelle is an LCF-style theorem prover, where the only trusted component is a small kernel that implements the inference rules of the logic and checks that they are correctly applied. All proofs must be fed through the kernel. Hence the results are highly trustworthy.

As indicated these proof scripts apply only to calculi with a single name sort. This restriction is a consequence of technicalities in Nominal Isabelle: it requires every name sort to be declared individually, and there are no facilities to reason parametrically over the set of name sorts.

Huffman and Urban have developed a new foundation for Nominal Isabelle that lifts the requirement to declare every name sort individually [HU10]. Unfortunately, the proof automation for reasoning about syntax quotiented by alpha-equivalence still assumes individually declared name sorts. Working around this with manually constructed quotients is possible in principle, but in practice this approach does not scale well enough to make the endeavour feasible given the size of our formalisation. A further difficulty is that Huffman and Urban's new foundation is still alpha-ware and is not backwards-compatible.

3.3. **Trivially name-sorted psi-calculi.** A *trivially name-sorted* psi-calculus is one where $\mathcal{S}_\nu = \mathcal{S}_\mathcal{N}$ and there is $S \subseteq \mathcal{S}$ such that $\prec = \mathcal{S}_\mathcal{N} \times S$, i.e., the sorts of names do not affect how they can be used for restriction and substitution.

When generalising the result for single name-sorted calculi above, the main discrepancy is that the mechanisation works with a single sort of names and thus would allow for ill-sorted alpha-renamings in the case of multiple name sorts. This is only a technicality, since every use of alpha-renaming in the formal proofs is to ensure that the bound names in patterns and substitutions avoid other bound names—thus, whenever we may work with an ill-sorted renaming, there would be a well-sorted renaming that suffices for the task.

Theorem 3.13. *In trivially name-sorted calculi, $\dot{\sim}_\Psi$, $\sim_\Psi$, $\dot{\approx}_\Psi$ and $\approx_\Psi$ are complete wrt. structural congruence for all Ψ, $\dot{\sim}$ is a* CCase*-pseudo-congruence, $\sim$ is a congruence, $\dot{\approx}$ is a pseudo-congruence, and $\approx$ is a congruence.*

Proof. By manually checking that all uses of alpha-equivalence in the proof of Theorem 3.12 admit a well-sorted alpha-renaming. □

3.4. **Arbitrary sorted psi-calculi.** We here extend the results of Theorem 3.12 to arbitrary sorted psi-calculi. The idea is to encode arbitrary sorted psi-calculi in trivially name-sorted psi-calculi by introducing an explicit error element $\bot$, resulting from application of ill-sorted substitutions. For technical reasons we must also include one extra condition `fail` (cf. Example 3.15) and in the patterns we need different error elements with different support (cf. Example 3.16).

Let I be a sorted psi-calculus with datatype parameters $\mathbf{T}_I, \mathbf{X}_I, \mathbf{C}_I, \mathbf{A}_I$. We construct a trivially name-sorted psi-calculus $U(I)$ with one extra sort, `error`, and constant symbols $\bot$ and `fail` with empty support of sort `error`, where $\bot$ is not a channel, never entailed, matches nothing and entails nothing but `fail`.

The parameters of $U(I)$ are defined by $U(I) = (\mathbf{T}_I \cup \{\bot\}, \mathbf{X}_I \cup \{(\bot, A) : A \subset_{\text{fin}} \mathcal{N}\}, \mathbf{C}_I \cup \{\bot, \mathtt{fail}\}, \mathbf{A}_I \cup \{\bot\})$. We define $\Psi \otimes \bot = \bot \otimes \Psi = \bot$ for all Ψ, and otherwise $\otimes$ is as in I. MATCH is the same in $U(I)$ as in I, plus $\text{MATCH}(M, \widetilde{x}, (\bot, S)) = \text{MATCH}(\bot, \widetilde{x}, X) = \emptyset$.

Channel equivalence $\leftrightarrow$ is the same in $U(I)$ as in I, plus $M \dot{\leftrightarrow} \bot = \bot \dot{\leftrightarrow} M = \bot$. For $\Psi \in \mathbf{A}_I$ we let $\Psi \vdash \varphi$ in $U(I)$ iff $\varphi \in \mathbf{C}_I$ and $\Psi \vdash \varphi$ in I, and we let $\bot \vdash \varphi$ iff $\varphi = \texttt{fail}$. Substitution is then defined in $U(I)$ as follows:

$$T[\widetilde{a} := \widetilde{N}]_{U(I)} := \begin{cases} T[\widetilde{a} := \widetilde{N}]_I & \text{if } \textsc{sort}(a_i) \prec_I \textsc{sort}(N_i) \text{ and} \\ & \quad N_i \neq \bot \text{ for all } i, \text{ and } T \neq (\bot, A) \\ (\bot, S \setminus \widetilde{a}) & \text{if } T = (\bot, S) \text{ is a pattern} \\ (\bot, \bigcup \textsc{vars}(T)) & \text{otherwise, if } T \text{ is a pattern} \\ \bot & \text{otherwise} \end{cases}$$

We define $\bowtie = (\mathcal{S} \times \{\texttt{error}\}) \cup (\{\texttt{error}\} \times \mathcal{S})$, and the compatibility predicates of $U(I)$ as $\underline{\propto} = \underline{\propto}_I \cup \bowtie$ and $\overline{\propto} = \underline{\propto}_I \cup \bowtie$ and $\prec = \mathcal{S}_{\mathcal{N}} \times \{s \in \mathcal{S} \ : \ \exists s' \in \mathcal{S}_{\mathcal{N}}.s' \prec_I s\}$ and $\mathcal{S}_\nu = \mathcal{S}_{\mathcal{N}}$.

Lemma 3.14. *$U(I)$ as defined above is a trivially name-sorted psi-calculus, and any well-formed process P in I is well-formed in $U(I)$.*

Proof. A straight-forward application of the definitions. □

The addition of `fail` is in order to ensure the compositionality of $\otimes$.

Example 3.15. Let $\mathbf{A} = \{1, 0\}$ and $\mathbf{C} = \{\varphi\}$ such that $\vdash = \{(1, \varphi)\}$ and $1 \otimes 0 = 1$. Now add an assertion $\bot$ such that $1 \otimes \bot = \bot$, and keep $\vdash$ unchanged. Compositionality no longer holds, since $0 \simeq \bot$, but $1 \otimes 0 = 1 \not\simeq \bot = 1 \otimes \bot$.

No variables can bind into equivariant patterns, so we need different error patterns with different support to ensure the preservation of pattern variables under substitution.

Example 3.16. Assume that the pattern X is equivariant. Then $\textsc{vars}(X) \subseteq \{\emptyset\}$.

Processes in I have the same transitions in $U(I)$.

Lemma 3.17. *If P is well-formed in I and $\Psi \neq \bot$, then $\Psi \rhd P \xrightarrow{\alpha} P'$ in $U(I)$ iff $\Psi \rhd P \xrightarrow{\alpha} P'$ in I.*

Proof. By induction on the derivation of the transitions. The cases In, Out, Case and Com use the fact that $\textsc{match}$, $\vdash$ and $\leftrightarrow$ are the same in I and $U(I)$, and that substitutions in I have the same effect when considered as substitutions in $U(I)$. □

Bisimulation in $U(I)$ coincides with bisimulation in I for processes in I.

Lemma 3.18. *Assume that P and Q are well-formed processes in I. Then $P \stackrel{.}{\sim}_\Psi Q$ in I iff $P \stackrel{.}{\sim}_\Psi Q$ in $U(I)$, and $P \stackrel{.}{\approx}_\Psi Q$ in I iff $P \stackrel{.}{\approx}_\Psi Q$ in $U(I)$.*

Proof. We show only the proof for the strong case; the weak case is similar. Let $\mathcal{R}$ be a bisimulation in $U(I)$. Then $\{(\Psi, P', Q') \in \mathcal{R} \ : \ \Psi \neq \bot \wedge P', Q' \text{ well-formed in } I\}$ is a bisimulation in I: the proof is by coinduction, using Lemma 3.17 and Theorem 2.11 in the simulation case.

Symmetrically, let $\mathcal{R}'$ be a bisimulation in I, and let $\mathcal{R}'_\bot = \{(\bot, P, Q) \ : \ \exists \Psi.(\Psi, P, Q) \in \mathcal{R}'\}$. Then $\mathcal{R}' \cup \mathcal{R}'_\bot$ is a bisimulation in $U(I)$: simulation steps from $\mathcal{R}'$ lead back to $\mathcal{R}'$ by Lemma 3.17. From $\mathcal{R}'_\bot$ there are no transitions, since $\bot$ entails no channel equivalence clauses. The other parts of Definition 3.1 are straightforward; when applying clause 3 with $\Psi' = \bot$ the resulting triple is in $\mathcal{R}'_\bot$. □

With Lemma 3.18, we can lift the structural congruence results for trivially name-sorted psi-calculi to arbitrary sorted calculi:

Theorem 3.19. *For all sorted psi-calculi, $\dot{\sim}_\Psi$, $\sim_\Psi$, $\dot{\approx}_\Psi$ and $\approx_\Psi$ are complete wrt. structural congruence for all Ψ.*

Proof. Fix a sorted psi-calculus I. For strong and weak bisimilarity, we show only the proof for commutativity of the parallel operator. The other cases are analogous.

Let P and Q be well-formed in I and $\Psi \neq \bot$. By Theorem 3.12, $P \,|\, Q \sim_\Psi Q \,|\, P$ holds in $U(I)$. By Definition 3.1, $(P|Q)\widetilde{\sigma} \dot{\sim}_\Psi (Q|P)\widetilde{\sigma}$ in $U(I)$ for all $\widetilde{\sigma}$. By Theorem 2.11, when $\widetilde{\sigma}$ is well-sorted then $(P|Q)\widetilde{\sigma}$ and $(Q|P)\widetilde{\sigma}$ are well-formed. By Lemma 3.18, $(P|Q)\widetilde{\sigma} \dot{\sim}_\Psi (Q|P)\widetilde{\sigma}$ in I for all well-sorted $\widetilde{\sigma}$. $P \,|\, Q \sim_\Psi Q \,|\, P$ in I follows by definition. $P \,|\, Q \approx_\Psi Q \,|\, P$ in I follows by Corollary 3.9. □

Using Lemma 3.18, we can also lift the congruence properties of strong and weak bisimilarity.

Theorem 3.20. *In all sorted psi-calculi, $\dot{\sim}$ is a* CCase*-pseudo-congruence and $\dot{\approx}$ is a pseudo-congruence.*

Proof. Fix a sorted psi-calculus I. We show only the proof that $\dot{\sim}$ is a congruence with respect to parallel operator, the other cases are analogous.

Assume $P \dot{\sim}_\Psi Q$ holds in I. By Lemma 3.18, $P \dot{\sim}_\Psi Q$ holds in $U(I)$. Theorem 3.12 thus yields $P \,|\, R \dot{\sim}_\Psi Q \,|\, R$ in $U(I)$, and Lemma 3.18 yields the same in I. □

Unfortunately, the approach of Theorems 3.19 and 3.20 does not work for proving congruence properties for $\sim$ or $\approx$, since the closure of bisimilarity under well-sorted substitutions does not imply its closure under ill-sorted substitutions: consider a sorted psi-calculus I such that $\mathbf{0} \sim (\!|\mathbf{1}|\!)$. Here $\mathbf{1}\sigma = \bot$ if σ is ill-sorted, but $\mathbf{0} \dot{\sim} (\!|\bot|\!)$ does not hold since only $\bot$ entails `fail`. We have instead performed a direct hand proof.

Theorem 3.21. *In all sorted psi-calculi, $\sim$ is a congruence and $\approx$ is a congruence.*

Proof. The proofs are identical, line by line, to the proofs for trivially name-sorted psi-calculi. Theorem 3.20 is used in every case. □

4. Representing Standard Process Calculi

We here consider psi-calculi corresponding to some variants of popular process calculi. One main point of our work is that we can represent other calculi directly as psi-calculi, without elaborate coding schemes. In the original psi-calculi we could in this way directly represent the monadic pi-calculus, but for the other calculi presented below a corresponding unsorted psi-calculus would contain terms with no counterpart in the represented calculus, as explained in Section 1.3. We establish that our formulations enjoy a strong operational correspondence with the original calculus, under trivial mappings that merely specialise the original concrete syntax (e.g., the pi-calculus prefix $a(x)$ maps to $\underline{a}(\lambda x)x$ in psi).

Because of the simplicity of the mapping and the strength of the correspondence we say that psi-calculi *represent* other process calculi, in contrast to *encoding* them. A representation is significantly stronger than standard correspondences, such as the approach to encodability proposed by Gorla [Gor10]. Gorla's criteria aim to capture the property that one language can encode the behaviour of another using some (possibly elaborate) protocol,

while our criteria aim to capture the property that a language for all practical purposes is a sub-language of another.

Definition 4.1. A *context* C of arity k is a psi-calculus process term with k occurrences of $\mathbf{0}$ replaced by a hole $[]$. We consider contexts as raw terms, i.e., no name occurrences are binding. The instantiation $C[P_1, \ldots, P_k]$ of a context C of arity k is the psi-calculus process resulting from the replacement of the leftmost occurrence of $[]$ with P_1, the second leftmost occurrence of $[]$ with P_2, and so on.

A psi-calculus is a *representation* of a process calculus with processes $P \in \mathcal{P}$ and labelled transition system $\rightarrow \subseteq \mathcal{P} \times \mathcal{A} \times \mathcal{P}$, if there exist an equivariant map $[\![\cdot]\!]$ from $\mathcal{P}$ to psi-calculus processes and an equivariant relation $\approxeq$ between $\mathcal{A}$ and psi-calculus actions such that

(1) $[\![\cdot]\!]$ is a simple homomorphism, i.e., for each process constructor f of $\mathcal{P}$ there is an equivariant psi-calculus context C such that $[\![f(P_1, \ldots, P_n)]\!] = C[[\![P_1]\!], \ldots, [\![P_n]\!]]$.
(2) $[\![\cdot]\!]$ is a strong operational correspondence (modulo structural equivalence), i.e.,
  (a) whenever $P \xrightarrow{\beta} P'$ then there exist α, Q such that $[\![P]\!] \xrightarrow{\alpha} Q$ and $[\![P']\!] \equiv Q$ and $\beta \approxeq \alpha$; and
  (b) whenever $[\![P]\!] \xrightarrow{\alpha} Q$ then there exist β, P' such that $P \xrightarrow{\beta} P'$ and $[\![P']\!] \equiv Q$ and $\beta \approxeq \alpha$.

A representation is *complete* if it additionally satisfies

(3) $[\![\cdot]\!]$ is surjective modulo strong bisimulation congruence, i.e., for each psi process P there is $Q \in \mathcal{P}$ such that $P \sim [\![Q]\!]$.

Any representation is a valid encoding in the sense of Gorla, but the converse is not necessarily true.

- In Gorla's approach, the contexts that process constructors are translated to may fix certain names, or translate one name into several names, in accordance with a renaming policy. We require equivariance, which admits no such special treatment of names.
- Gorla uses three criteria for semantic correspondence: weak operational correspondence modulo some equivalence for silent transitions, that the translation does not introduce divergence, and that reducibility to a success process in the source and target processes coincides. Clearly strong operational correspondence modulo structural equivalence implies all of these criteria.

Our use of structural equivalence in the operational correspondence allows to admit representations of calculi that use a structural congruence rule to define a labelled semantics (cf. Section 4.4).

Below, we use the standard notion of simultaneous substitution. Since the calculi we represent do not use environments, we let the assertions be the singleton $\{\mathbf{1}\}$ in all examples, with $\mathbf{1} \vdash \top$ and $\mathbf{1} \nvdash \bot$. Proofs of lemmas and theorems can be found in Appendix A.

4.1. **Unsorted Polyadic pi-calculus.** In the polyadic pi-calculus [Mil93] the only values that can be transmitted between agents are tuples of names. Tuples cannot be nested. The processes are defined as follows.

$$P, Q \quad ::= \quad \mathbf{0} \;\mid\; x(\tilde{y}).P \;\mid\; \overline{x}\langle\tilde{y}\rangle.P \;\mid\; [a = b]P \;\mid\; \nu x\, P \;\mid\; !P \;\mid\; P \,|\, Q \;\mid\; P + Q$$

An input binds a tuple of distinct names and can only communicate with an output of equal length, resulting in a simultaneous substitution of all names. In the unsorted polyadic pi-calculus there are no further requirements on agents, in particular $a(x).P \mid \overline{a}\langle y,z\rangle.Q$ is a valid agent. This agent has no communication action since the lengths of the tuples mismatch.

We now present the psi-calculus **PPI**, which we will show represents the polyadic pi-calculus.

PPI	
$\mathbf{T} = \mathcal{N} \cup \{\langle\widetilde{a}\rangle : \widetilde{a} \in \mathcal{N}^*\}$	$\mathcal{S} = \{\mathtt{chan}, \mathtt{tup}\}$
$\mathbf{C} = \{\top\} \cup \{a = b \mid a, b \in \mathcal{N}\}$	$\mathcal{S}_{\mathcal{N}} = \{\mathtt{chan}\}$
$\mathbf{X} = \{\langle\widetilde{a}\rangle : \widetilde{a} \in \mathcal{N}^* \wedge \widetilde{a} \text{ distinct}\}$	$\textsc{sort}(a) = \mathtt{chan}$
$\dot{\leftrightarrow} =$ identity on names	$\textsc{sort}(\langle\widetilde{a}\rangle) = \mathtt{tup}$
$\mathbf{1} \vdash a = a$	$\mathcal{S}_\nu = \{\mathtt{chan}\}$
$\textsc{vars}(\langle\widetilde{a}\rangle) = \{\widetilde{a}\}$	$\prec = \{(\mathtt{chan}, \mathtt{chan})\}$
$\textsc{match}(\langle\widetilde{a}\rangle, \widetilde{x}, \langle\widetilde{y}\rangle) = \{\widetilde{c}\}$ if $\{\widetilde{x}\} = \{\widetilde{y}\}$ and $\langle\widetilde{y}\rangle[\widetilde{x} := \widetilde{c}] = \langle\widetilde{a}\rangle$	$\overline{\propto} = \underline{\propto} = \{(\mathtt{chan}, \mathtt{tup})\}$
$\textsc{match}(M, \widetilde{x}, \langle\widetilde{y}\rangle) = \emptyset$ otherwise	

This being our first substantial example, we give a detailed explanation of the new instance parameters. Patterns $\mathbf{X}$ are finite vectors of distinct names. The sorts $\mathcal{S}$ are `chan` for channels and `tup` for tuples (of names); the only sort of names $\mathcal{S}_{\mathcal{N}}$ is channels, as is the sort of restricted names. The only sort of substitutions ($\prec$) are channels for channels; the only sort of sending ($\overline{\propto}$) and receiving ($\underline{\propto}$) is tuples over channels. In an input prefix all names in the tuple must be bound (VARS) and a vector of names $\widetilde{a}$ matches a pattern $\widetilde{y}$ if the lengths match and all names in the pattern are bound (in some arbitrary order).

As an example the agent $\underline{a}(\lambda x, y)\langle x, y\rangle \,.\, \overline{a}\ \langle y\rangle \,.\, \mathbf{0}$ is well-formed, since $\mathtt{chan} \mathrel{\underline{\propto}} \mathtt{tup}$ and $\mathtt{chan} \mathrel{\overline{\propto}} \mathtt{tup}$, with $\textsc{vars}(\langle x, y\rangle) = \{\{x, y\}\}$. This demonstrates that **PPI** disallows anomalies such as nested tuples but does not enforce a sorting discipline to guarantee that names communicate tuples of the same length.

To prove that **PPI** is a psi-calculus, we need to check the requisites on the parameters (data types and operations) defined above. Clearly the parameters are all equivariant, since no names appear free in their definitions. For the original psi-calculus parameters (Definition 2.1), the requisites are symmetry and transitivity of channel equivalence, which hold because of the same properties of (entailment of) name equality, and abelian monoid laws and compositionality for assertion composition, which trivially hold since $\mathbf{A} = \{\mathbf{1}\}$. The standard notion of simultaneous substitution of names for names preserves sorts, and also satisfies the other requirements of Definition 2.4. To check the requisites on pattern matching (Definition 2.5), it is easy to see that MATCH generates only well-sorted substitutions (of names for names), and that $\mathrm{n}(\widetilde{b}) = \mathrm{n}(\langle\widetilde{a}\rangle)$ whenever $\widetilde{b} \in \textsc{match}(\langle\widetilde{a}\rangle, \widetilde{x}, \langle\widetilde{y}\rangle)$ Finally, for all name swappings $(\widetilde{x}\ \widetilde{y})$ we have $\textsc{match}(\langle\widetilde{a}\rangle, \widetilde{x}, \langle\widetilde{z}\rangle) = \textsc{match}(\langle\widetilde{a}\rangle, \widetilde{y}, (\widetilde{x}\ \widetilde{y}) \cdot \langle\widetilde{z}\rangle)$.

PPI is a representation of the polyadic pi-calculus as presented by Sangiorgi [San93] (with replication instead of process constants).

Definition 4.2 (Polyadic Pi-Calculus to **PPI**).
Let $[\![\cdot]\!]$ be the function that maps the polyadic pi-calculus to **PPI** processes as follows. The function $[\![\cdot]\!]$ is homomorphic for $\mathbf{0}$, restriction, replication and parallel composition, and is

otherwise defined as follows:

$$\begin{array}{rcl} [\![P+Q]\!] &=& \mathbf{case}\ \top : [\![P]\!]\ [\!]\ \top : [\![Q]\!] \\ [\![[x=y]P]\!] &=& \mathbf{case}\ x=y : [\![P]\!] \\ [\![x(\tilde{y}).P]\!] &=& \underline{x}(\lambda\tilde{y})\langle\tilde{y}\rangle.[\![P]\!] \\ [\![\overline{x}\langle\tilde{y}\rangle.P]\!] &=& \overline{x}\langle\tilde{y}\rangle.[\![P]\!] \end{array}$$

Similarly, we also translate the actions of polyadic pi-calculus. Here each action corresponds to a set of psi actions, since in a pi-calculus output label "the order of the bound names is immaterial" [SW01, p. 129], which is not the case in psi-calculi.

$$\begin{array}{rcl} [\![(\nu\tilde{y})\overline{x}\langle\tilde{z}\rangle]\!] &=& \{\overline{x}\ (\nu\tilde{y}')\ \langle\tilde{z}\rangle\ :\ \tilde{y}' \text{ is a permutation of } \tilde{y}\} \\ [\![x\langle\tilde{z}\rangle]\!] &=& \{\underline{x}\ \langle\tilde{z}\rangle\} \\ [\![\tau]\!] &=& \{\tau\} \end{array}$$

Although the binders in bound output actions are ordered in psi-calculi, they can be arbitrarily reordered.

Lemma 4.3. *If $\Psi \rhd P \xrightarrow{\overline{M}\,(\nu\tilde{a})\,N} Q$ and $\tilde{c}$ is a permutation of $\tilde{a}$ then $\Psi \rhd P \xrightarrow{\overline{M}\,(\nu\tilde{c})\,N} Q$.*

Proof. By induction on the derivation of the transition. The base case is trivial. In the Open rule, we use the induction hypothesis to reorder the bound names in the premise as desired; we can then add the opened name at the appropriate position in the action in the conclusion of the rule. The other induction cases are trivial. □

We can now show that $[\![\cdot]\!]$ is a strong operational correspondence.

Theorem 4.4. *If P and Q are polyadic pi-calculus processes, then:*

(1) *If $P \xrightarrow{\beta} P'$ then for all $\alpha \in [\![\beta]\!]$ we have $[\![P]\!] \xrightarrow{\alpha} [\![P']\!]$; and*

(2) *If $[\![P]\!] \xrightarrow{\alpha} P''$ then there is β such that $P \xrightarrow{\beta} P'$ and $\alpha \in [\![\beta]\!]$ and $[\![P']\!] = P''$.*

Proof. By induction on the derivation of the transitions, using Lemma 4.3 in the **OPEN** case of (1). □

We have now shown that the polyadic pi-calculus can be embedded in **PPI**, with an embedding $[\![\cdot]\!]$ that is a strong operational correspondence.

In order to investigate surjectivity properties of the embedding $[\![\cdot]\!]$, we also define a translation $\overline{P}$ in the other direction.

Definition 4.5 (**PPi** to Polyadic Pi-Calculus)**.** The translation $\bar{\cdot}$ is homomorphic for **0**, restriction, replication and parallel composition, and is otherwise defined as follows:

$$\begin{array}{rcl} \overline{(\!|\mathbf{1}|\!)} &=& \mathbf{0} \\ \overline{\mathbf{case}\ \varphi_1 : P_1\ [\!]\ \dots\ [\!]\ \varphi_n : P_n} &=& \overline{\varphi_1 : P_1} + \cdots + \overline{\varphi_n : P_n} \\ \overline{\underline{x}(\lambda\tilde{y})\langle\tilde{z}\rangle.P} &=& x(\tilde{z}).\overline{P} \\ \overline{\overline{x}\langle\tilde{y}\rangle.P} &=& \overline{x}\langle\tilde{y}\rangle.\overline{P} \end{array}$$

where condition-guarded processes are translated as

$$\begin{array}{rcl} \overline{x=y : P} &=& [x=y]\overline{P} \\ \overline{\top : P} &=& \overline{P}. \end{array}$$

Above, note that the order of the binders in input prefixes is ignored. To show that the reverse translation is an inverse of $[\![\cdot]\!]$ modulo bisimilarity, we need to prove that their order does not matter.

Lemma 4.6. *In* **PPI**, $\underline{x}(\lambda\widetilde{y})\langle\widetilde{z}\rangle.P \sim \underline{x}(\lambda\widetilde{z})\langle\widetilde{z}\rangle.P$.

Proof. Straightforward from the definitions of MATCH and substitution on patterns. □

We now show that the embeddings $\overline{\cdot}$ and $[\![\cdot]\!]$ are inverses, modulo bisimilarity.

Theorem 4.7. *If P is a* **PPI** *process, then* $P \sim [\![\overline{P}]\!]$.

Proof. By structural induction on P. The input case uses Lemma 4.6. For **case** agents, we use an inner induction on the number of branches, with Lemma 3.3 applied in the induction case. □

Let the relation $\sim_e^c$ be early congruence of polyadic pi-calculus agents as defined in [San93]. Then we have

Corollary 4.8. *If P is a polyadic pi-calculus process, then* $P \sim_e^c \overline{[\![P]\!]}$.

We also have

Corollary 4.9. *If P and Q are polyadic pi-calculus process, then $P \sim_e^c Q$ iff* $[\![P]\!] \sim [\![Q]\!]$.

Proof. Follows from the strong operational correspondence of Theorem 4.4, and $[\![\cdot]\!]$ commuting with substitutions. □

This shows that every **PPI** process corresponds to a polyadic pi-calculus process, modulo strong bisimulation congruence, since $\overline{\cdot}$ is surjective on the bisimulation classes of polyadic pi-calculus, and the inverse of $[\![\cdot]\!]$. In other words, **PPI** is a *complete representation.*

Theorem 4.10. **PPI** *is a complete representation of the polyadic pi-calculus.*

Proof. We let $\beta \approxeq \alpha$ iff $\alpha \in [\![\beta]\!]$.
(1) $[\![\cdot]\!]$ is a simple homomorphism by definition.
(2) $[\![\cdot]\!]$ is a strong operational correspondence by Theorem 4.4.
(3) $[\![\cdot]\!]$ is surjective modulo strong bisimulation congruence by Theorem 4.7. □

4.2. **LINDA** [Gel85]**.** A process calculus with LINDA-like pattern matching can easily be obtained from the **PPI** calculus, by modifying the possible binding names in patterns.

LINDA
Everything as in **PPI** except:
$\mathbf{X} = \{\langle\widetilde{a}\rangle : \widetilde{a} \subset_{\text{fin}} \mathcal{N}\}$
$\text{VARS}(\langle\widetilde{a}\rangle) = \mathcal{P}(\widetilde{a})$
$\text{MATCH}(\langle\widetilde{a}\rangle, \widetilde{x}, \langle\widetilde{y}\rangle) = \{\widetilde{c}\}$ if $\{\widetilde{x}\} \subseteq \{\widetilde{y}\}$ and $\langle\widetilde{y}\rangle[\widetilde{x} := \widetilde{c}] = \langle\widetilde{a}\rangle$

Here, any subset of the names occurring in a pattern may be bound in the input prefix; this allows to only receive messages with particular values at certain positions (sometimes called "structured names" [Gel85]) We also do not require patterns to be linear, i.e., the same variable may occur more than once in a pattern, and the pattern only matches a tuple if each occurrence of the variable corresponds to the same name in the tuple.

As an example, $\underline{a}(\lambda x)\langle x, x, z\rangle.P \mid \overline{a}\langle c, c, z\rangle.Q \xrightarrow{\tau} P[x := c] \mid Q$ while the agent $\underline{a}(\lambda x)\langle x, x, z\rangle.P \mid \overline{a}\langle c, d, z\rangle.Q$ has no τ transition.

To prove that **LINDA** is a psi-calculus, the interesting case is the preservation of variables of substitution on patterns in Definition 2.4, i.e., that $\widetilde{x} \in \text{VARS}(\langle\widetilde{y}\rangle)$ and $\widetilde{x}\#\sigma$

implies $\widetilde{x} \in \text{VARS}(\langle\widetilde{y}\rangle\sigma)$. This holds because standard substitution preserves names and structure: there is $\widetilde{z}$ such that $\langle\widetilde{y}\rangle\sigma = \langle\widetilde{z}\rangle$, and if $x \in \widetilde{y}$ and $x\#\sigma$, then $x \in \widetilde{z}$.

4.3. **Sorted polyadic pi-calculus.** Milner's classic sorting [Mil93] regime for the polyadic pi-calculus ensures that pattern matching in inputs always succeeds, by enforcing that the length of the pattern is the same as the length of the received tuple. This is achieved as follows. Milner assumes a countable set of subject sorts S ascribed to names, and a partial function $\mathtt{ob} : \mathrm{S} \rightharpoonup \mathrm{S}^*$, assigning a sequence of object sorts to each sort in its domain. The intuition is that if a has sort s then any communication along a must be a tuple of sort $\mathtt{ob}(s)$. An agent is *well-sorted* if for any input prefix $a(b_1, \ldots b_n)$ it holds that a has some sort s where $\mathtt{ob}(s)$ is the sequence of sorts of $b_1, \ldots, b_n$ and similarly for output prefixes.

SORTEDPPI	
Everything as in **PPI** except:	
$\mathcal{S}_\mathcal{N} = \mathcal{S}_\nu = \mathrm{S}$	$\mathcal{S} = \mathrm{S} \cup \{\langle\widetilde{s}\rangle : \widetilde{s} \in \mathrm{S}^*\}$
$\prec = \{(s, s) : s \in \mathrm{S}\}$	$\overline{\propto} = \underline{\propto} = \{(s, \langle\mathtt{ob}(s)\rangle) : s \in \mathrm{S}\}$
$\text{SORT}(\langle a_1, \ldots, a_n\rangle) = \langle\text{SORT}(a_1), \ldots, \text{SORT}(a_n)\rangle$	
$\text{MATCH}(\langle\widetilde{a}\rangle, \widetilde{x}, \langle\widetilde{y}\rangle) = \{\pi \cdot \widetilde{a}\}$ if $\widetilde{x} = \pi \cdot \widetilde{y}$ and $\text{SORT}(\langle\widetilde{a}\rangle) = \text{SORT}(\langle\widetilde{y}\rangle)$	

We need to show that MATCH always generates well-sorted substitutions: this holds since whenever $\widetilde{c} \in \text{MATCH}(\langle\widetilde{a}\rangle, \widetilde{x}, \langle\widetilde{y}\rangle)$ we have that $[\widetilde{x} := \widetilde{c}] = [\pi \cdot \widetilde{y} := \pi \cdot \widetilde{a}]$ and $\text{SORT}(y_i) = \text{SORT}(a_i)$ for all i.

As an example, let $\text{SORT}(a) = s$ with $\mathtt{ob}(s) = t_1, t_2$ and $\text{SORT}(x) = t_1$ with $\mathtt{ob}(t_1) = t_2$ and $\text{SORT}(y) = t_2$ then the agent $\underline{a}(\lambda x, y)(x, y) . \overline{x}\ y . \mathbf{0}$ is well-formed, since $s \mathrel{\underline{\propto}} t_1, t_2$ and $t_1 \mathrel{\overline{\propto}} t_2$, with $\text{VARS}(x, y) = \{\{x, y\}\}$.

A formal comparison with the system in [Mil93] is complicated by the fact that Milner uses so called concretions and abstractions as agents. Restricting attention to agents in the normal sense we have the following result, where $[\![\cdot]\!]$ is the function from the previous example.

Theorem 4.11. *P is well-sorted iff $[\![P]\!]$ is well-formed.*

Proof. A trivial induction over the structure of P, observing that the requirements are identical. □

Theorem 4.12. SORTEDPPI *is a complete representation of the sorted polyadic pi-calculus.*

Proof. The operational correspondence in Theorem 4.4 still holds when restricted to well-formed agents. The inverse translation $\bar{\cdot}$ maps well-formed agents to well-sorted processes, so the surjectivity result in Theorem 4.7 still applies. □

4.4. **Polyadic synchronisation pi-calculus.** Carbone and Maffeis [CM03] explore the so called pi-calculus with polyadic synchronisation, ${}^e\pi$, which can be thought of as a dual to the polyadic pi-calculus. Here action subjects are tuples of names, while the objects transmitted are just single names. It is demonstrated that this allows a gradual enabling of communication by opening the scope of names in a subject, results in simple encodings

of localities and cryptography, and gives a strictly greater expressiveness than standard pi-calculus. The processes of ${}^e\pi$ are defined as follows.

$$\begin{array}{rcl} P, Q & ::= & \mathbf{0} \mid \Sigma_i \alpha_i.P_i \mid P \,|\, Q \mid (\nu a)P \mid !P \\ \alpha & ::= & \tilde{a}(x) \mid \tilde{a}\langle b \rangle \end{array}$$

In order to represent ${}^e\pi$, only minor modifications to the representation of the polyadic pi-calculus in Section 4.1 are necessary. To allow tuples in subject position but not in object position, we invert the relations $\overline{\propto}$ and $\underline{\propto}$. Moreover, ${}^e\pi$ does not have name matching conditions $a = b$, since they can be encoded (see [CM03]).

PSPI	
Everything as in **PPI** except:	
$\mathbf{C} = \{\top, \bot\}$	$\widetilde{a} \leftrightarrow \widetilde{b}$ is $\top$ if $\widetilde{a} = \widetilde{b}$, and $\bot$ otherwise
$\mathbf{X} = \mathcal{N}$	$\textsc{vars}(x) = \{\{x\}\}$
$\overline{\propto} = \underline{\propto} = \{(\mathtt{tup}, \mathtt{chan})\}$	$\textsc{match}(a, x, x) = \{a\}$

To obtain a representation, we consider a dialect of ${}^e\pi$ without the τ prefix. This has no cost in terms of expressiveness since the τ prefix can be encoded within ${}^e\pi$ using a communication over a restricted fresh name. However, the **PSPI** context $C[] = (\nu\, a)(\overline{\langle a \rangle}\ a.\mathbf{0} \mid \underline{\langle a \rangle}(\lambda a)a.[])$ that encodes the prefix is not admissible as part of a representation since it depends on the name a and so is not equivariant.

The ${}^e\pi$ calculus also uses an operational semantics with late input, unlike psi-calculi. In order to yield a representation, we consider an early version $\longrightarrow^e$ of the semantics, obtained by turning bound input actions into free input actions at top-level.

$$\textsc{eIn}\ \frac{P \xrightarrow{\tilde{x}(y)} P'}{P \xrightarrow{\tilde{x}\ z}{}^e\ P'\{z/y\}} \qquad \textsc{Out}\ \frac{P \xrightarrow{\tilde{x}\langle c \rangle} P'}{P \xrightarrow{\tilde{x}\langle c \rangle}{}^e\ P'} \qquad \textsc{BOut}\ \frac{P \xrightarrow{\tilde{x}\langle \nu c \rangle} P'}{P \xrightarrow{\tilde{x}\langle \nu c \rangle}{}^e\ P'} \qquad \textsc{Tau}\ \frac{P \xrightarrow{\tau} P'}{P \xrightarrow{\tau}{}^e\ P'}$$

Definition 4.13 (Polyadic synchronisation pi-calculus to **PSPI**). $[\![\cdot]\!]$ is homomorphic for $\mathbf{0}$, restriction, replication and parallel composition, and is otherwise defined as follows:

$$\begin{array}{rcl} [\![\Sigma_i \alpha_i.P_i]\!] & = & \mathbf{case}\ \top_i : [\![\alpha_i.P_i]\!] \\ [\![\widetilde{x}\langle y \rangle.P]\!] & = & \overline{\langle \widetilde{x} \rangle}\ y.[\![P]\!] \\ [\![\widetilde{x}(y).P]\!] & = & \underline{\langle \widetilde{x} \rangle}(\lambda y)y.[\![P]\!] \end{array}$$

We translate bound and free output, free input, and tau actions in the following way.

$$\begin{array}{rcl} [\![\tilde{x}\langle \nu c \rangle]\!] & = & \overline{\langle \widetilde{x} \rangle}\ (\nu c)\ c \\ [\![\tilde{x}\langle c \rangle]\!] & = & \overline{\langle \widetilde{x} \rangle}\ c \\ [\![\tilde{x}\ y]\!] & = & \underline{\langle \widetilde{x} \rangle}\ y \\ [\![\tau]\!] & = & \tau \end{array}$$

The transition system in ${}^e\pi$ is given up to structural congruence, i.e., for all α we have $\xrightarrow{\alpha} = (\equiv\xrightarrow{\alpha}\equiv)$.

Definition 4.14. $\equiv$ is the least congruence satisfying alpha conversion, the commutative monoidal laws with respect to both (|,0) and (+,0) and the following axioms[1]:

$$(\nu x)P \mid Q \equiv (\nu x)(P \mid Q) \text{ if } x\#Q \qquad (\nu x)P \equiv P \text{ if } x\#P$$

The proofs of operational correspondence are similar to the polyadic pi-calculus case. We have the following initial results for late input actions.

Lemma 4.15.

(1) *If* $P \xrightarrow{\tilde{x}(y)} P'$ *then for all* z, $[\![P]\!] \xrightarrow{\langle\tilde{x}\rangle\, z} P''$ *where* $P'' \equiv [\![P']\!][y := z]$.
(2) *If* $[\![P]\!] \xrightarrow{\langle\tilde{x}\rangle\, z} P''$ *then for all* $y\#P$, $P \xrightarrow{\tilde{x}(y)} P'$ *where* $[\![P'\{z/y\}]\!] = P''$.

Proof. By induction on the derivation of the transitions. ☐

This in turn yields the desired operational correpondence.

Theorem 4.16.

(1) *If* $P \xrightarrow{\alpha}{}^{e} P'$, *then* $[\![P]\!] \xrightarrow{[\![\alpha]\!]} P''$ *where* $P'' \equiv [\![P']\!]$.
(2) *If* $[\![P]\!] \xrightarrow{\alpha'} P''$, *then* $P \xrightarrow{\alpha}{}^{e} P'$ *where* $[\![\alpha]\!] = \alpha'$ *and* $[\![P']\!] = P''$.

Proof. By induction on the derivation of the transitions. ☐

Again, these results lead us to say that the polyadic synchronization pi-calculus can be *represented* as a psi-calculus.

Theorem 4.17. PSPI *is a representation of the polyadic synchronization pi-calculus.*

Proof. We let $\beta \approxeq \alpha$ iff $\alpha = [\![\beta]\!]$.
(1) $[\![\cdot]\!]$ is a simple homomorphism by definition.
(2) $[\![\cdot]\!]$ is a strong operational correspondence by Theorem 4.4. ☐

To investigate the surjectivity properties of $[\![\cdot]\!]$, we need to consider the fact that polyadic synchronization pi has only mixed (i.e., prefix-guarded) choice.

Definition 4.18 (Case-guarded)**.** A **PSPI** process is case-guarded if in all its subterms of the form **case** $\varphi_1 : P_1 \,[\!]\, \cdots \,[\!]\, \varphi_n : P_n$, for all $i \in \{1, \ldots, n\}$, $\varphi_i = \top$ implies $P_i = \overline{M}\ N.Q$ or $P_i = \underline{M}(\lambda\tilde{x})X.Q$.

We define the translation $\overline{R}$ from case-guarded **PSPI** processes to ${}^{e}\pi$ as the translation with the same name from **PPI**, except that $\bot$-guarded branches of **case** statements are discarded.

Theorem 4.19. *For all case-guarded* **PSPI** *processes* R *we have* $R \sim [\![\overline{R}]\!]$.

Proof. By structural induction on R. For **case** agents, we use an inner induction on the number of branches, with Lemma 3.3 applied in the induction case. ☐

Corollary 4.20. *If* P *is a polyadic synchronization pi-calculus process, then* $P \dot{\sim} \overline{[\![P]\!]}$.

Corollary 4.21. *For all* ${}^{e}\pi$ *processes* P, Q, $P \dot{\sim} Q$ *(i.e.,* P *and* Q *are early labelled congruent) iff* $[\![P]\!] \sim [\![Q]\!]$.

Proof. By strong operational correspondence 4.16, and $[\![\cdot]\!]$ commuting with substitutions. ☐

[1]The original definition of $\equiv$ [CM03] includes an additional axiom $[x = x]P \equiv P$ allowing to contract successful matches, but this axiom is omitted here since the ${}^{e}\pi$ calculus does not include the match construct. Unusually, the definition of $\equiv$ does not admit commuting restrictions, i.e., $(\nu x)(\nu y)P \not\equiv (\nu y)(\nu x)P$.

We thus have that polyadic synchronization pi corresponds to the case-guarded **PSPI** processes, modulo strong bisimulation.

4.5. **Value-passing CCS.** Value-passing CCS [Mil89] is an extension of pure CCS to admit arbitrary data from some set $\mathbf{V}$ to be sent along channels; there is no dynamic connectivity so channel names cannot be transmitted. When a value is received in a communication it replaces the input variable everywhere, and where this results in a closed expression it is evaluated, so for example $a(x) . \overline{c}(x+3)$ can receive 2 along a and become $\overline{c}\ 5$. There are conditional **if** constructs that can test if a boolean expression evaluates to true, as in $a(x) . \mathbf{if}\ x > 3\ \mathbf{then}\ P$. Formally, the value-passing CCS processes are defined by the following grammar with x, y ranging over names, v over values, b over boolean expressions, and L over sets of names.

$$P, Q \ ::= \ x(y).P \ \mid \ \overline{x}(v).P \ \mid \ \Sigma_i\ P_i \ \mid \ \mathbf{if}\ b\ \mathbf{then}\ P \ \mid \ P \setminus L \ \mid \ P \,|\, Q \ \mid \ !P \ \mid \ \mathbf{0}$$

To represent this as a psi-calculus we assume an arbitrary set of expressions $e \in \mathbf{E}$ including at least the values $\mathbf{V}$. A subset of $\mathbf{E}$ is the boolean expressions $b \in \mathbf{E_B}$. Names are either used as channels (and then have the sort `chan`) or expression variables (of sort `exp`); only the latter can appear in expressions and be substituted by values. An expression is closed if it has no name of sort `exp` in its support, otherwise it is open. The values $v \in \mathbf{V}$ are closed and have sort `value`; all other expressions have sort `exp`. The boolean values are $\mathbf{V_B} := \mathbf{V} \cap \mathbf{E_B} = \{\top, \bot\}$, and $\mathbf{1} \vdash \top$ but $\neg(\mathbf{1} \vdash \bot)$. We let E be an evaluation function on expressions, that takes each closed expression to a value and leaves open expressions unchanged. We write $e\{\widetilde{V}/\widetilde{x}\}$ for the result of syntactically replacing all $\widetilde{x}$ simultaneously by $\widetilde{V}$ in the (boolean) expression e, and assume that the result is a valid (boolean) expression. For example $(x+3)\{2/x\} = 2{+}3$, and $E(2+3) = 5$. We define substitution on expressions to use evaluation, i.e. $e[\widetilde{x} := \widetilde{V}] = E(e\{\widetilde{V}/\widetilde{x}\})$. As an example, $(x+3)[x := 2] = E((x+3)\{2/x\}) = E(2+3) = 5$. We use the single-variable patterns of Example 2.6.

VPCCS	
$\mathbf{T} = \mathcal{N} \cup \mathbf{E}$	$\mathcal{S}_\mathcal{N} = \{\mathtt{chan}, \mathtt{exp}\}$
$\mathbf{C} = \mathbf{E_B}$	$\mathcal{S} = \mathcal{S}_\mathcal{N} \cup \{\mathtt{value}\}$
$\mathbf{A} = \{\mathbf{1}\}$	$v \in \mathbf{V} \Rightarrow \textsc{sort}(v) = \mathtt{value}$
$\mathbf{X} = \mathcal{N}$	$e \in \mathbf{E} \setminus \mathbf{V} \Rightarrow \textsc{sort}(e) = \mathtt{exp}$
$a \dot{\leftrightarrow} a = \top$	$e \in \mathbf{E} \Rightarrow e[\widetilde{x} := \widetilde{M}] = E(e\{\widetilde{M}/\widetilde{x}\})$
$e \dot{\leftrightarrow} e' = \bot$ otherwise	$\prec = \{(\mathtt{exp}, \mathtt{value})\}$
$\textsc{vars}(a) = \{a\}$	$\mathcal{S}_\nu = \{\mathtt{chan}\}$
$\textsc{match}(v, a, a) = \{v\}$ if $v \in \mathbf{V}$	$\overline{\propto} = \underline{\propto} = \{(\mathtt{chan}, \mathtt{exp}), (\mathtt{chan}, \mathtt{value})\}$
$\textsc{match}(M, \widetilde{x}, a) = \emptyset$ otherwise	

Closed value-passing CCS processes correspond to **VPCCS** agents P where all free names are of sort `chan`. To prove that **VPCCS** is a psi-calculus, the interesting case is when the sort of a term is changed by substitution: let e be an open term, and σ a substitution such that $\mathrm{n}(e) \subseteq \mathrm{dom}(\sigma)$. Here $\textsc{sort}(e) = \mathtt{exp}$ and $\textsc{sort}(e\sigma) = \mathtt{value}$; this satisfies Definition 2.4 since $\mathtt{value} \leq \mathtt{exp}$ in the subsorting preorder (here $\mathtt{exp} \leq \mathtt{value}$ also holds, but is immaterial since there are no names of sort `value`).

We show that **VPCCS** represents value-passing CCS as defined by Milner [Mil89], with the following modifications:

- We use replication instead of process constants.
- We consider only finite sums. Milner allows for infinite sums without specifying exactly what infinite sets are allowed and how they are represented, making a fully formal comparison difficult. Introducing infinite sums naively in psi-calculi means that agents might exhibit cofinite support and exhaust the set of names, rendering crucial operations such as α-converting all bound names to fresh names impossible.
- We do not consider the relabelling construct $P[f]$ of CCS at all. Injective relabelings are redundant in CCS [GSV04], and the construct is not included in the psi-calculi framework.
- We only allow finite sets L in restrictions $P \setminus L$. With finite sums, this results in no loss of expressivity since agents have finite support.

Milner's restrictions are of sets of names, which we represent as a sequence of ν-binders. To create a unique such sequence from L, we assume an injective and support-preserving function $\overrightarrow{\cdot} : \mathcal{P}_{\text{fin}}(\mathcal{N}_{\mathtt{chan}}) \to (\mathcal{N}_{\mathtt{chan}})^*$. For instance, $\overrightarrow{L}$ may be defined as sorting the names in L according to some total order on $\mathcal{N}_{\mathtt{chan}}$, which is always available since $\mathcal{N}_{\mathtt{chan}}$ is countable.

The mapping $[\![\cdot]\!]$ from value-passing CCS into **VPCCS** is defined homomorphically on parallel composition, output and $\mathbf{0}$, and otherwise as follows.

$$\begin{array}{rcl} [\![x(y).P]\!] & = & \underline{x}(\lambda y)y.[\![P]\!] \\ [\![\Sigma_i\, P_i]\!] & = & \mathbf{case}\ \top : [\![P_1]\!] \ [\!]\ \cdots\ [\!]\ \top : [\![P_i]\!] \\ [\![\mathbf{if}\ b\ \mathbf{then}\ P]\!] & = & \mathbf{case}\ b : [\![P]\!] \\ [\![P \setminus L]\!] & = & (\nu \overrightarrow{L})[\![P]\!] \end{array}$$

We translate the value-passing CCS actions as follows

$$\begin{array}{rcl} [\![x(v)]\!] & = & \underline{x}\ v \\ [\![\overline{x}(v)]\!] & = & \overline{x}\ v \\ [\![\tau]\!] & = & \tau \end{array}$$

As an example, in a version of **VPCCS** where the expressions $\mathbf{E}$ include natural numbers and operations on those,

$$\begin{array}{ll} \multicolumn{2}{l}{\underline{a}(\lambda x)x\,.\,\mathbf{case}\ x > 3 : \overline{c}(x+3)} \\ \xrightarrow{\underline{a}\ 4} & (\mathbf{case}\ x > 3 : \overline{c}(x+3))[x := 4] \\ = & \mathbf{case}\ E((x>3)\{^4\!/\!_x\}) : \overline{c}(E((x+3)\{^4\!/\!_x\})) \\ = & \mathbf{case}\ E(4>3) : \overline{c}(E(4+3)) \\ = & \mathbf{case}\ \top : \overline{c}7 \\ \xrightarrow{\overline{c}\ 7} & \mathbf{0} \end{array}$$

In our psi semantics, expressions in processes are evaluated when they are closed by reception of variables (e.g. in the first transition above), while Milner simply identifies closed expressions with their values [Mil89, p55f].

Lemma 4.22. *If P is a closed* **VPCCS** *process and $P \xrightarrow{\alpha} P'$, then P' is closed.*

Theorem 4.23. *If P and Q are closed value-passing CCS processes, then*

(1) *if $P \xrightarrow{\alpha} P'$ then $[\![P]\!] \xrightarrow{[\![\alpha]\!]} [\![P']\!]$; and*

(2) *if $[\![P]\!] \xrightarrow{\alpha'} P''$ then $P \xrightarrow{\alpha} P'$ where $[\![\alpha]\!] = \alpha'$ and $[\![P']\!] = P''$.*

Proof. By induction on the derivations of P' and P'', respectively. The full proof is given in Appendix A.3. □

As before, this yields a representation theorem.

Theorem 4.24. VPCCS *is a representation of the closed agents of value-passing CCS (modulo the modifications described above).*

Proof. We let $\beta \approxeq \alpha$ iff $\alpha = [\![\beta]\!]$.
(1) $[\![\cdot]\!]$ is a simple homomorphism by definition.
(2) $[\![\cdot]\!]$ is a strong operational correspondence by Theorem 4.23. □

To investigate the surjectivity of the encoding, we let $\mathcal{P} = \{P \ : \ \textsc{sort}(\mathrm{n}(P)) \subseteq \{\mathtt{chan}\}\}$ be the **VPCCS** processes where all fre names are of channel sort.

Lemma 4.25. *If $P \in \mathcal{P}$, then there is a CCS process Q such that $P \sim [\![Q]\!]$.*

Proof. As before, we define an inverse translation $\bar{\cdot}$, that is homomorphic except for

$$\overline{\mathbf{case}\ b_1 : P_1 \ [\!]\ \cdots \ [\!]\ b_i : P_i} = (\mathbf{if}\ b_1\ \mathbf{then}\ \overline{P_1}) + \cdots + (\mathbf{if}\ b_i\ \mathbf{then}\ \overline{P_i})$$

Using Lemma 3.3, we get $P \sim [\![\overline{P}]\!]$. □

Example 4.26 (Value-passing pi-calculus)**.** To demonstrate the modularity of psi-calculi, assume that we wish a variant of the pi-calculus enriched with values in the same way as value-passing CCS. This is achieved with only a minor change to **VPCCS**:

VPPI
Everything as in **VPCCS** except:
$\textsc{match}(z, a, a) = \{z\}$ if $z \in \mathbf{V} \cup \mathcal{N}_{ch}$
$\prec = \{(\mathtt{exp}, \mathtt{value}), (\mathtt{chan}, \mathtt{chan})\}$
$\overline{\propto} = \underline{\propto} = \{(\mathtt{chan}, \mathtt{exp}), (\mathtt{chan}, \mathtt{value}), (\mathtt{chan}, \mathtt{chan})\}$

Here also channel names can be substituted for other channel names, and they can be sent and received along channel names.

5. Advanced Data Structures

We here demonstrate that we can accommodate a variety of term structures for data and communication channels; in general these can be any kind of data, and substitution can include any kind of computation on these structures. This indicates that the word "substitution" may be a misnomer — a better word may be "effect" — though we keep it to conform with our earlier work. We focus on our new contribution in the patterns and sorts, and therefore make the following definitions that are common to all the examples (unless explicitly otherwise defined).

$\mathbf{A} = \{\mathbf{1}\}$	$\mathbf{1} \otimes \mathbf{1} = \mathbf{1}$	$\mathbf{C} = \{\top, \bot\}$
$\vdash = \{(\mathbf{1}, \top)\}$	$M \leftrightarrow M = \top$	$M \leftrightarrow N = \bot$ if $M \neq N$
$\prec = \{(s, s) \ : \ s \in \mathcal{S}\}$	$\underline{\propto} = \overline{\propto} = \mathcal{S} \times \mathcal{S}$	$\mathcal{S}_\nu = \mathcal{S}_\mathcal{N} = \mathcal{S}$

If t and u are from some term algebra, we write $t \preceq u$ when t is a (non-strict) subterm of u.

5.1. **Convergent rewrite systems on terms.** In Example 4.26, the value language consisted of closed terms, with an opaque notion of evaluation. We can instead work with terms containing names and consider deterministic computations specified by a convergent rewrite system. The interesting difference is in which terms are admissible as patterns, and which choices of $\textsc{vars}(X)$ are valid. We first give a general definition and then give a concrete instance in Example 5.1.

Let Σ be a sorted signature with sorts $\mathcal{S}$, and $\cdot\Downarrow$ be normalization with respect to a convergent sort-preserving rewrite system on the nominal term algebra over $\mathcal{N}$ generated by the signature Σ. We let terms M range over the range of $\Downarrow$, i.e., the normal forms. We write ρ for sort-preserving capture-avoiding simultaneous substitutions $\{\widetilde{M}/\widetilde{a}\}$ where every M_i is in normal form; here $n(\rho) = \mathrm{n}(\widetilde{M}, \widetilde{a})$. A term M is stable if for all ρ, $M\rho\Downarrow = M\rho$. The patterns are all instances of stable terms, i.e., $X = M\rho$ where M is stable. Such a pattern X can bind any combination of names occurring in M but not in ρ. As an example, any term M is a pattern (since any name x is stable and $M = x\{M/x\}$) that can be used to match the term M itself (since $\emptyset \subseteq \mathrm{n}(x) \setminus \mathrm{n}(M, x) = \emptyset$).

REWRITE($\Downarrow$)	
$\mathbf{T} = \mathbf{X} = \mathrm{range}(\Downarrow)$	$\textsc{match}(M, \widetilde{x}, X) = \{\widetilde{L} : M = X\{\widetilde{L}/\widetilde{x}\}\}$
$M[\widetilde{y} := \widetilde{L}] = M\{\widetilde{L}/\widetilde{y}\}\Downarrow$	$\textsc{vars}(X) = \bigcup\{\mathcal{P}(\mathrm{n}(M) \setminus \mathrm{n}(\rho)) : M \text{ stable} \wedge X = M\rho\}$

We need to show that the patterns are closed under substitution, including preservation of $\textsc{vars}$ (cf. Definition 2.4), and that matching satisfies the criteria of Definition 2.5. Since any term is a pattern, the patterns are closed under substitution. Since term substitution $\{\cdot/\cdot\}$ and normalization $\Downarrow$ are both sort-preserving, term and pattern substitution $[\cdot := \cdot]$ is also sort-preserving.

To show preservation of pattern variables, assume that $\widetilde{x} \in \textsc{vars}(X)$ is a tuple of distinct names. By definition there are M and ρ such that $X = M\rho$ with M stable and $\widetilde{x} \subseteq \mathrm{n}(M) \setminus \mathrm{n}(\rho)$. Assume that $\widetilde{x}\#\sigma$; then $X\sigma = (M\rho)\sigma = M(\sigma \circ \rho)$ with $\widetilde{x}\#\sigma \circ \rho$, so $\widetilde{x} \in \textsc{vars}(X\sigma)$.

For the criteria of Definition 2.5, additionally assume that $\widetilde{L} \in \textsc{match}(N, \widetilde{x}, X)$ and let $\sigma = [\widetilde{x} := \widetilde{L}]$. Since $\{\widetilde{L}/\widetilde{x}\}$ is well-sorted, so is $[\widetilde{x} := \widetilde{L}]$. We also immediately have $\mathrm{n}(\widetilde{L}) = \mathrm{n}(N) \cup (\mathrm{n}(X) \setminus \widetilde{x})$, and alpha-renaming of matching follows from the same property for term substitution.

Example 5.1 (Peano arithmetic)**.** As a simple instance of **REWRITE**($\Downarrow$), we may consider Peano arithmetic. The rewrite rules for addition (below) induce a convergent rewrite system $\Downarrow^{\mathrm{Peano}}$, where the stable terms are those that do not contain any occurrence of `plus`.

PEANO		
Everything as in **REWRITE($\Downarrow$)** except:		
$\mathcal{S} = \{\mathtt{nat}, \mathtt{chan}\}$		
$\Sigma = \{\mathtt{zero} : \mathtt{nat},$	$\mathtt{succ} : \mathtt{nat} \to \mathtt{nat}$	$\mathtt{plus} : \mathtt{nat} \times \mathtt{nat} \to \mathtt{nat}\}$
$\mathtt{plus}(K, \mathtt{zero}) \to K$	$\mathtt{plus}(K, \mathtt{succ}(M)) \to \mathtt{plus}(\mathtt{succ}(K), M)$	
$\textsc{vars}(\mathtt{succ}^n(a)) = \{\emptyset, \{a\}\}$	$\textsc{vars}(M) = \{\emptyset\}$ otherwise	

Writing i for $\texttt{succ}^i(\texttt{zero})$, the agent $(\nu a)(\overline{a}\ 2 \mid \underline{a}(\lambda y)\texttt{succ}(y) . \overline{c}\ \texttt{plus}(3, y))$ of **REWRITE**($\Downarrow^{\text{Peano}}$) has one visible transition, with the label $\overline{c}\,4$. In particular, the object of the label is $\texttt{plus}(3, y)[y := 1] = \texttt{plus}(3, y)\{1/y\}\Downarrow^{\text{Peano}} = 4$.

5.2. **Symmetric cryptography.** We can also consider variants of **REWRITE**($\Downarrow$), such as a simple Dolev-Yao style [DY83] cryptographic message algebra for symmetric cryptography, where we ensure that the encryption keys of received encryptions can not be bound in input patterns, in agreement with cryptographic intuition.

The rewrite rule describing decryption $\texttt{dec}(\texttt{enc}(M, K), K) \rightarrow M$ induces a convergent rewrite system $\Downarrow^{\text{enc}}$, where the terms not containing $\texttt{dec}$ are stable. The construction of **REWRITE**($\Downarrow$) yields that $\widetilde{x} \in \text{VARS}(X)$ if $\widetilde{x} \subseteq \text{n}(X)$ are pair-wise different and no x_i occurs as a subterm of a $\texttt{dec}$ in X. This construction would still permit to bind the keys of an encrypted message upon reception, e.g. $\underline{a}(\lambda m, k)\texttt{enc}(m, k) . P$ would be allowed although it does not make cryptographic sense. Therefore we further restrict $\text{VARS}(X)$ to those sets not containing names that occur in key position in X, thus disallowing the binding of k above. Below we give the formal definition (recall that $\preceq$ is the subterm preorder).

SYMSPI
Everything as in **REWRITE($\Downarrow^{\text{enc}}$)** except:
$\mathcal{S} = \{\texttt{message}, \texttt{key}\}$
$\Sigma = \{\texttt{enc} : \texttt{message} \times \texttt{key} \rightarrow \texttt{message}, \quad \texttt{dec} : \texttt{message} \times \texttt{key} \rightarrow \texttt{message}\}$
$\texttt{dec}(\texttt{enc}(M, K), K) \rightarrow M$
$\text{VARS}(X) = \mathcal{P}(n(X) \setminus \{a : a \preceq \texttt{dec}(Y_1, Y_2) \preceq X \vee (a \preceq Y_2 \wedge \texttt{enc}(Y_1, Y_2) \preceq X)\})$

The proof of the conditions of Definition 2.4 and Definition 2.5 for patterns is the same as for **REWRITE**($\cdot$) in Section 5.1 above.

As an example, the agent

$$(\nu a, k)(\overline{a}\ \texttt{enc}(\texttt{enc}(M, l), k) \mid \underline{a}(\lambda y)\texttt{enc}(y, k) . \overline{c}\ \texttt{dec}(y, l))$$

has a visible transition with label $\overline{c}\,M$, where one of the leaf nodes of the derivation is

$$\underline{a}(\lambda y)\texttt{enc}(y, k) . \overline{c}\ \texttt{dec}(y, l) \xrightarrow{\underline{a}\ \texttt{enc}(\texttt{enc}(M,l),k)} \overline{c}\ \texttt{dec}(y, l)[y := \texttt{enc}(M, l)]$$

since $\texttt{enc}(M, l) \in \text{MATCH}(\texttt{enc}(\texttt{enc}(M, l), k), y, \texttt{enc}(y, k))$. The resulting process is

$$\overline{c}\ \texttt{dec}(y, l)[y := \texttt{enc}(M, l)] = \overline{c}\ \texttt{dec}(y, l)\{\texttt{enc}(M,l)/y\} \Downarrow = \overline{c}\ \texttt{dec}(\texttt{enc}(M, l), l) \Downarrow = \overline{c}\ M.$$

5.3. **Asymmetric cryptography.** A more advanced version of Section 5.2 is the treatment of data in the pattern-matching spi-calculus [HJ06], to which we refer for more examples and motivations of the definitions below. The calculus uses asymmetric encryption, and includes a non-homomorphic definition of substitution that does not preserve sorts, and a sophisticated way of computing permitted pattern variables. This example highlights the flexibility of sorted psi-calculi in that such specialized modelling features can be presented in a form that is very close to the original.

We start from the term algebra T_Σ over the unsorted signature

$$\Sigma = \{(),\ (\cdot, \cdot),\ \texttt{eKey}(\cdot),\ \texttt{dKey}(\cdot),\ \texttt{enc}(\cdot, \cdot),\ \texttt{enc}^{-1}(\cdot, \cdot)\}$$

$$\textsc{DY True}\ \frac{}{\widetilde{M} \Vdash} \qquad \textsc{DY Id}\ \frac{\widetilde{M}, N \Vdash \widetilde{L}}{\widetilde{M}, N \Vdash N, \widetilde{L}} \qquad \textsc{DY Copy}\ \frac{\widetilde{M} \Vdash N, \widetilde{L}}{\widetilde{M} \Vdash N, N, \widetilde{L}} \qquad \textsc{DY Nil}\ \frac{\widetilde{M} \Vdash \widetilde{L}}{\widetilde{M} \Vdash (), \widetilde{L}} \qquad \textsc{DY Pair}\ \frac{\widetilde{M} \Vdash N, N', \widetilde{L}}{\widetilde{M} \Vdash (N, N'), \widetilde{L}}$$

$$\textsc{DY Split}\ \frac{\widetilde{M}, N, N' \Vdash \widetilde{L}}{\widetilde{M}, (N, N') \Vdash \widetilde{L}} \qquad \textsc{DY Key}\ \frac{\widetilde{M} \Vdash N, \widetilde{L} \quad f \in \{\mathtt{eKey}, \mathtt{dKey}\}}{\widetilde{M} \Vdash f(N), \widetilde{L}} \qquad \textsc{DY Encrypt}\ \frac{\widetilde{M} \Vdash N, N', \widetilde{L}}{\widetilde{M} \Vdash \mathtt{enc}(N, N'), \widetilde{L}}$$

$$\textsc{DY Decrypt}\ \frac{\widetilde{M} \Vdash N' \qquad \widetilde{M}, N \Vdash \widetilde{L}}{\widetilde{M}, \mathtt{enc}^{-1}(N, N') \Vdash \widetilde{L}} \qquad \textsc{DY Unencrypt}\ \frac{\widetilde{M} \Vdash N' \qquad \widetilde{M}, N \Vdash \widetilde{L}}{\widetilde{M}, \mathtt{enc}(N, \mathtt{eKey}(N')) \Vdash \widetilde{L}}$$

Table 2: Dolev-Yao derivability [HJ06].

The $\mathtt{eKey}(M)$ and $\mathtt{dKey}(M)$ constructions represent the encryption and decryption parts of the key pair M, respectively. The operation $\mathtt{enc}^{-1}(M, N)$ is encryption of M with the inverse of the decryption key N, which is not an implementable operation but only permitted to occur in patterns. We add a sort system on T_Σ with sorts $\mathcal{S} = \{\mathtt{impl}, \mathtt{pat}, \bot\}$, where $\mathtt{impl}$ denotes implementable terms not containing $\mathtt{enc}^{-1}$, and $\mathtt{pat}$ those that may only be used in patterns. The sort $\bot$ denotes ill-formed terms, which do not occur in well-formed processes. Names stand for implementable terms, so we let $\mathcal{S}_\mathcal{N} = \{\mathtt{impl}\}$. Substitution is defined homomorphically on the term algebra, except to avoid unimplementable subterms on the form $\mathtt{enc}^{-1}(M, \mathtt{dKey}(N))$.

In order to define $\textsc{vars}(X)$, we write $\widetilde{M} \Vdash \widetilde{N}$ if all $N_i \in \widetilde{N}$ can be deduced from $\widetilde{M}$ in the Dolev-Yao message algebra (i.e., using cryptographic operations such as encryption and decryption). For the precise definition, see Table 2. The definition of $\textsc{vars}(X)$ below allows to bind a set S of names only if all names in S can be deduced from the message term X using the other names occurring in X. This excludes binding an unknown key (cf. Section 5.2).

PMSPI

$\mathbf{T} = \mathbf{X} = T_\Sigma \qquad \mathcal{S} = \{\mathtt{impl}, \mathtt{pat}, \bot\} \qquad \mathcal{S}_\mathcal{N} = \{\mathtt{impl}\}$

$\prec \, = \overline{\propto} = \{(\mathtt{impl}, \mathtt{impl})\} \qquad \underline{\propto} = \{(\mathtt{impl}, \mathtt{impl}), (\mathtt{impl}, \mathtt{pat})\}$

$\textsc{sort}(M) = \mathtt{impl}$ if $\forall N_1, N_2.\ \mathtt{enc}^{-1}(N_1, N_2) \not\preceq M$

$\textsc{sort}(M) = \bot$ if $\exists N_1, N_2.\ \mathtt{enc}^{-1}(N_1, \mathtt{dKey}(N_2)) \preceq M$

$\textsc{sort}(M) = \mathtt{pat}$ otherwise

$\textsc{match}(M, \widetilde{x}, X) = \{\widetilde{L} \ :\ M = X[\widetilde{x} := \widetilde{L}]\}$

$\textsc{vars}(X) = \{S \subseteq \mathrm{n}(X) \ :\ (\mathrm{n}(X) \setminus S), X \Vdash S\}$

$$
\begin{array}{rll}
x[\widetilde{y} := \widetilde{L}] &= L_i & \text{if } y_i = x \\
x[\widetilde{y} := \widetilde{L}] &= x & \text{otherwise.} \\
\mathtt{enc}^{-1}(M_1, M_2)[\widetilde{y} := \widetilde{L}] &= \mathtt{enc}(M_1[\widetilde{y} := \widetilde{L}], \mathtt{eKey}(N)) & \text{when } M_2[\widetilde{y} := \widetilde{L}] = \mathtt{dKey}(N) \\
f(M_1, \ldots, M_n)[\widetilde{y} := \widetilde{L}] &= f(M_1[\widetilde{y} := \widetilde{L}], \ldots, M_n[\widetilde{y} := \widetilde{L}]) & \text{otherwise.}
\end{array}
$$

As an example, consider the following transitions in **PMSPI**:

$$\begin{aligned}(\nu a,k,l)(\ &\overline{a}\ \mathtt{enc}(\mathtt{dKey}(l),\mathtt{eKey}(k)).\overline{a}\ \mathtt{enc}(M,\mathtt{eKey}(l))\\ &\mid \underline{a}(\lambda y)\mathtt{enc}(y,\mathtt{eKey}(k))\,.\,\underline{a}(\lambda z)\mathtt{enc}^{-1}(z,y)\,.\,\overline{c}\ z)\\ &\xrightarrow{\tau} (\nu a,k,l)(\overline{a}\ \mathtt{enc}(M,\mathtt{eKey}(l)) \mid \underline{a}(\lambda z)\mathtt{enc}(z,\mathtt{eKey}(l))\,.\,\overline{c}\ z)\\ &\xrightarrow{\tau} (\nu a,k,l)\overline{c}\ M.\end{aligned}$$

Note that $\sigma = [y := \mathtt{dKey}(l)]$ resulting from the first input changed the sort of the second input pattern: $\textsc{sort}(\mathtt{enc}^{-1}(z,y)) = \mathtt{pat}$, but $\textsc{sort}(\mathtt{enc}^{-1}(z,y)\sigma) = \textsc{sort}(\mathtt{enc}(z,\mathtt{eKey}(l))) = \mathtt{impl}$. However, this is permitted by Definition 2.4 (Substitution), since $\mathtt{impl} \leq \mathtt{pat}$ (implementable terms can be used as channels or messages whenever patterns can be).

Terms (and patterns) are trivially closed under substitution. All terms in the domain of a well-sorted substitution have sort $\mathtt{impl}$, so well-sorted substitutions cannot introduce subterms of the forms $\mathtt{enc}^{-1}(N_1,N_2)$ or $\mathtt{enc}^{-1}(N_1,\mathtt{dKey}(N_2))$ where none existed; thus $\textsc{sort}(M\sigma) \leq \textsc{sort}(M)$ as required by Definition 2.4.

To show preservation of pattern variables, we first need some technical results about Dolev-Yao derivability.

Lemma 5.2.
(1) *If* $\widetilde{M} \Vdash \widetilde{N}$, *then* $\widetilde{M'}\widetilde{M} \Vdash \widetilde{N}$.
(2) *If* $\widetilde{M} \Vdash \widetilde{N}$, *then* $\widetilde{M}\sigma \Vdash \widetilde{N}\sigma$.
(3) *If* $\textsc{sort}(N) = \mathtt{impl}$, *then* $\mathrm{n}(N) \Vdash N$.
(4) *If* $\widetilde{M}, N \Vdash \widetilde{L}$ *and* $\textsc{sort}(N) = \mathtt{impl}$ *and* $\widetilde{M} \Vdash N$, *then* $\widetilde{M} \Vdash \widetilde{L}$.

Lemma 5.3 (Preservation of pattern variables)**.**
If $\widetilde{x}\#\sigma$ *and* $(\mathrm{n}(X) \setminus \widetilde{x}), X \Vdash \widetilde{x}$ *then* $(\mathrm{n}(X\sigma) \setminus \widetilde{x}), X\sigma \Vdash \widetilde{x}$.

Proof. Let $\widetilde{M} = (\mathrm{n}(X)\setminus\widetilde{x})\sigma$. By Lemma 5.2(2) we get $\widetilde{M}, X\sigma \Vdash \widetilde{x}$, so $(\mathrm{n}(X\sigma)\setminus\widetilde{x}), \widetilde{M}, X\sigma \Vdash \widetilde{x}$ by Lemma 5.2(1). Since $\mathrm{n}(\widetilde{M}) = (\mathrm{n}(X\sigma)\setminus\widetilde{x})$, Lemma 5.2(3) yields that $(\mathrm{n}(X\sigma)\setminus\widetilde{x}) \Vdash \widetilde{M}$. Finally, by Lemma 5.2(4) we get $(\mathrm{n}(X\sigma) \setminus \widetilde{x}), X\sigma \Vdash \widetilde{x}$. □

The requisites on matching (Definition 2.5) follow from those on substitution. Lemma 5.3 implies that the set of (well-sorted) processes [HJ06] is closed under (well-sorted) substitution, a result which appears not to have been published previously.

5.4. **Nondeterministic computation.** The previous examples considered total deterministic notions of computation on the term language. Here we consider a data term language equipped with partial non-deterministic evaluation: a lambda calculus extended with the erratic choice operator $\cdot \mathbin{[]} \cdot$ and the reduction rule $M_1 \mathbin{[]} M_2 \to M_i$ if $i \in \{1,2\}$. Due to non-determinism and partiality, evaluation cannot be part of the substitution function. Instead, we define the MATCH function to collect all evaluations of the received term, which are non-deterministically selected from by the IN rule. This example also highlights the use of object languages with binders, a common application of nominal logic.

We let substitution on terms be the usual capture-avoiding syntactic replacement, and define reduction contexts $\mathcal{R} ::= [\,] \mid \mathcal{R}\ M \mid (\boldsymbol{\lambda} x.M)\ \mathcal{R}$ (we here use the boldface $\boldsymbol{\lambda}$ rather than the λ used in input prefixes). Reduction $\to$ is the smallest pre-congruence for reduction contexts that contain the rules for β-reduction ($\boldsymbol{\lambda} x.M\ N \to M[x := N]$) and $\cdot \mathbin{[]} \cdot$ (see above).

We use the single-name patterns of Example 2.6, but include evaluation in matching.

NDLAM
$\mathcal{S} = \{s\}$ $\quad\quad$ $\mathbf{X} = \mathcal{N}$
$M ::= a \mid M\ M \mid \boldsymbol{\lambda} x.M \mid M \,[\!]\, M$ $\quad$ where x binds into M in $\boldsymbol{\lambda} x.M$
$\textsc{match}(M, x, x) = \{N\ :\ M \rightarrow^* N \not\rightarrow\}$
$\textsc{match}(M, \widetilde{y}, x) = \emptyset$ otherwise

To avoid confusing the λ of the input prefix and the $\boldsymbol{\lambda}$ of the term language, we write $\underline{a}(x)$ for $\underline{a}(\lambda x)x$. As an example, the agent $P \stackrel{def}{=} (\nu a)(\underline{a}(y) \,.\, \overline{c}\ y \,.\, \mathbf{0} \mid \overline{a}\ ((\boldsymbol{\lambda} x.x\ x) \,[\!]\, (\boldsymbol{\lambda} x.x)) \,.\, \mathbf{0})$ has the following transitions:

$$P \xrightarrow{\tau} (\nu a)(\overline{c}\ \boldsymbol{\lambda} x.xx \,.\, \mathbf{0} \mid \mathbf{0}) \xrightarrow{\overline{c}\ \boldsymbol{\lambda} x.xx} \mathbf{0}$$
$$P \xrightarrow{\tau} (\nu a)(\overline{c}\ \boldsymbol{\lambda} x.x \,.\, \mathbf{0} \mid \mathbf{0}) \xrightarrow{\overline{c}\ \boldsymbol{\lambda} x.x} \mathbf{0}.$$

6. Conclusions and further work

We have described two features that taken together significantly improve the precision of applied process calculi: generalised pattern matching and substitution, which allow us to model computations on an arbitrary data term language, and a sort system which allows us to remove spurious data terms from consideration and to ensure that channels carry data of the appropriate sort. The well-formedness of processes is thereby guaranteed to be preserved by transitions. Using these features we have provided representations of other process calculi, ranging from the simple polyadic pi-calculus to the spi-calculus and non-deterministic computations, in the psi-calculi framework. The critera for representation (rather than encoding) are stronger than standard correspondences e.g. by Gorla, and mean that the psi-calculus and the process calculus that it represents are for all practical purposes one and the same.

The meta-theoretic results carry over from the original psi formulations, and have been machine-checked in Isabelle for the case of a single name sort (e.g. the calculi **PPI**, **LINDA** and **PSPI** in Section 4, and the calculi **PMSPI** and **NDLAM** in Section 5). We have also added sorts to an existing tool for psi-calculi [BGRV15], the Psi-calculi Workbench (Pwb), which provides an interactive simulator and automatic bisimulation checker. Users of the tool need only implement the parameters of their psi-calculus instances, supported by a core library. In the tool we currently support only tuple patterns, similarly to the **PPI** calculus of Section 4.1.

Future work includes developing a symbolic semantics with more elaborate pattern matching. For this, a reformulation of the operational semantics of Table 1 in the late style, where input objects are not instantiated until communication takes place, is necessary.

A comparison of expressiveness to calculi with non-binary (e.g., join-calculus [FG96] or Kell calculus) or bidirectional (e.g., dyadic interaction terms [Hon93] or the concurrent pattern calculus [GWGJ10]) communication primitives would be interesting. We here inherit positive results from the pi calculus, such as the encoding of the join-calculus.

We aim to extend the use of sorts and generalized pattern matching to other variants of psi-calculi, including higher-order psi calculi [PBRÅP13] and reliable broadcast psi-calculi [ÅPBP+13]. Although assertions and conditions are unsorted, we intend to investigate adding sorts and pattern-matching to psi-calculi with non-trivial assertions [BJPV11].

As discussed in Section 3.2, further work is needed for scalable mechanised reasoning about theories that are parametric in an arbitrary but fixed name sorting.

Acknowledgments. We thank the anonymous reviewers for their helpful comments.

Appendix A. Full proofs for Section 4

We will assume that the reader is acquainted with the relevant psi-calculi presented in Section 4, as well as the definitions, notation and terminology of Sangiorgi [San93] for polyadic pi-calculus, Carbone and Maffeis [CM03] for polyadic synchronisation pi-calculus, and Milner [Mil89] for CCS and VPCCS. We will use their notation except for bound names, where we will adopt the notation of nominal sets, e.g., we will write $\mathrm{bn}(\alpha)\#Q$ instead of $\mathrm{bn}(\alpha) \cap \mathrm{fn}(Q) = \emptyset$.

A.1. **Polyadic Pi-Calculus.**

This section contains full proofs of Section 4.1 for the polyadic pi-calculus example. We use definitions and results of Sangiorgi [San93]. However, we opted to replace process constants with replication.

For convenience, we repeat definition of the encoding function given in Example 4.1.

Definition A.1 (Polyadic Pi-Calculus to **PPi**)**.**
Agents:

$$\begin{array}{rcl}
\llbracket P+Q \rrbracket &=& \mathbf{case}\ \top : \llbracket P \rrbracket \ [\!]\ \top : \llbracket Q \rrbracket \\
\llbracket [x=y]P \rrbracket &=& \mathbf{case}\ x = y : \llbracket P \rrbracket \\
\llbracket x(\tilde{y}).P \rrbracket &=& \underline{x}(\lambda\tilde{y})\langle\tilde{y}\rangle.\llbracket P \rrbracket \\
\llbracket \overline{x}\langle\tilde{y}\rangle.P \rrbracket &=& \overline{x}\langle\tilde{y}\rangle.\llbracket P \rrbracket \\
\llbracket 0 \rrbracket &=& 0 \\
\llbracket P \mid Q \rrbracket &=& \llbracket P \rrbracket \mid \llbracket Q \rrbracket \\
\llbracket \nu x P \rrbracket &=& (\nu x)\llbracket P \rrbracket \\
\llbracket !P \rrbracket &=& !\llbracket P \rrbracket
\end{array}$$

Actions:

$$\begin{array}{rcl}
\llbracket (\nu\tilde{y}')\overline{z}\langle\tilde{y}\rangle \rrbracket &=& \overline{z}\,(\nu\tilde{y}')\,\langle\tilde{y}\rangle \\
\llbracket x\langle\tilde{z}\rangle \rrbracket &=& \underline{x}\,\langle\tilde{z}\rangle \\
\llbracket \tau \rrbracket &=& \tau
\end{array}$$

In the output action $\tilde{y}'$ bind into $\tilde{y}$ and the residual process, but not into z.

Definition A.2 (**PPi** to Polyadic Pi-Calculus)**.**
Process:

$$\begin{array}{rcl}
\overline{(\!|\mathbf{1}|\!)} &=& \mathbf{0} \\
\overline{\mathbf{0}} = \overline{\mathbf{case}} &=& \mathbf{0} \\
\overline{\mathbf{case}\ \varphi_1 : P_1 \ [\!]\ \dots\ [\!]\ \varphi_n : P_n} &=& \overline{\varphi_1 : P_1} + \cdots + \overline{\varphi_n : P_n} \\
\overline{!P} &=& !\overline{P} \\
\overline{(\nu x)P} &=& \nu x \overline{P} \\
\overline{P \mid Q} &=& \overline{P} \mid \overline{Q} \\
\overline{\underline{x}(\lambda\tilde{y})\langle\tilde{y}\rangle.P} &=& x(\tilde{y}).\overline{P} \\
\overline{\overline{x}\langle\tilde{y}\rangle.P} &=& \overline{x}\langle\tilde{y}\rangle.\overline{P}
\end{array}$$

Case clause:

$$\begin{array}{rcl}\overline{x = y : P} & = & [x = y]\overline{P} \\ \overline{\top : P} & = & \overline{P}\end{array}$$

We prove that the substitution function distributes over the encoding function.

Lemma A.3. $\llbracket P\rrbracket[\tilde y := \tilde z] = \llbracket P\{\tilde z/\tilde y\}\rrbracket$

Proof. By induction on P. We consider only the agents where $\mathrm{bn}(P)\#P\{\tilde z/\tilde y\}$ [San93, Definition 2.1.1]. We demonstrate the non-trivial cases of the proof in the following.

- case $P = P' + Q$.

$$\begin{array}{rcll}\llbracket P' + Q\rrbracket[\tilde y := \tilde z] & = & \mathbf{case}\ \top[\tilde y := \tilde z] : \llbracket P'\rrbracket[\tilde y := \tilde z] \ [\,]\ \top[\tilde y := \tilde z] : \llbracket Q\rrbracket[\tilde y := \tilde z] & \\ & = & \mathbf{case}\ \top : \llbracket P'\rrbracket[\tilde y := \tilde z] \ [\,]\ \top : \llbracket Q\rrbracket[\tilde y := \tilde z] & \\ & = & \mathbf{case}\ \top : \llbracket P'\{\tilde z/\tilde y\}\rrbracket \ [\,]\ \top : \llbracket Q\{\tilde z/\tilde y\}\rrbracket & \text{(IH)} \\ & = & \llbracket P'\{\tilde z/\tilde y\} + Q\{\tilde z/\tilde y\}\rrbracket & \\ & = & \llbracket (P' + Q)\{\tilde z/\tilde y\}\rrbracket & \end{array}$$

- case $P = [x = y]Q$.

$$\begin{array}{rcll}\llbracket [x = y]Q\rrbracket[\tilde y := \tilde z] & = & \mathbf{case}\ x[\tilde y := \tilde z] = y[\tilde y := \tilde z] : \llbracket Q\rrbracket[\tilde y := \tilde z] & \\ & = & \mathbf{case}\ x[\tilde y := \tilde z] = y[\tilde y := \tilde z] : \llbracket Q\{\tilde z/\tilde y\}\rrbracket & \text{(IH)} \\ & = & [x\{\tilde z/\tilde y\} = y\{\tilde z/\tilde y\}]\llbracket Q\{\tilde z/\tilde y\}\rrbracket & \\ & = & \llbracket ([x = y]Q)\{\tilde z/\tilde y\}\rrbracket & \end{array}$$

- case $P = a(\tilde x).Q$

$$\begin{array}{rcll}\llbracket a(\tilde x).Q\rrbracket[\tilde y := \tilde z] & = & \underline{a[\tilde y := \tilde z]}(\lambda\widetilde{x})\langle\widetilde{x}\rangle.\llbracket Q\rrbracket[\tilde y := \tilde z] & \text{(From assumption } \tilde x\#[\tilde y := \tilde z]) \\ & = & \underline{a[\tilde y := \tilde z]}(\lambda\widetilde{x})\langle\widetilde{x}\rangle.\llbracket Q\{\tilde z/\tilde y\}\rrbracket & \text{(IH)} \\ & = & a\{\tilde z/\tilde y\}(\tilde x).\llbracket Q\{\tilde z/\tilde y\}\rrbracket & \\ & = & \llbracket (a(\tilde x).Q)\{\tilde z/\tilde y\}\rrbracket & \end{array}$$

□

The following is the proof of the strong operational correspondence with respect to the labeled semantics of polyadic pi-calculus [San93, page 30].

Proof of Theorem 4.4.

(1) We show that if $P \xrightarrow{\beta} P'$ then for all $\alpha \in \llbracket\beta\rrbracket$ we have $\llbracket P\rrbracket \xrightarrow{\alpha} \llbracket P'\rrbracket$ by induction on the derivation of the transition.

ALP:

Trivial, since psi-calculi processes are identified up to alpha equivalence.

OUT:

Assume $\overline{x}\langle\tilde y\rangle.P \xrightarrow{\overline{x}\langle\tilde y\rangle} P$ and $\alpha \in \{\overline{x}\ \langle\tilde y\rangle\} = \llbracket\overline{x}\langle\tilde y\rangle\rrbracket$. Since $\mathbf{1} \vdash x \dot\leftrightarrow x$ and $\llbracket\overline{x}\ \langle\tilde y\rangle.P\rrbracket = \overline{x}\ \langle\tilde y\rangle.\llbracket P\rrbracket$ and $\alpha = \overline{x}\ \langle\tilde y\rangle$, we can derive $\overline{x}\ \langle\tilde y\rangle.\llbracket P\rrbracket \xrightarrow{\overline{x}\ \langle\tilde y\rangle} \llbracket P\rrbracket$.

INP:

Assume $x(\tilde y).P \xrightarrow{x\langle\tilde z\rangle} P\{\tilde z/\tilde y\}$, and $\tilde z$ and $\tilde y$ are of the same arity (in the trminology of Sangiorgi, $\tilde z : \tilde y$), and also $\alpha \in \llbracket\beta\rrbracket = \{\underline{x}\ \langle\tilde z\rangle\}$. Note that $\llbracket x(\tilde y).P\rrbracket = \underline{x}(\lambda\widetilde{y})\langle\widetilde{y}\rangle.\llbracket P\rrbracket$ and $\widetilde{z} \in \textsc{match}(\langle\widetilde{z}\rangle, \widetilde{y}, \langle\widetilde{y}\rangle)$. By using $\mathbf{1} \vdash x \dot\leftrightarrow x$, we can derive $\underline{x}(\lambda\widetilde{y})\langle\widetilde{y}\rangle.\llbracket P\rrbracket \xrightarrow{\underline{x}\ \langle\tilde z\rangle} \llbracket P\rrbracket[\widetilde{y} := \widetilde{z}]$ with the In rule. By applying Lemma A.3, we complete this proof case.

SUM:

Assume $P+Q \xrightarrow{\beta} P'$ and $\alpha \in [\![\beta]\!]$, and also $P \xrightarrow{\beta} P'$. The induction hypothesis is that for every $\alpha \in [\![\beta]\!]$, $[\![P]\!] \xrightarrow{\alpha} [\![P']\!]$. We can then derive $\mathbf{case}\ \top : [\![P]\!]\ [\!]\ \top : [\![Q]\!] \xrightarrow{\alpha} [\![P']\!]$ with the Case rule for every $\alpha \in [\![\beta]\!]$.

PAR:

Assume $P \mid Q \xrightarrow{\beta} P' \mid Q$ and $\alpha \in [\![\beta]\!]$, and $P \xrightarrow{\beta} P'$ with $\mathrm{bn}(\beta) \cap \mathrm{fn}(Q) = \emptyset$. The induction hypothesis is that for every $\alpha \in [\![\beta]\!]$, $[\![P]\!] \xrightarrow{\alpha} [\![P']\!]$. From the definition of $[\![\beta]\!]$ we get that $\mathrm{bn}(\alpha)\#[\![Q]\!]$ for any $\alpha \in [\![\beta]\!]$. By applying the Par rule, we obtain the required transitions $[\![P]\!] \mid [\![Q]\!] \xrightarrow{\alpha} [\![P']\!] \mid [\![Q]\!]$.

COM:

Assume $P \mid Q \xrightarrow{\tau} \nu\tilde{y}'(P' \mid Q')$ with $\tilde{y}' \cap \mathrm{fn}(Q) = \emptyset$. Also assume $P \xrightarrow{(\nu\tilde{y'})\overline{x}\langle\tilde{y}\rangle} P'$ and $Q \xrightarrow{x\langle\tilde{y}\rangle} Q'$. The induction hypothesis is that for every $\alpha' \in [\![(\nu\tilde{y'})\overline{x}\langle\tilde{y}\rangle]\!]$ and $\alpha'' \in [\![x\langle\tilde{y}\rangle]\!]$, $[\![P]\!] \xrightarrow{\alpha'} [\![P']\!]$ and $[\![Q]\!] \xrightarrow{\alpha''} [\![Q']\!]$ Moreover, we note that $\mathbf{1} \vdash x \leftrightarrow x$ and $\tilde{y}'\#[\![Q]\!]$. We then choose α' and α'' and alpha-variants of the frames of $[\![P]\!]$ and $[\![Q]\!]$ that are sufficiently fresh to allow the derivation $[\![P]\!] \mid [\![Q]\!] \xrightarrow{\tau} (\nu\widetilde{y'})([\![P']\!] \mid [\![Q']\!])$ with the Com rule.

MATCH:

Assume $[x = x]P \xrightarrow{\beta} P'$ and $\alpha \in [\![\beta]\!]$, as well as $P \xrightarrow{\beta} P'$. The induction hypothesis is that $[\![P]\!] \xrightarrow{\alpha} [\![P']\!]$. Since $\mathbf{1} \vdash x = x$ and $\mathbf{case}\ x = x : [\![P]\!] = [\![[x = x]P]\!]$, we derive $\mathbf{case}\ x = x : [\![P]\!] \xrightarrow{\alpha} [\![P']\!]$ with the Case rule.

REP:

Assume $!P \xrightarrow{\beta} P'$ and $\alpha \in [\![\beta]\!]$. Moreover, assume $P \mid !P \xrightarrow{\beta} P'$ and hence by the induction hypothesis $[\![P \mid !P]\!] \xrightarrow{\alpha} [\![P']\!]$. We compute $[\![P]\!] \mid ![\![P]\!] = [\![P \mid !P]\!]$ and apply the Rep rule to obtain $![\![P]\!] \xrightarrow{\alpha} [\![P']\!]$.

RES:

Assume $\nu xP \xrightarrow{\beta} \nu xP'$ where $x \notin n(\beta)$ and $\alpha \in [\![\beta]\!]$. Also assume $P \xrightarrow{\beta} P'$. The induction hypothesis is $[\![P]\!] \xrightarrow{\alpha} [\![P']\!]$. Now by obtaining $x\#\alpha$ from assumptions and computing $[\![\nu xP]\!] = (\nu x)[\![P]\!]$, we derive $(\nu x)[\![P]\!] \xrightarrow{\alpha} (\nu x)[\![P']\!]$ with the Scope rule.

OPEN:

Let $\beta = (\nu x, \tilde{y'})\overline{z}\langle\tilde{y}\rangle$. Assume $\nu xP \xrightarrow{\beta} P'$ and $x \neq z, x \in \tilde{y} - \tilde{y'}$ and $\alpha \in [\![\beta]\!] = \{\overline{z}\,(\nu\tilde{y}'')\,\langle\tilde{y}\rangle \;:\; \tilde{y}'' = \pi \cdot x, \tilde{y}'\}$. The induction hypothesis is that for every $\alpha' \in [\![(\nu\tilde{y}')\overline{z}\langle\tilde{y}\rangle]\!] = \{\overline{z}\,(\nu\tilde{y}'')\,\langle\tilde{y}\rangle \;:\; \tilde{y}'' = \pi \cdot \tilde{y'}\}$ we have $[\![P]\!] \xrightarrow{\alpha'} [\![P']\!]$. We choose $\alpha' = \overline{z}\,(\nu\tilde{y}')\,\tilde{y}$ and, by having $[\![\nu xP]\!] = (\nu x)[\![P]\!]$, we derive $(\nu x)[\![P]\!] \xrightarrow{\overline{z}\,(\nu x, \tilde{y}')\,\langle\tilde{y}\rangle} [\![P']\!]$ with the Open rule. The side conditions of Open ($x\#\tilde{y}', z$ and $x \in n(\tilde{y})$) follow from assumptions.

From the assumption $\alpha \in [\![\beta]\!]$, it follows that, for any permutation π, α is of the form $\overline{z}\,(\nu\pi \cdot x, \tilde{y}')\,\langle\tilde{y}\rangle$. By applying Lemma 4.3, we get the required α and transition $(\nu x)[\![P]\!] \xrightarrow{\alpha} [\![P']\!]$. And this concludes this proof case.

(2) We now show that if $[\![P]\!] \xrightarrow{\alpha} P''$ then $P \xrightarrow{\beta} P'$ where $\alpha \in [\![\beta]\!]$ and $[\![P']\!] = P''$. We proceed by by induction on the derivation of the transition. We show the interesting cases:

Case:
Assume $[\![P]\!] \xrightarrow{\alpha} P''$. By inversion of the CASE rule, $[\![P]\!]$ is of the form **case** $\tilde{\varphi} : \tilde{P}$. Since $P_C =$ **case** $\widetilde{\varphi} : \widetilde{P}$ is in the range of $[\![\cdot]\!]$, either $P_C = \top : [\![P]\!] \;[\!]\; \top : [\![Q]\!]$, $P_C = \top : [\![Q]\!] \;[\!]\; \top : [\![P]\!]$ or $P_C =$ **case** $x = y : [\![P]\!]$. We proceed by case analysis:
(a) When $P_C = \top : [\![P]\!] \;[\!]\; \top : [\![Q]\!]$, we note that $[\![P + Q]\!] = P_C$ and imitate the derivation of P'' from P_C with the derivation $P + Q \xrightarrow{\beta} P'$, using the **SUM** rule and the fact obtained from induction hypothesis $\alpha \in [\![\beta]\!]$.
(b) The case when $P_C = \top : [\![Q]\!] \;[\!]\; \top : [\![P]\!]$ is symmetric to the previous case.
(c) When $P_C =$ **case** $x = y : [\![P]\!]$, since $\mathbf{1} \vdash x = y$ by the induction hypothesis, $x = y$. We note that $[\![[x = x]P]\!] = P_C$ and imitate the derivation of P'' from P_C with the derivation $[x = x]P \xrightarrow{\beta} P'$, using the **MATCH** rule and the fact obtained from induction hypothesis $\alpha \in [\![\beta]\!]$.

Open:
Assume $[\![P]\!] \xrightarrow{\overline{z}\,(\nu\tilde{y} \cup \{x\})\,\langle\tilde{y}'\rangle} P''$. Because P'' is derived with the OPEN rule, $[\![P]\!]$ is of the form $(\nu x)R$. Since $(\nu x)R$ is in the range of $[\![\cdot]\!]$, $P = \nu x R'$, where $R = [\![R']\!]$. From induction hypothesis, we have that $R \xrightarrow{\overline{z}\,(\nu\tilde{y})\,\langle\tilde{y}'\rangle} P''$ and $\overline{z}\,(\nu\tilde{y})\,\langle\tilde{y}'\rangle \in [\![\beta']\!]$ and $R' \xrightarrow{\beta'} P'$ and lastly $[\![P']\!] = P''$. Thus, we use $\beta' = (\nu\tilde{y})\overline{z}\langle\tilde{y}'\rangle$ as it gives us $\overline{z}\,(\nu\tilde{y})\,\langle\tilde{y}'\rangle \in [\![\beta']\!]$ to derive, by using the rule **OPEN**, $\nu x R' \xrightarrow{(\nu x,\tilde{y})\overline{z}\langle\tilde{y}'\rangle} P'$. Clearly, $\overline{z}\,(\nu\tilde{y} \cup \{x\})\,\langle\tilde{y}'\rangle \in [\![(\nu x, \tilde{y})\overline{z}\langle\tilde{y}'\rangle]\!]$ for every insertion of x. □

From the strong operational correspondence, we obtain full abstraction. We use Sangiorgi's definition of bisimulation and congruence for the polyadic pi-calculus [San93, page 42].

Theorem A.4. *For polyadic-pi calculus agents P and Q we have $P \sim_e^c Q$ iff $[\![P]\!] \sim [\![Q]\!]$.*

Proof. For direction $\Leftarrow$, assume $[\![P]\!] \sim [\![Q]\!]$. We claim that the relation $\mathcal{R} = \{(P, Q) : [\![P]\!] \sim [\![Q]\!]\}$ is an *early congruence* in the polyadic pi-calculus.

Firs let us consider the simulation case. Assume $P \xrightarrow{\beta} P'$. Then, we need to show that there exists Q' such that $Q \xrightarrow{\beta} Q'$ and $(P', Q') \in \mathcal{R}$. By Theorem 4.4 (1), we get $[\![P]\!] \xrightarrow{\alpha} [\![P']\!]$ for any $\alpha \in [\![\beta]\!]$. By Theorem 4.4 (2) and using the assumption $\alpha \in [\![\beta]\!]$ as well as the fact $[\![P]\!] \sim [\![Q]\!]$, we derive $[\![Q]\!] \xrightarrow{\alpha} [\![Q']\!]$. From the simulation clause and that $[\![P]\!]$ and $[\![Q]\!]$ are congruent we get that $[\![P']\!] \sim [\![Q']\!]$. Hence, $(P', Q') \in \mathcal{R}$. The symmetry case follows from the symmetry of $\sim$. Thus, $\mathcal{R}$ is an early bisimulation. Since $\mathcal{R}$ is closed under all substitutions by Lemma A.3, it is also an early congruence.

Now let us consider the other direction $\Rightarrow$. First, assume $P \sim_e^c Q$. We claim the relation $\{(\mathbf{1}, [\![P]\!], [\![Q]\!]) : P \sim_e^c Q\}$ is a congruence in **PPI**. The static equivalence and extension of arbitrary assertion cases are trivial since there is unit assertion only. Symmetry follows from symmetry of $\sim_e^c$, and simulation follows by Theorem 4.4 and the fact that $\sim_e^c$ is an early congruence. □

Proof of Theorem 4.7. By structural induction on P. We only consider the **case** agent since the other cases are trivial.

$P = \mathbf{case}\ \varphi_1 : P_1 \mathbin{[\!]} \dots \mathbin{[\!]} \varphi_n : P_n$**:**

We have one induction hypothesis IH_i for every $i \in \{1..n\}$, namely that $P_i \sim \llbracket \overline{P_i} \rrbracket$.

We proceed by induction on n.

Base case $n = 0$**:**

$\llbracket \overline{\mathbf{case}} \rrbracket = \llbracket \mathbf{0} \rrbracket = \mathbf{0}$. By reflexivity of $\sim$, $\mathbf{0} \sim \mathbf{0}$.

Induction step $n + 1$**:**

The IH for this case is

$$\llbracket \overline{\mathbf{case}\ \varphi_1 : P_1 \mathbin{[\!]} \dots \mathbin{[\!]} \varphi_n : P_n} \rrbracket \sim \mathbf{case}\ \varphi_1 : P_1 \mathbin{[\!]} \dots \mathbin{[\!]} \varphi_n : P_n = P'$$

We need to show that $Q \sim \llbracket \overline{Q} \rrbracket$ for $Q = \mathbf{case}\ \varphi_1 : P_1 \mathbin{[\!]} \dots \mathbin{[\!]} \varphi_n : P_n \mathbin{[\!]} \varphi_{n+1} : P_{n+1}$.
We thus compute

$$\begin{aligned}
\llbracket \overline{Q} \rrbracket &= \llbracket \overline{\varphi_1 : P_1} + \dots + \overline{\varphi_n : P_n} + \overline{\varphi_{n+1} : P_{n+1}} \rrbracket \\
&= \mathbf{case}\ \top : \llbracket \overline{\varphi_1 : P_1} \rrbracket \mathbin{[\!]} \dots \mathbin{[\!]} \top : \llbracket \overline{\varphi_n : P_n} \rrbracket \mathbin{[\!]} \top : \llbracket \overline{\varphi_{n+1} : P_{n+1}} \rrbracket \\
&\sim \text{(by Lemma 3.3)} \\
&\ \mathbf{case}\ \top : (\mathbf{case}\ \top : \llbracket \overline{\varphi_1 : P_1} \rrbracket \mathbin{[\!]} \dots \mathbin{[\!]} \top : \llbracket \overline{\varphi_n : P_n} \rrbracket) \mathbin{[\!]} \top : \llbracket \overline{\varphi_{n+1} : P_{n+1}} \rrbracket \\
&\sim \text{(by IH)} \\
&\ \mathbf{case}\ \top : (\mathbf{case}\ \varphi_1 : P_1 \mathbin{[\!]} \dots \mathbin{[\!]} \varphi_n : P_n) \mathbin{[\!]} \top : \llbracket \overline{\varphi_{n+1} : P_{n+1}} \rrbracket \\
&= \mathbf{case}\ \top : P' \mathbin{[\!]} \top : \llbracket \overline{\varphi_{n+1} : P_{n+1}} \rrbracket \\
&= Q'
\end{aligned}$$

We distinguish two cases of φ_{n+1}:

Case $\varphi_{n+1} = \top$**:**

$$\begin{aligned}
Q' &= \mathbf{case}\ \top : P' \mathbin{[\!]} \top : \llbracket \overline{\top : P_{n+1}} \rrbracket \\
&= \mathbf{case}\ \top : P' \mathbin{[\!]} \top : \llbracket \overline{P_{n+1}} \rrbracket \\
&\sim \text{(by } \mathrm{IH}_{n+1}\text{)} \\
&\ \mathbf{case}\ \top : P' \mathbin{[\!]} \top : P_{n+1} \\
&\sim \text{(by Lemma 3.3)} \\
&\ \mathbf{case}\ \varphi_1 : P_1 \mathbin{[\!]} \dots \mathbin{[\!]} \varphi_n : P_n \mathbin{[\!]} \top : P_{n+1} = Q
\end{aligned}$$

We conclude this case.

Case $\varphi_{n+1} = x = y$**:**

$$\begin{aligned}
Q' &= \mathbf{case}\ \top : P' \mathbin{[\!]} \top : \llbracket \overline{x = y : P_{n+1}} \rrbracket \\
&= \mathbf{case}\ \top : P' \mathbin{[\!]} \top : (\mathbf{case}\ x = y : \llbracket \overline{P_{n+1}} \rrbracket) \\
&\sim \text{(by } \mathrm{IH}_{n+1}\text{)} \\
&\ \mathbf{case}\ \top : P' \mathbin{[\!]} \top : (\mathbf{case}\ x = y : P_{n+1}) \\
&\sim \text{(by Lemma 3.3)} \\
&\ \mathbf{case}\ \varphi_1 : P_1 \mathbin{[\!]} \dots \mathbin{[\!]} \varphi_n : P_n \mathbin{[\!]} \top : (\mathbf{case}\ x = y : P_{n+1}) \\
&\sim \text{(by Lemma 3.3)} \\
&\ \mathbf{case}\ \varphi_1 : P_1 \mathbin{[\!]} \dots \mathbin{[\!]} \varphi_n : P_n \mathbin{[\!]} x = y : P_{n+1} = Q
\end{aligned}$$

By concluding this case, we conclude the proof. □

Lemma A.5. $\llbracket \cdot \rrbracket$ *is injective, that is, for all* P, Q*, if* $\llbracket P \rrbracket = \llbracket Q \rrbracket$ *then* $P = Q$*.*

Proof. By induction on P and Q while inspecting all possible cases. □

Lemma A.6. $[\![\cdot]\!]$ *is surjective up to* $\sim$*, that is, for every* P *there is a* Q *such that* $[\![Q]\!] \sim P$*.*

Proof. By induction on the well-formed agent P.

Case $\underline{x}(\lambda\widetilde{y})\langle\widetilde{y}\rangle.P'$**:**
By induction there is Q' such that $[\![Q']\!] \sim P'$. Let $Q = x(\tilde{y}).Q'$. Then $[\![Q]\!] = [\![x(\tilde{y}).Q']\!] = \underline{x}(\lambda\widetilde{y})\langle\widetilde{y}\rangle.[\![Q']\!] \sim \underline{x}(\lambda\widetilde{y})\langle\widetilde{y}\rangle.P' = P$.

Case $\overline{x}\langle\tilde{y}\rangle.P'$**:**
By induction there is Q' such that $[\![Q']\!] \sim P'$. Let $Q = \overline{x}\langle\tilde{y}\rangle.Q'$. Now $[\![Q]\!] = \overline{x}\langle\tilde{y}\rangle.[\![Q']\!] \sim \overline{x}\langle\tilde{y}\rangle.P' = P$.

Case $P \mid P'$**:**
By induction there are Q', Q'' such that $[\![Q']\!] \sim P$ and $[\![Q'']\!] \sim P'$. Then let $Q = Q' \mid Q''$, obtaining $[\![Q]\!] = [\![Q']\!] \mid [\![Q'']\!] \sim P \mid P' = P$.

Case $(\nu x)P$**:**
By induction there is Q' such that $[\![Q']\!] \sim P$. Let $Q = \nu x Q'$. Then $[\![Q]\!] = (\nu x)[\![Q']\!] \sim (\nu x)P$.

Case $!P$**:**
By induction there is Q' such that $[\![Q']\!] \sim P$. Let $Q = !Q'$. Then $[\![Q]\!] = ![\![Q']\!] \sim !P$.

Case $(\!|\mathbf{1}|\!)$**:**
Let $Q = \mathbf{0}$. Then $[\![Q]\!] = \mathbf{0} \sim (\!|\mathbf{1}|\!)$.

Case $\textbf{case}\ \widetilde{\varphi} : \widetilde{P'}$**:**
The induction hypothesis $\mathrm{IH}_{\textbf{case}}$ is that for every P_i' there is Q_i' such that $[\![Q_i']\!] \sim P_i'$. The proof proceeds by induction on the length of $\widetilde{\varphi}$.

Base case:
Let $Q = \mathbf{0}$, then $[\![Q]\!] = \mathbf{0} \sim \textbf{case}$.

Induction step:
At this step, we get the following IH

$$[\![Q'']\!] \sim \textbf{case}\ \varphi_1 : P_1 \ [\!] \ \dots \ [\!] \ \varphi_n : P_n$$

We need to find $[\![Q]\!]$ such that

$$[\![Q]\!] \sim \textbf{case}\ \varphi_1 : P_1 \ [\!] \ \dots \ [\!] \ \varphi_n : P_n \ [\!] \ \varphi_{n+1} : P_{n+1}$$

By $\mathrm{IH}_{\textbf{case}}$ for P_{n+1}' we get $[\![Q_{n+1}']\!] \sim P_{n+1}$. We proceed by case analysis on φ_{n+1}.

Case $\varphi_{n+1} = \top$**:**
Let $Q = Q'' + Q_{n+1}'$. Then

$$\begin{array}{rcl}
[\![Q]\!] & = & \textbf{case}\ \top : [\![Q'']\!] \ [\!] \ \top : [\![Q_{n+1}']\!] \\
& \sim & \textbf{case}\ \top : (\textbf{case}\ \varphi_1 : P_1 \ [\!] \ \dots \ [\!] \ \varphi_n : P_n) \\
& & [\!] \ \top : [\![Q_{n+1}']\!] \\
& \sim & \textbf{case}\ \top : (\textbf{case}\ \varphi_1 : P_1 \ [\!] \ \dots \ [\!] \ \varphi_n : P_n) \\
& & [\!] \ \top : P_{n+1} \\
& \sim & \text{(by Lemma 3.3)} \\
& & \textbf{case}\ \varphi_1 : P_1 \ [\!] \ \dots \ [\!] \ \varphi_n : P_n \\
& & [\!] \ \top : P_{n+1}
\end{array}$$

Case $\varphi_{n+1} = x = y$:
Let $Q = Q'' + [x = y]Q'_{n+1}$. Then

$$
\begin{array}{rcl}
[\![Q]\!] & = & \mathbf{case}\ \top : [\![Q'']\!]\ [\!]\ \top : [\![[x = y]Q'_{n+1}]\!] \\
& \sim & \mathbf{case}\ \top : (\mathbf{case}\ \varphi_1 : P_1\ [\!]\ \ldots\ [\!]\ \varphi_n : P_n) \\
& & [\!]\ \top : (\mathbf{case}\ x = y : [\![Q'_{n+1}]\!]) \\
& \sim & \mathbf{case}\ \top : (\mathbf{case}\ \varphi_1 : P_1\ [\!]\ \ldots\ [\!]\ \varphi_n : P_n) \\
& & [\!]\ \top : (\mathbf{case}\ x = y : P_{n+1}) \\
& \sim & \text{(by Lemma 3.3)} \\
& & \mathbf{case}\ \varphi_1 : P_1\ [\!]\ \ldots\ [\!]\ \varphi_n : P_n \\
& & [\!]\ \top : (\mathbf{case}\ x = y : P_{n+1}) \\
& \sim & \text{(by permuting and applying Lemma 3.3)} \\
& & \mathbf{case}\ \varphi_1 : P_1\ [\!]\ \ldots\ [\!]\ \varphi_n : P_n\ [\!]\ x = y : P_{n+1}
\end{array}
$$

This case concludes the proof. □

A.2. **Polyadic Synchronisation Pi-Calculus.** In this section, we include the full proofs of Section 4.4. We use definitions and results for polyadic synchronisation pi-calculus, ${}^{e}\pi$, by Carbone and Maffeis [CM03].

We give an explicit definition of encoding function defined in Example 4.4.

Definition A.7 (Polyadic synchronisation pi-calculus to **PSPi**)**.**
Agents:

$$
\begin{array}{rcl}
[\![\widetilde{x}(y).P]\!] & = & \underline{\underline{\langle\widetilde{x}\rangle}}(\lambda y)y.[\![P]\!] \\
[\![\widetilde{x}\langle y\rangle.P]\!] & = & \overline{\langle\widetilde{x}\rangle}\ y.[\![P]\!] \\
[\![P \mid Q]\!] & = & [\![P]\!] \mid [\![Q]\!] \\
[\![(\nu x)P]\!] & = & (\nu x)[\![P]\!] \\
[\![!P]\!] & = & ![\![P]\!] \\
[\![0]\!] & = & 0 \\
[\![\Sigma_i \alpha_i.P_i]\!] & = & \mathbf{case}\ \top_i : [\![\alpha_i.P_i]\!]
\end{array}
$$

Actions:

$$
\begin{array}{rcl}
[\![\tilde{x}\langle\nu c\rangle]\!] & = & \overline{\langle\tilde{x}\rangle}\ (\nu c)\ c \\
[\![\tilde{x}\langle c\rangle]\!] & = & \overline{\langle\tilde{x}\rangle}\ c \\
[\![\tau]\!] & = & \tau \\
[\![\tilde{x}(y)]\!] & = & \text{undefined}
\end{array}
$$

Definition A.8 (**PSPi** to Polyadic synchronisation pi-calculus)**.**

$$
\begin{array}{rcl}
\overline{(\!|\mathbf{1}|\!)} & = & \mathbf{0} \\
\overline{\mathbf{0}} & = & \mathbf{0} \\
\overline{!P} & = & !\overline{P} \\
\overline{(\nu x)P} & = & (\nu x)\overline{P} \\
\overline{P \mid Q} & = & \overline{P} \mid \overline{Q} \\
\overline{\langle\widetilde{a}\rangle y.P} & = & \overline{a}\langle y\rangle.\overline{P} \\
\overline{\underline{\widetilde{x}}(\lambda y)y.P} & = & \overline{x}(y).\overline{P} \\
\overline{\tau.P} & = & \tau.\overline{P} \\
\overline{\mathbf{case}\ \top : \alpha_i.P_i} & = & \Sigma_i\overline{\alpha_i.P_i}
\end{array}
$$

Lemma A.9. *If $P \equiv Q$ then $[\![P]\!] \sim [\![Q]\!]$*

Proof. The relation $\mathcal{R} = \{(P, Q) : [\![P]\!] \sim [\![Q]\!]\}$ satisfies the axioms defining $\equiv$ and is also a process congruence. Since $\equiv$ is the least such congruence, $\equiv \subseteq \mathcal{R}$. □

Proof of Lemma 4.15.

(1) By induction on the derivation of P', avoiding z.

Prefix:

Here $\Sigma_i \tilde{x}_i(y_i).P_i \xrightarrow{\tilde{x}_i(y_i)} P_i$. We have that

$$[\![\Sigma_i \tilde{x}_i(y_i).P_i]\!] \;=\; \mathbf{case}\ \top : \underline{\langle\tilde{x}\rangle}(\lambda y_1)y_1.[\![P_1]\!] \ [\!]\ \cdots\ [\!]\ \top : \underline{\langle\tilde{x}\rangle}(\lambda y_i)y_i.[\![P_i]\!]$$

Since $\textsc{match}(z, \langle y_i\rangle, y_i) = \{z\}$, we can use the Case and In rules to derive the transition

$$\mathbf{case}\ \top : \underline{\langle\tilde{x}_1\rangle}(\lambda y_1)y_1.[\![P_1]\!] \ [\!]\ \cdots\ [\!]\ \top : \underline{\langle\tilde{x}_i\rangle}(\lambda y_i)y_i.[\![P_i]\!] \xrightarrow{\langle\tilde{x}\rangle\, z} [\![P_i]\!][y_i := z]$$

Finally, we have $P'' = [\![P_i]\!][y_i := z]$ and use reflexivity of $\sim$ to conclude this case.

Bang:

Here $P \mid !P \xrightarrow{\tilde{x}(y)} P'$ and by induction, $[\![P]\!] \mid ![\![P]\!] \xrightarrow{\langle\tilde{x}\rangle\, z} P''$ with $P'' \sim [\![P']\!][y := z]$. By rule Rep, we also have that $![\![P]\!] \xrightarrow{\langle\tilde{x}\rangle\, z} P''$.

Par:

Here $P \xrightarrow{\tilde{x}(y)} P'$, $y\#Q$ and by induction, $[\![P]\!] \xrightarrow{\langle\tilde{x}\rangle\, z} P''$ with $P'' \sim [\![P']\!][y := z]$. Using the Par rule we derive $[\![P]\!] \mid [\![Q]\!] \xrightarrow{\langle\tilde{x}\rangle\, z} P' \mid [\![Q]\!]$. Since $\sim$ is closed under $\mid$, $P'' \mid [\![Q]\!] \sim [\![P']\!][y := z] \mid [\![Q]\!]$. Finally, since $y\#Q$, $[\![P']\!][y := z] \mid [\![Q]\!] = [\![P' \mid Q]\!][y := z]$.

Struct:

Here $P \equiv Q$, $Q \xrightarrow{\tilde{x}(y)} Q'$ and $Q' \equiv P'$. By induction we obtain Q'' such that $[\![Q]\!] \xrightarrow{\langle\tilde{x}\rangle\, z} Q''$ where $Q'' \sim [\![Q']\!][y := z]$. By Lemma A.9, $[\![P]\!] \sim [\![Q]\!]$ and $[\![Q']\!] \sim [\![P']\!]$, and by expanding the definition of $\sim$, we obtain $[\![Q']\!][y := z] \sim [\![P']\!][y := z]$. Since $[\![P]\!] \sim [\![Q]\!]$ and $[\![Q]\!] \xrightarrow{\langle\tilde{x}\rangle\, z} Q''$, there exists P'' such that $[\![P]\!] \xrightarrow{\langle\tilde{x}\rangle\, z} P''$ and $Q'' \sim P''$. By using the transitivity of $\sim$, we conclude $P'' \sim [\![P']\!][y := z]$.

Res:

Here $P \xrightarrow{\tilde{x}(y)} P'$, $a \neq y$, $a \neq z$ and $a\#\tilde{x}$. By induction, $[\![P]\!] \xrightarrow{\langle\tilde{x}\rangle\, z} P''$ with $P'' \sim [\![P']\!][y := z]$. We can then derive $(\nu a)[\![P]\!] \xrightarrow{\langle\tilde{x}\rangle\, z} (\nu a)P''$. Since $\sim$ is closed under restriction, $(\nu a)P'' \sim (\nu a)([\![P']\!][y := z])$. Finally, a is sufficiently fresh to show that $(\nu a)([\![P']\!][y := z]) = ((\nu a)[\![P']\!])[y := z]$

(2) By induction on the derivation of P'', avoiding y.

Par:

Here $[\![P]\!] \xrightarrow{\langle\tilde{x}\rangle\, z} P''$, $y\#P, Q$, and by induction $P \xrightarrow{\tilde{x}(y)} P'$ where $[\![P'\{z/y\}]\!] = P''$. By Par using $y\#Q$, we derive $P \mid Q \xrightarrow{\tilde{x}(y)} P' \mid Q$. Finally, we note that since $y\#Q$, $[\![(P' \mid Q)\{z/y\}]\!] = P'' \mid [\![Q]\!]$.

Case:

Here $P_C \xrightarrow{\langle\tilde{x}\rangle\, z} P''$, where $P_C = \mathbf{case}\ \widetilde{\varphi} : \widetilde{Q}$ is in the range of $[\![\cdot]\!]$. Hence

P_C must be the encoding of some prefix-guarded sum, i.e., $P_C = [\![\Sigma_i \alpha_i.P_i]\!] =$ **case** $\top : [\![\alpha_1]\!].[\![P_1]\!] \; [\!] \; \ldots \; [\!] \; \top : [\![\alpha_i]\!].[\![P_i]\!]$. By transition inversion, we can deduce that for some j, $\alpha_j = \tilde{x}(y)$ and $[\![P_j]\!][y := z] = P''$. By the PREFIX rule, $\Sigma_i \alpha_i.P_i \xrightarrow{\tilde{x}(y)} P_j$.

Out:
A special case of CASE.

Rep:
Here $[\![P]\!] \mid ![\![P]\!] \xrightarrow{\langle\tilde{x}\rangle\, z} P''$. By induction $P \mid !P \xrightarrow{\tilde{x}(y)} P'$ where $[\![P'\{z/y\}]\!] = P''$. Using the BANG rule, we derive $!P \xrightarrow{\tilde{x}(y)} P'$.

Scope:
Here $[\![P]\!] \xrightarrow{\underline{x}\,\langle\tilde{z}\rangle} P''$, $y \# P, Q$ and $a \# \tilde{x}, y, z$. By induction $P \xrightarrow{\tilde{x}(y)} P'$ with $[\![P'\{z/y\}]\!] = P''$. Since $a \# \tilde{x}, y, z$, we obtain $(\nu a)P \xrightarrow{\tilde{x}(y)} (\nu a)P'$ by the RES rule. Finally, $[\![((\nu a)P')\{z/y\}]\!] = (\nu a)P''$. □

We give a proof for the strong operational correspondence.

Proof of Theorem 4.16.

(1) By induction on the derivation of P'. In case of input rule EIN, we apply Lemma 4.15 (1). The other interesting cases are:

Comm:
Here $P \xrightarrow{\overline{\tilde{x}}\langle y\rangle} P'$ and $Q \xrightarrow{\tilde{x}(z)} Q'$. By induction, $[\![P]\!] \xrightarrow{\overline{\langle\tilde{x}\rangle}\, y} P''$ where $P'' \sim [\![P']\!]$ and by Lemma 4.15 (1), $[\![Q]\!] \xrightarrow{\langle\tilde{x}\rangle\, y} Q''$ such that $[\![Q']\!][z := y] \sim Q''$. The COM rule lets us derive the transition

$$[\![P]\!] \mid [\![Q]\!] \xrightarrow{\tau} P'' \mid Q''$$

To complete the induction case, we note that $(\nu y)(P'' \mid Q'') \sim [\![(\nu y)(P' \mid Q'\{y/z\})]\!]$

Close:
Here $P \xrightarrow{\overline{\tilde{x}}\langle\nu y\rangle} P'$ and $Q \xrightarrow{\tilde{x}(y)} Q'$. We assume $y \# Q$; if not, y can be α-converted so that this holds. By induction, $[\![P]\!] \xrightarrow{\overline{\langle\tilde{x}\rangle}\,(\nu y)\, y} P''$ where $P'' \sim [\![P']\!]$ and by Lemma 4.15 (1), $[\![Q]\!] \xrightarrow{\langle\tilde{x}\rangle\, y} Q''$ such that $[\![Q']\!][y := y] = [\![Q']\!] \sim Q''$. The COM rule lets us derive the transition

$$[\![P]\!] \mid [\![Q]\!] \xrightarrow{\tau} (\nu y)(P'' \mid Q'')$$

To complete the induction case, we note that $(\nu y)(P'' \mid Q'') \sim [\![(\nu y)(P' \mid Q')]\!]$

Open:
Here $P \xrightarrow{\overline{\tilde{x}}\langle y\rangle} P'$ with $y \neq x$, and by induction, $[\![P]\!] \xrightarrow{\overline{\langle\tilde{x}\rangle}\, y} P''$ where $P'' \sim [\![P']\!]$. By OPEN, we derive $(\nu y)[\![P]\!] \xrightarrow{\overline{\langle\tilde{x}\rangle}\,(\nu y)\, y} P''$.

(2) By induction on the derivation of P''. The cases not shown are similar to Lemma 4.15 (2).

Com:
Here $[\![P]\!] \xrightarrow{\overline{\langle\tilde{x}\rangle}\,(\nu\tilde{y'})\, y} P''$, $[\![Q]\!] \xrightarrow{\langle\tilde{x}\rangle\, y} Q''$ and $y' \# Q$. Either $\tilde{y'} = \epsilon$ or $\tilde{y'} = y$; we proceed by case analysis.

(a) If $\tilde{y'} = \epsilon$, we have $P \xrightarrow{\overline{\tilde{x}}\langle y\rangle} P'$ where $[\![P']\!] = P''$ by induction and, by Lemma 4.15 (2), $Q \xrightarrow{\tilde{x}(z)} Q'$ where $[\![Q'\{y/z\}]\!] = Q''$. The COMM rule then lets us derive $P \mid Q \xrightarrow{\tau} P' \mid Q'\{y/z\}$.

(b) If $\tilde{y'} = y$, we have $P \xrightarrow{\overline{\tilde{x}}\langle\nu y\rangle} P'$ where $[\![P']\!] = P''$ by induction and, by Lemma 4.15 (2), $Q \xrightarrow{\tilde{x}(y)} Q'$ where $[\![Q'\{y/y\}]\!] = [\![Q']\!] = Q''$. The CLOSE rule then lets us derive $P \mid Q \xrightarrow{\tau} (\nu y)(P' \mid Q')$.

Open:
Here $[\![P]\!] \xrightarrow{\overline{\langle\tilde{x}\rangle}\ y} P''$ with $y \neq x$. By induction, $P \xrightarrow{\overline{\tilde{x}}\langle y\rangle} P'$ where $[\![P']\!] = P''$. By rule OPEN, $(\nu y)P \xrightarrow{\overline{\tilde{x}}\langle\nu y\rangle} P'$. □

We give the full abstraction result for this calculus. The definition of congruence for polyadic synchronisation pi-calculus can be found in [CM03] on page 6.

Theorem A.10. *For all* $^{e}\pi$ *processes* P *and* Q, $P \sim Q$ *iff* $[\![P]\!] \sim [\![Q]\!]$

Proof. $\mathcal{R} = \{(P, Q) : [\![P]\!] \sim [\![Q]\!]\}$ is an early congruence in the polyadic synchronisation pi-calculus; if $P \mathrel{\mathcal{R}} Q$ then

(1) If $P \xrightarrow{\tilde{x}(y)} P'$ and $[\![P]\!] \sim [\![Q]\!]$, since $\mathcal{R}$ is equivariant, we can assume that $y\#P, Q$ without loss of generality. Fix z. By Lemma 4.15 (1), $[\![P]\!] \xrightarrow{\langle\tilde{x}\rangle\ z} P''$ where $P'' \sim [\![P']\!][y := z] = [\![P'\{z/y\}]\!]$. Hence, since $[\![P]\!] \sim [\![Q]\!]$, $[\![Q]\!] \xrightarrow{\langle\tilde{x}\rangle\ z} Q''$ where $P'' \sim Q''$. Hence, by Lemma 4.15 (2) using $y\#Q$, $Q \xrightarrow{\tilde{x}(y)} Q'$ where $[\![Q'\{z/y\}]\!] = Q''$. By transitivity, $[\![P'\{z/y\}]\!] \sim [\![Q'\{z/y\}]\!]$.

(2) If $P \xrightarrow{\alpha} P'$ and $[\![P]\!] \sim [\![Q]\!]$, since $\mathcal{R}$ is equivariant, we can assume that $\mathrm{bn}(\alpha)\#P, Q$ without loss of generality. By Theorem 4.16 (1), we have that $[\![P]\!] \xrightarrow{[\![\alpha]\!]} P''$ with $P'' \sim [\![P']\!]$. Hence, since $[\![P]\!] \sim [\![Q]\!]$ and $\mathrm{bn}(\alpha)\#Q$, there is a Q'' such that $[\![Q]\!] \xrightarrow{[\![\alpha]\!]} Q''$ and $Q'' \sim P''$. By Theorem 4.16 (2), there is Q' such that $Q \xrightarrow{\alpha} Q'$ and $[\![Q']\!] = Q''$. By transitivity, $[\![P']\!] \sim [\![Q']\!]$.

Symmetrically, we show that $\mathcal{R} = \{(\mathbf{1}, [\![P]\!], [\![Q]\!]) : P \sim Q\}$ is a congruence in **PSPI**:

Static equivalence:
Trivial since there is only a unit assertion.

Symmetry:
By symmetry of $\sim$

Simulation:
Here $[\![P]\!] \xrightarrow{\alpha'} P''$ and $P \sim Q$. We proceed by case analysis on α':

(1) If $\alpha' = \underline{\langle\tilde{x}\rangle}\ z$, then by Lemma 4.15 (2) and a sufficiently fresh y, $P \xrightarrow{\tilde{x}(y)} P'$ where $[\![P'\{z/y\}]\!] = P''$. Since $P \sim Q$, there exists Q' such that $Q \xrightarrow{\tilde{x}(y)} Q'$ and $P'\{z/y\} \sim Q'\{z/y\}$. Hence, by Lemma 4.15 (1), $[\![Q]\!] \xrightarrow{\langle\tilde{x}\rangle\ z} Q''$ where $Q'' \sim [\![Q']\!][y := z] = [\![Q'\{z/y\}]\!]$. We have that $P'' = [\![P'\{z/y\}]\!] \mathrel{\mathcal{R}} [\![Q'\{z/y\}]\!] \sim Q''$, which suffices.

(2) If α' is not an input, since $\mathcal{R}$ is equivariant, we can assume that $\mathrm{bn}(\alpha')\#P, Q$ without loss of generality. Since $[\![P]\!] \xrightarrow{\alpha'} P''$, by Theorem 4.16 (2) we have that $P \xrightarrow{\alpha} P'$

where $[\![\alpha]\!] = \alpha'$ and $[\![P']\!] = P''$. Since $P \sim Q$, there is Q' such that $Q \xrightarrow{\alpha} Q'$ and $P' \sim Q'$. By Theorem 4.16 (1), $[\![Q]\!] \xrightarrow{[\![\alpha]\!]} Q''$, where $Q'' \sim [\![Q']\!]$. Hence $P'' = [\![P']\!] \mathcal{R} [\![Q']\!] \sim Q''$, which suffices.

Extension of arbitrary assertion:
Trivial since there is only a unit assertion. □

Lemma A.11. $[\![\cdot]\!]$ *is surjective up to* $\sim$ *on the set of case-guarded processes, that is, for every case-guarded* P *there is a* Q *such that* $[\![Q]\!] \sim P$.

Proof. By induction on the well-formed agent P.

Case $\underline{\langle\widetilde{x}\rangle}(\lambda y)y.P'$**:**
It is valid to consider only this form, since $\{y\} \in \textsc{vars}(y)$. The IH is for some Q', $[\![Q']\!] \sim P'$. Let $Q = \widetilde{x}(y).Q'$. Then $[\![Q]\!] = \underline{\langle\widetilde{x}\rangle}(\lambda y)y.[\![Q']\!] \sim \underline{\langle\widetilde{x}\rangle}(\lambda y)y.P'$.

Case $\overline{\langle\widetilde{x}\rangle}\ y.P'$**:**
From IH, we get for some Q', $[\![Q']\!] \sim P'$. Let $Q = \widetilde{x}\langle y\rangle.Q'$. Then $[\![Q]\!] = \overline{\langle\widetilde{x}\rangle}\ y.[\![Q']\!] \sim \overline{\langle\widetilde{x}\rangle}\ y.P'$.

Case $P' \mid P''$**:**
From IH, for some Q', Q'', we have $[\![Q']\!] \sim P'$ and $[\![Q'']\!] \sim P''$. Let $Q = Q' \mid Q''$. Then $[\![Q]\!] = [\![Q']\!] \mid [\![Q'']\!] \sim P' \mid P''$.

Case $(\nu x)P'$**:**
Let $Q = \nu x Q'$, then by the induction hypothesis $[\![Q]\!] = (\nu x)[\![Q']\!] \sim (\nu x)P'$.

Case $!P'$**:**
Let $Q = !Q'$ (Q' from IH). $[\![Q]\!] = ![\![Q']\!] \sim !P'$.

Case $\mathbf{0}$**:**
Then $[\![\mathbf{0}]\!] = \mathbf{0} \sim \mathbf{0}$.

Case $(\!|\mathbf{1}|\!)$**:**
Then $[\![\mathbf{0}]\!] = \mathbf{0} \sim (\!|\mathbf{1}|\!)$.

Case case $\widetilde{\varphi} : \widetilde{P'}$**:**
For induction hypothesis $\mathrm{IH}_{\mathbf{case}}$, we have for every i there is Q'_i such that $[\![Q'_i]\!] \sim P'_i$. The proof proceeds by induction on the length of $\widetilde{\varphi}$.

Base case:
Let $Q = \mathbf{0}$, then $[\![Q]\!] = \mathbf{0} \sim \mathbf{case}$.

Induction step:
In this case, we get the following IH

$$[\![Q'']\!] \sim \mathbf{case}\ \varphi_1 : P_1\ [\!]\ \dots\ [\!]\ \varphi_n : P_n$$

We need to show that there is some $[\![Q]\!]$ such that

$$[\![Q]\!] \sim \mathbf{case}\ \varphi_1 : P_1\ [\!]\ \dots\ [\!]\ \varphi_n : P_n\ [\!]\ \varphi_{n+1} : P_{n+1} = P$$

First, we note that $\mathrm{IH}_{\mathbf{case}}$ holds for every i and in particular $i = n + 1$, thus we get $[\![Q'_{n+1}]\!] \sim P_{n+1}$. Second, we note that φ_{n+1} has two forms, thus we proceed by case analysis on φ_{n+1}.

Case $\varphi_{n+1} = \bot$:
Let $Q = Q''$. Then

$$\begin{array}{rcl} [\![Q]\!] & = & [\![Q'']\!] \\ & \sim & \mathbf{case}\ \varphi_1 : P_1 \ [\!] \ldots [\!]\ \varphi_n : P_n \\ & \sim & \mathbf{case}\ \varphi_1 : P_1 \ [\!] \ldots [\!]\ \varphi_n : P_n \ [\!]\ \bot : P_{n+1} \end{array}$$

We conclude the case.

Case $\varphi_{n+1} = \top$:
From the assumption, we know that P_{n+1} is of form $\alpha.P'_{n+1}$ and that $[\![Q'_{n+1}]\!] \sim \alpha.P'_{n+1}$. By investigating the construction of Q'_{n+1} we can conclude that $Q'_{n+1} = \alpha.Q''_{n+1}$ where $[\![Q''_{n+1}]\!] \sim P'_{n+1}$. The agent from IH Q'' is either $\mathbf{0}$, or prefixed agent, or a mixed sum.
In case $Q'' = \mathbf{0}$, let $Q = Q'_{n+1}$, then $[\![Q]\!] = [\![Q'_{n+1}]\!] \sim P$.
In case Q'' is prefixed agent, let $Q = Q'' + Q'_{n+1}$. Since Q'' and Q'_{n+1} are prefixed, Q is well formed. Then $[\![Q]\!] = \mathbf{case}\ \top : [\![Q'']\!]\ [\!]\ \top : [\![Q'_{n+1}]\!] \sim \mathbf{case}\ \varphi_1 : P_1\ [\!] \ldots [\!]\ \varphi_n : P_n\ [\!]\ \top : P_{n+1}$.
In case Q'' is a sum, let $Q = Q'' + Q'_{n+1}$. Since Q'_{n+1} is guarded, Q is well formed. Then

$$\begin{array}{rcl} [\![Q]\!] & = & \mathbf{case}\ \top : [\![Q'']\!]\ [\!]\ \top : [\![Q'_{n+1}]\!] \\ & \sim & \mathbf{case}\ \top : (\mathbf{case}\ \varphi_1 : P_1\ [\!] \ldots [\!]\ \varphi_n : P_n) \\ & & [\!]\ \top : [\![Q'_{n+1}]\!] \\ & \sim & \text{(by Lemma 3.3)} \\ & & \mathbf{case}\ \varphi_1 : P_1\ [\!] \ldots [\!]\ \varphi_n : P_n \\ & & [\!]\ \top : [\![Q'_{n+1}]\!] \\ & \sim & \mathbf{case}\ \varphi_1 : P_1\ [\!] \ldots [\!]\ \varphi_n : P_n \\ & & [\!]\ \top : P'_{n+1} \end{array}$$

This concludes the proof. □

Lemma A.12. $[\![\cdot]\!]$ *is injective, that is, for all* P, Q, *if* $[\![P]\!] = [\![Q]\!]$ *then* $P = Q$.

Proof. By induction on P and Q while inspecting all the possible cases. □

A.3. **Value-passing CCS.** This section contains the full proofs of the results found in Section 4.5 for the value-passing CCS.

Lemma A.13. *If P is a* **VPCCS** *process such that* $P \xrightarrow{\overline{M}\,(\nu\widetilde{x})\,N} P''$ *then* $\widetilde{x} = \epsilon$

Proof. By induction on the derivation of P'. Obvious in all cases except Open, where we derive a contradiction since only values can be transmitted and yet only channels can be restricted - hence the name a is both a name and a value. □

We prove strong operational correspondence using the implicit translation from value-passing CCS to CCS of Milner [Mil89, Section 2.6, p. 56]. If L is a set of labels, we write $L\#\alpha$ to mean that for every $\ell \in L$ there is no v such that $\alpha = \ell_v$ or $\alpha = \overline{\ell}_v$.

Proof of Theorem 4.23.

(1) By induction on the derivation of P'.

Act:
We have that $\alpha.P \xrightarrow{\alpha} P$. Since $\alpha.P$ is a closed value-passing CCS agent, α cannot be a free input. Thus, α is an output action $\alpha = \overline{x}(v)$ for some x and v. The Out rule then admits the derivation $[\![\overline{x}(v).P]\!] = \overline{x}\ v.[\![P]\!] \xrightarrow{\overline{x}\ v} [\![P]\!]$.

Sum:
There are two cases to consider: either $\Sigma_i P_i$ is the encoding of an input, or a summation.

(a) If it is an encoding of an input $\Sigma_i P_i = x(y).P = \Sigma_v x(v).P\{v/y\}$, then the action α must be the free input action $x(v)$ for some value v. Thus, for each v, we can derive $[\![x(y).P]\!] = \underline{x}(\lambda y)y.[\![P]\!] \xrightarrow{\underline{x}\ v} [\![P\{v/y\}]\!]$ using the In rule.

(b) Otherwise it is a summation. We assume $\Sigma_i P_i \xrightarrow{\alpha} P'$. From induction hypothesis, we have $P_i \xrightarrow{\alpha} P'$, and

$$[\![P_i]\!] \xrightarrow{[\![\alpha]\!]} [\![P']\!]$$

for any i. By using this and the Case rule, we derive

$$[\![\Sigma_i P_i]\!] = \mathbf{case}\ \top : [\![P_1]\!]\ [\!]\ \cdots\ [\!]\ \top : [\![P_i]\!] \xrightarrow{\alpha} [\![P']\!]$$

as required.

Com1:
Here $P \xrightarrow{\alpha} P'$, and by induction $[\![P]\!] \xrightarrow{[\![\alpha]\!]} [\![P']\!]$. The Par rule admits derivation of the transition $[\![P]\!] \mid [\![Q]\!] \xrightarrow{[\![\alpha]\!]} [\![P']\!] \mid [\![Q]\!]$, as, by using Lemma A.13, freshness side condition is vacuous.

Com2:
Symmetric to Com1.

Com3:
Here $P \xrightarrow{\alpha} P'$ and $Q \xrightarrow{\overline{\alpha}} Q'$. Since α is in the range of $\widehat{\cdot}$, there are x and v such that $\alpha = x(v)$ and $\overline{\alpha} = \overline{x}(v)$ (or vice versa, in which case read the next sentence symmetrically). By the induction hypotheses, $[\![P]\!] \xrightarrow{\underline{x}\ v} [\![P']\!]$ and $[\![Q]\!] \xrightarrow{\overline{x}\ v} [\![Q']\!]$. Then $[\![P]\!] \mid [\![Q]\!] \xrightarrow{\tau} [\![P']\!] \mid [\![Q']\!]$ by the Com rule.

Res:
Here $P \setminus L \xrightarrow{\alpha} P' \setminus L$ with $L\#\alpha$. Hence $\overrightarrow{L}\#[\![\alpha]\!]$. By induction $[\![P]\!] \xrightarrow{[\![\alpha]\!]} [\![P']\!]$. We use the Res rule $|L|$ times to derive $(\nu\overrightarrow{L})[\![P]\!] \xrightarrow{[\![\alpha]\!]} (\nu\overrightarrow{L})[\![P']\!]$.

Rep:
Here $P \mid !P \xrightarrow{\alpha} P'$. By induction $[\![P]\!] \mid ![\![P]\!] \xrightarrow{[\![\alpha]\!]} [\![P']\!]$. By the Rep rule $![\![P]\!] \xrightarrow{[\![\alpha]\!]} [\![P']\!]$

(2) By induction on the derivation of P'.

In:
Here $\underline{x}(\lambda y)y.[\![P]\!] \xrightarrow{\underline{x}\ v} [\![P\{v/y\}]\!]$. We match this by deriving $x(y).P \xrightarrow{x(v)} P\{v/y\}$ using the Act and Sum rules, where $[\![x(y).P]\!] = \underline{x}(\lambda y)y.[\![P]\!]$.

Out:
Here $\overline{x}\, v.[\![P]\!] \xrightarrow{\overline{x}\, v} [\![P]\!]$. We match this by deriving $\overline{x}(v).P \xrightarrow{\overline{x}(v)} P$ using the Act rule.

Com:
Here $[\![P]\!] \xrightarrow{\overline{x}\,(\nu\tilde{y})\, v} P''$, $[\![Q]\!] \xrightarrow{\underline{x}\, v} Q''$. By Lemma A.13, $\tilde{y} = \epsilon$, and by induction, $P \xrightarrow{\overline{x}(v)} P'$ and $Q \xrightarrow{x(v)} Q'$, where $[\![P']\!] = P''$ and $[\![Q']\!] = Q''$. Using the Com3 rule we derive $P \mid Q \xrightarrow{\tau} P' \mid Q'$

Par:
Straightforward.

Case:
Our case statement can either be the encoding of either a summation or an **if** statement. We proceed by case analysis:

(a) Here $[\![P_j]\!] \xrightarrow{\alpha'} P''$. By induction, $P_j \xrightarrow{\alpha} P'$ where $[\![\alpha]\!] = \alpha'$ and $P'' = [\![P']\!]$. By Sum, $\Sigma_i P_i \xrightarrow{\alpha} P'$.

(b) Here $[\![P]\!] \xrightarrow{\alpha'} P''$ and $\mathbf{1} \vdash b$. By induction, $P \xrightarrow{\alpha} P'$ where $[\![\alpha]\!] = \alpha'$ and $[\![P']\!] = P''$. Since b evaluates to true, **if** b **then** $P \xrightarrow{\alpha} P'$.

Rep:
Straightforward.

Scope:
Here $[\![P]\!] \xrightarrow{\alpha'} P''$ with $x\sharp\alpha'$ and by induction, $P \xrightarrow{\alpha} P'$ where $\alpha' = [\![\alpha]\!]$ and $P'' = [\![P']\!]$. Hence we can derive $P \setminus \{x\} \xrightarrow{\alpha} P' \setminus \{x\}$ by the Res rule.

Open:
Impossible, by Lemma A.13. □

References

[AF01] Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. In *Proceedings of POPL '01*, pages 104–115. ACM, January 2001.

[ÅP10] Johannes Åman Pohjola. Verifying psi-calculi. M. Sc. thesis IT ; 10 052, Uppsala University, Department of Information Technology, 2010.

[ÅP15] Johannes Åman Pohjola. Isabelle proof scripts for sorted psi-calculi. Available at `http://www.it.uu.se/research/group/mobility/theorem/sortedPsi.tar.gz`, 2015.

[ÅPBP+13] Johannes Åman Pohjola, Johannes Borgström, Joachim Parrow, Palle Raabjerg, and Ioana Rodhe. Negative premises in applied process calculi. Technical Report 2013-014, Department of Information Tecnology, Uppsala University, 2013.

[Ben10] Jesper Bengtson. *Formalising process calculi*. PhD thesis, Uppsala University, 2010.

[BGP+14] Johannes Borgström, Ramūnas Gutkovas, Joachim Parrow, Björn Victor, and Johannes Åman Pohjola. A sorted semantic framework for applied process calculi (extended abstract). In Martín Abadi and Alberto Lluch Lafuente, editors, *Trustworthy Global Computing*, number 8358 in Lecture Notes in Computer Science, pages 103–118. Springer, 2014.

[BGRV15] Johannes Borgström, Ramūnas Gutkovas, Ioana Rodhe, and Björn Victor. A parametric tool for applied process calculi. *ACM Transactions on Embedded Computing Systems*, 14(1), 2015.

[BJPV11] Jesper Bengtson, Magnus Johansson, Joachim Parrow, and Björn Victor. Psi-calculi: a framework for mobile processes with nominal data and logic. *LMCS*, 7(1:11), 2011.

[Bla11] Bruno Blanchet. Using Horn clauses for analyzing security protocols. In Véronique Cortier and Steve Kremer, editors, *Formal Models and Techniques for Analyzing Security Protocols*, volume 5 of *Cryptology and Information Security Series*, pages 86–111. IOS Press, March 2011.

[CGK+13] Sjoerd Cranen, Jan Friso Groote, Jeroen J. A. Keiren, Frank P. M. Stappers, Erik P. de Vink, Wieger Wesselink, and Tim A. C. Willemse. An overview of the mCRL2 toolset and its recent advances. In Nir Piterman and Scott A. Smolka, editors, *TACAS*, volume 7795 of *Lecture Notes in Computer Science*, pages 199–213. Springer, 2013.

[CM03] Marco Carbone and Sergio Maffeis. On the expressive power of polyadic synchronisation in π-calculus. *Nordic Journal of Computing*, 10(2):70–98, 2003.

[DY83] Danny Dolev and Andrew C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

[EOW07] Burak Emir, Martin Odersky, and John Williams. Matching objects with patterns. In *Proceedings of the 21st European Conference on Object-Oriented Programming*, ECOOP'07, pages 273–298, Berlin, Heidelberg, 2007. Springer-Verlag.

[FG96] Cédric Fournet and Georges Gonthier. The reflexive CHAM and the join-calculus. In *Proc. POPL*, pages 372–385, 1996.

[FGM05] Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. A type discipline for authorization policies. In Mooly Sagiv, editor, *Proc. of ESOP 2005*, volume 3444 of *LNCS*, pages 141–156. Springer, 2005.

[Gel85] David Gelernter. Generative communication in Linda. *ACM TOPLAS*, 7(1):80–112, January 1985.

[Giv14] Thomas Given-Wilson. On the expressiveness of intensional communication. In Johannes Borgström and Silvia Crafa, editors, *Proceedings of EXPRESS/SOS 2014*, volume 160 of *EPTCS*, pages 30–46, 2014.

[Gor10] Daniele Gorla. Towards a unified approach to encodability and separation results for process calculi. *Information and Computation*, 208(9):1031–1053, 2010.

[GP01] Murdoch J. Gabbay and Andrew M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13:341–363, 2001.

[GSV04] Pablo Giambiagi, Gerardo Schneider, and Frank D. Valencia. On the expressiveness of infinite behavior and name scoping in process calculi. In Igor Walukiewicz, editor, *Proceedings of FOSSACS 2004*, volume 2987 of *LNCS*, pages 226–240. Springer, 2004.

[GWGJ10] Thomas Given-Wilson, Daniele Gorla, and Barry Jay. Concurrent pattern calculus. In Cristian Calude and Vladimiro Sassone, editors, *Theoretical Computer Science*, volume 323 of *IFIP Advances in Information and Communication Technology*, pages 244–258. Springer, 2010.

[HJ06] Christian Haack and Alan Jeffrey. Pattern-matching spi-calculus. *Information and Computation*, 204(8):1195–1263, 2006.

[Hon93] Kohei Honda. Types for dyadic interaction. In Eike Best, editor, *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer, 1993.

[HU10] Brian Huffman and Christian Urban. A new foundation for Nominal Isabelle. In *Proceedings of the First international conference on Interactive Theorem Proving*, ITP'10, pages 35–50. Springer, 2010.

[Hüt11] Hans Hüttel. Typed psi-calculi. In Joost-Pieter Katoen and Barbara König, editors, *CONCUR 2011 – Concurrency Theory*, volume 6901 of *LNCS*, pages 265–279. Springer, 2011.

[Hüt14] Hans Hüttel. Types for resources in ψ -calculi. In Martín Abadi and Alberto Lluch Lafuente, editors, *Trustworthy Global Computing*, LNCS, pages 83–102. Springer International Publishing, 2014.

[HV13] Hans Hüttel and Vasco T Vasconcelos. The foundations of behavioural types. State-of-the art report of WG1 of the BETTY project (EU COST Action IC1201). To appear, 2013.

[JBPV10] Magnus Johansson, Jesper Bengtson, Joachim Parrow, and Björn Victor. Weak equivalences in psi-calculi. In *Proc. of LICS 2010*, pages 322–331. IEEE, 2010.

[JVP12] Magnus Johansson, Björn Victor, and Joachim Parrow. Computing strong and weak bisimulations for psi-calculi. *Journal of Logic and Algebraic Programming*, 81(3):162–180, 2012.

[Kri09] Neelakantan R. Krishnaswami. Focusing on pattern matching. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 366–378, New York, NY, USA, 2009. ACM.

[LSD11] Yang Liu, Jun Sun, and Jin Song Dong. PAT 3: An extensible architecture for building multi-domain model checkers. In Tadashi Dohi and Bojan Cukic, editors, *ISSRE '11*, pages 190–199. IEEE, 2011.

[Mil89] Robin Milner. *Communication and Concurrency.* Prentice-Hall, Inc., 1989.

[Mil93] Robin Milner. The polyadic π-calculus: A tutorial. In Friedrich L. Bauer, Wilfried Brauer, and Helmut Schwichtenberg, editors, *Logic and Algebra of Specification*, volume 94 of *Series F*. NATO ASI, Springer, 1993.

[PBRÅP13] Joachim Parrow, Johannes Borgström, Palle Raabjerg, and Johannes Åman Pohjola. Higher-order psi-calculi. *Mathematical Structures in Computer Science*, FirstView, June 2013.

[Pit03] Andrew M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 186:165–193, 2003.

[San93] Davide Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms.* PhD thesis, University of Edinburgh, 1993. CST-99-93 (also published as ECS-LFCS-93-266).

[SLDC09] Jun Sun, Yang Liu, Jin Song Dong, and Chunqing Chen. Integrating specification and programs for system modeling and verification. In *TASE '09*, pages 127–135. IEEE Computer Society, 2009.

[SNM07] Don Syme, Gregory Neverov, and James Margetson. Extensible pattern matching via a lightweight language extension. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, pages 29–40, New York, NY, USA, 2007. ACM.

[SS05] Alan Schmitt and Jean-Bernard Stefani. The Kell calculus: A family of higher-order distributed process calculi. In Corrado Priami and Paola Quaglia, editors, *Global Computing*, volume 3267 of *LNCS*, pages 146–178. Springer Berlin Heidelberg, 2005.

[SW01] Davide Sangiorgi and David Walker. *The π-calculus: a Theory of Mobile Processes.* Cambridge University Press, 2001.

[Urb08] Christian Urban. Nominal techniques in Isabelle/HOL. *Journal of Automated Reasoning*, 40(4):327–356, May 2008.

This work is licensed under the Creative Commons Attribution-NoDerivs License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nd/2.0/ or send a letter to Creative Commons, 171 Second St, Suite 300, San Francisco, CA 94105, USA, or Eisenacher Strasse 2, 10777 Berlin, Germany

Paper III

A Session Type System for Unreliable Broadcast Communication

Ramūnas Gutkovas[1], Dimitrios Kouzapas[2], and Simon J. Gay[2]

[1] Department of Information Technology, Uppsala University, Sweden
ramunas.gutkovas@it.uu.se
[2] School of Computing Science, University of Glasgow, UK
{dimitrios.kouzapas, simon.gay}@glasgow.ac.uk

Abstract. Session types are formal specifications of communication protocols, allowing protocol implementations to be verified by typechecking. Up to now, session type disciplines have assumed that the communication medium is reliable, with no loss of messages. However, unreliable broadcast communication is common in a wide class of distributed systems such as ad-hoc and wireless sensor networks. Often such systems have structured communication patterns that should be amenable to analysis by means of session types. We introduce a process calculus with unreliable broadcast communication, and equip it with a session type system that we show is sound. We capture two common operations, scatter and gather, inhabiting dual session types. To cope with unreliability in a session we introduce an autonomous recovery mechanism that does not require acknowledgements from session participants. Our session type formalisation is the first to consider unreliable communication.

1 Introduction

Session types [10] are formal specifications of communication protocols, allowing protocol implementations to be verified by typechecking. They provide a discipline for ensuring good communication properties in concurrent systems. Session types have been applied to a range of programming language frameworks and paradigms [2] and several session type technologies have been developed [1].

One of the basic assumptions underlying session types up to now, is the reliability of communication. It is always assumed that messages are never lost and are delivered to the receiver. This is not a realistic assumption to make when it comes to the communication that takes place in ad-hoc and wireless sensor networks. Often networks that use a shared and stateless communication medium use broadcast to deliver messages. Furthermore, messages are lost due to broadcast collisions in the shared medium, and the stateless nature of the communication medium. Nevertheless, in the presence of unreliability, these networks feature structured communication, and we would like to be able to describe it using session types.

In this paper we are the first to introduce a safe session type system for unreliable broadcast communication. The communication semantics that we propose

can capture the broadcast and gather operations in the presence of unreliability. To cope with message loss we propose a recovery mechanism that is enabled only when conditions that imply a recovery situation are met. More, specifically in our setting we assume:

Locality: Processes exist as network components, because we want to model real conditions in wireless and ad-hoc networks. Although we do not consider mobility in the current setting, process locality enables the possibility of adding network component mobility in the future. Without mobility we can still capture many classes of networks.

Channel connectivity: Communication takes place under conditions of local connectivity. An interaction between the endpoints of a channel may only take place if the two endpoints exist in connected locations. The channel connectivity assumption enables the specification of more realistic wireless network scenarios such as spatial distribution of processes, and processes with different communication ranges.

Synchronous unreliable broadcast semantics: In unreliable broadcast semantics, when a process sends a message, an arbitrary (possibly empty) set of receiver processes can receive the message. We assume synchronous broadcasting communication, since we require a stateless communication medium. A stateless communication medium allows us to model more realistic scenarios such as wireless sensor networks.

Unreliable gather semantics: Another typical communication pattern in wireless networks is the gather operation, where a process expects to receive values from an arbitrary (possibly empty) set of senders.

Message loss: An additional realistic assumption is that sent messages that are supposed to be involved in a gather operation might be lost. Nevertheless, it is also assumed that even if a send message is never received the sender process continues its computation, thus enabling a form of a recovery mechanism.

Recovery: When messages are lost it is unavoidable to observe processes that are not synchronised with the overall protocol. To overcome this problem we propose a recovery semantics that is enabled in a non-deterministic fashion only when a process is waiting to receive a message. A non-deterministic hard recovery allows us to model general recovery situations such as message receive time-out, or message collision detection.

1.1 Motivation

To further motivate our contribution consider the π-calculus process:

$$P = s_!\langle v\rangle.P_0 \mid \prod_{i\in I} s_?(x).P_i$$

where process $\prod_{i\in I} s_?(x).P_i$ is a parallel composition of input prefixed processes that receive a message on channel s. Consider also an unreliable broadcasting semantics for the above process:

$$P \to P_0 \mid \prod_{i\in J} P_j\{v/x\} \mid \prod_{k\in K} s_?(x).P_k$$

where $I = J \cup K$ for disjoint J and K; and broadcast value v is only received by a subset of the receiving processes.

We claim that session types cannot model the above interaction without introducing a complicated and unrealistic level of abstraction. Binary session types are inadequate for the above scenario, since they only describe two-endpoint communication.

Modelling unreliable broadcasting communication in multiparty session types needs to account for message loss in the communication medium; a receiver may or may not receive depending on whether the message was lost in the communication medium or not. The only operation that can model a choice in multiparty session types is the choice operator. For the purposes of demonstration consider the multiparty syntax presented in [7]; type $\mathtt{p} \to \mathtt{q} : T; G$ denotes a type where role $\mathtt{p}$ sends a message with type T to role $\mathtt{q}$ and then continues as G, and type $\mathtt{p} \to \mathtt{q} : \{l_i : G_i\}_{i \in I}$ denotes the type where role $\mathtt{p}$ makes a choice labelled l_i on role $\mathtt{q}$ and continues as G_i for $i \in I$. A multiparty global protocol that attempts to describe an unreliable communication medium using the multiparty choice operation follows:

$$\mathtt{p} \to \mathtt{m} : T;\ \mathtt{m} \to \mathtt{q} : \{\mathit{send} : \mathtt{m} \to \mathtt{q} : T; \mathtt{end},\ \ \mathit{lose} : \mathtt{end}\}.$$

Consider that role $\mathtt{p}$ wants to broadcast a message of type T to role $\mathtt{q}$ with role $\mathtt{m}$ representing the communication medium. Role $\mathtt{p}$ first sends the message to the communication medium $\mathtt{m}$ and then role $\mathtt{m}$ informs role $\mathtt{q}$ of its choice whether to lose or send the message. The above specification has three deficiencies:

- (i) role $\mathtt{q}$ is informed that a message was broadcast and never received, which is unrealistic since in real applications a role does not know whether a message supposed to be received was sent or not;
- (ii) there is an unnecessary loss of abstraction by modeling the communication medium in terms of a role. Often, someone who implements broadcasting applications does not need to know lower level communication details; and
- (iii) extending the above scenario for more than one receiver would lead to a complicated communication protocol.

Our solution generalises the theory of binary session types to cope with the semantics of unreliable broadcast and gather. Concretely, a typed session endpoint is dedicated to the process that initiates a session, whereas multiple dually typed endpoints are used by the set of processes accepting the session. For example consider the network:

$$[a_!(s).\breve{s}_!\langle 1\rangle.\breve{s}_?(y).\mathbf{0}]_l \mid \prod_{i \in I}[a_?(s).s_?(x).s_!\langle x+i\rangle.\mathbf{0}]_{l_i}$$

where $[P]_l$ denotes process P in location l. A broadcast interaction on name a will create a new session with endpoints s and $\breve{s}$.

$$\begin{aligned} & [a_!(s).\breve{s}_!\langle 1\rangle.\breve{s}_?(y).\mathbf{0}]_l \mid \prod_{i \in I}[a_?(s).s_?(x).s_!\langle x+i\rangle.\mathbf{0}]_i \\ \to\ & (\nu\, s)([\breve{s}_!\langle 1\rangle.\breve{s}_?(y).\mathbf{0}]_l \mid \prod_{j \in J}[s_?(x).s_!\langle x+j\rangle.\mathbf{0}]_j) \mid \prod_{k \in K}[a_?(s).s_?(x).s_!\langle x+k\rangle.\mathbf{0}]_k \end{aligned}$$

Session endpoint $\breve{s}$ is expected to be unique used only by the session initiation process, whereas the session broadcast interaction may create more than one instances instances of endpoint s. A broadcast action happens from endpoint $\breve{s}$ towards endpoints s, where in the above example corresponds to the broadcasting of value 1 from endpoint $\breve{s}$ to endpoints s. Dually, a gather operation synchronises on the sending prefixes of endpoints s and the receive prefix of endpoint $\breve{s}$, which simultaneously gathers all sent information. In the above example a gather operation corresponds to the sending of values $x + i$ from endpoints s to endpoint $\breve{s}$. Endpoint $\breve{s}$ will gather all sent values modulo a pre-defined aggregation operator $\odot$ and substitute the result for the variable y.

The one to many correspondence between endpoints $\breve{s}$ and s enables the use of standard session type duality. In the above example the type of $\breve{s}$ is $!\texttt{int}.?\texttt{int}.\texttt{end}$ with $\texttt{int}$ being a base type and dually, the type of s is $?\texttt{int}.!\texttt{int}.\texttt{end}$.

Outline. In Section 2 we introduce our unreliable broadcast calculus. In Section 3, we define a session type system and prove the main results of this paper. In Section 4, we use our framework to correctly implement a data aggregation protocol from wireless sensor networks. Lastly, we conclude in Section 5.

2 A Broadcast Calculus

In this section, we introduce a broadcast process calculus, its syntax and semantics. Its main defining features are the unreliability of the communication medium and the aggregate broadcast communication primitives. The syntax of the calculus is defined in two levels: processes and networks with locations.

2.1 Syntax

Definition 1 (Process). *We need the following set of data. Let $\mathcal{C}$ be a countable set of (shared)* channels *ranged over by $a, b, c, \ldots$; $\mathcal{S}$ be a countable set of* session channels *ranged over by $s, s', \ldots$; $\mathcal{X}$ be a countable set of* variables *ranged over by $x, y, z, \ldots$; and $\mathcal{V}$ be a countable set of* process variables *ranged over by $X, Y, \ldots$ We assume that these sets are disjoint.*

We parameterise syntax with expressions and conditions. Let $\mathcal{E}$ be a non-empty set of expressions ranged over by $e, e', \ldots$. We require parameters: a binary operation $\odot$ on $\mathcal{E}$ that we call aggregation operation, and a distinguished element $\mathbf{1}$ *of $\mathcal{E}$ called unit. Let $\mathcal{F}$ be a non-empty set of conditions ranged over by φ, and a truth predicate $\Vdash \subseteq \mathcal{F}$, where we write $\Vdash \varphi$ for $\varphi \in \Vdash$. Both $\mathcal{E}$ and $\mathcal{F}$ may contain variables and the set of free variables is denoted with* $\mathrm{fn}(e)$ *and* $\mathrm{fn}(\varphi)$*. There is also a substitution function $e\{e'/x\}$ defined on $\mathcal{E}$ and $\mathcal{F}$ such that* $\mathrm{fn}(e\{e'/x\}) \subseteq \mathrm{fn}(e) \cup \mathrm{fn}(e')$ *and similarly for conditions. Then, the syntax*

of processes is defined as follows:

$$
\begin{array}{rcll}
\kappa & ::= & s \mid \breve{s} & \\
P, Q, R & ::= & a_!(s).P & \textit{(Request)} \\
& \mid & a_?(s).P & \textit{(Accept)} \\
& \mid & \kappa_!\langle e \rangle.P & \textit{(Send/Scatter)} \\
& \mid & \kappa_?(x).P & \textit{(Receive/Gather)} \\
& \mid & \breve{s} \oplus \ell.P & \textit{(Selection)} \\
& \mid & s \,\&\, \{\ell_1 : P_1, \ldots, \ell_n : P_n\} & \textit{(Branching)} \\
& \mid & \mathsf{if}\ \varphi\ \mathsf{then}\ P\ \mathsf{else}\ Q & \textit{(Conditional)} \\
& \mid & \mu X.P & \textit{(Recursion)} \\
& \mid & X & \textit{(Process Variable)} \\
& \mid & \mathbf{0} & \textit{(Inaction)}
\end{array}
$$

(Request) and (Accept) bind s in P; and (Receive) binds x in P. Also, (Recursion) is a binder and binds X in P. We define $\mathrm{fn}(P)$ *to be a set of free channels, session channels, variables, and process variables of P in the standard way. We identify processes up to α-equivalence.*

In (Request), (Accept), (Send), (Receive), (Branching) and (Selection) forms, a, κ, and s are called subjects. The forms themselves are called prefixes.

We impose a well-formedness condition on the processes above: all the free occurrences of s in P of (Request) are under $\breve{s}$, and free occurrences of s in P of (Accept) are not *under $\breve{s}$.*

The (Request) and (Accept) forms are used to initiate a session s on a shared channel a. They are not symmetric operations in the sense that (Request) is a broadcast operation that may initiate a session with multiple partners, while (Accept) accepts a session from a unique process. We distinguish these endpoints by annotating the process requesting a session with $\breve{s}$ (aggregate s), and in the accepting process we make no annotations for s.

There are two flavours of sending and receiving operations determined by the session channel endpoint annotation. The operation $\breve{s}_!\langle e \rangle.P$ is a broadcast (Scatter) to all session partners, while $s_!\langle e \rangle.P$ is a send operation to one partner. Analogously, $\breve{s}_?(x).P$ is an aggregation of messages received (Gather) from the partners, binding it to x, while $s_?(x).P$ receives a message from the unique partner. We restrict selection only on $\breve{s}$ in (Selection) and a choice of branch can be only made on s (Branching).

It is worth pointing out that processes are not concurrent; we introduce concurrency in the network level. There is no loss of generality, however, since we may have several processes running in a network in the same location. We use the notion of a location of distributed π-calculus [9], although in our formalisation the locations are fixed and the processes are not mobile, that is, they cannot migrate from one location to another.

Definition 2 (Network). *Let $\mathcal{N} = \mathcal{C} \cup \mathcal{S}$ ranged over by n, and let $\mathcal{L}$ be a countable set of* locations *ranged over by l. Then, the network is defined by the*

following grammar:

$$\begin{array}{rcll} \psi & ::= & \varepsilon \ \mid \ \psi \mid s & \textit{(Session State)} \\ N, M & ::= & [P \triangleright R \mid \psi]_l & \textit{(Node)} \\ & \mid & N \mid M & \textit{(Parallel)} \\ & \mid & (\nu\, n)N & \textit{(Restriction)} \end{array}$$

We extend the fn *function to networks. We may write* $[P \triangleright Q]_l$ *for* $[P \triangleright Q \mid \varepsilon]_l$. *We define* $\mathsf{cnt}(s, \psi) = c$ *to denote the number of occurrences* c *of* s *in* ψ *by structural recursion as* $\mathsf{cnt}(s, \varepsilon) = 0$, *and* $\mathsf{cnt}(s, \psi \mid s') = \mathsf{cnt}(s, \psi)$ *if* $s \neq s'$, *otherwise* $\mathsf{cnt}(s, \psi \mid s) = 1 + \mathsf{cnt}(s, \psi)$.

The form (Node) consists of a process P that is executing in a location l, a recovery process R that may take over if P cannot proceed in a session, and a session state store ψ that tracks the number of session interactions the process performed thus far. A process may participate in several sessions and therefore ψ is used to track more than one session. Intuitively, a network is a parallel composition of nodes. (Restriction) binds both session and shared channels.

2.2 Operational Semantics

We define the operational semantics as a reduction relation on networks. In the standard way, we make use of structural congruence. We also make use of notation $k \# P$, pronounced as k is fresh for P, to mean that $k \notin \mathrm{fn}(P)$ where $k \in \mathcal{S} \cup \mathcal{C} \cup \mathcal{X} \cup \mathcal{V}$, and similarly for networks $k \# N$. We write $(\nu\, \widetilde{n})N$ for the network $(\nu\, n_1) \dots (\nu\, n_m)N$ where the sequence $\widetilde{n} = (n_1, \dots, n_m)$ may be empty.

Definition 3 (Structural Congruence). *Structural congruence on processes (resp., session state and networks) is defined to be the smallest congruence relation satisfying the following rules:*

$$\begin{array}{rll} \mu X.P \equiv P\{\mu X.P/X\} & & \\ \psi \equiv \psi' & & \textit{if } \psi \textit{ is a permutation of } \psi' \\ N \equiv [\mathbf{0} \triangleright \mathbf{0} \mid \varepsilon]_l \mid N & & \\ N \mid M \equiv M \mid N & & \\ (M \mid N) \mid S \equiv M \mid (N \mid S) & & \\ (\nu\, n)N \mid M \equiv (\nu\, n)(N \mid M) & \textit{if } n \# M & \\ [P \triangleright R \mid \psi]_l \equiv [P' \triangleright R' \mid \psi']_l & \textit{if } P \equiv P' \textit{ and } R \equiv R' \textit{ and } \psi \equiv \psi' & \end{array}$$

Structural congruence on processes includes unrolling of recursion. We identify session states up to permutation (reordering). The structural congruence on network is standard: parallel composition is commutative and associative, with $[\mathbf{0} \triangleright \mathbf{0} \mid \varepsilon]_l$ as the unit, and the scope of a restricted channel is amenable to extrusion. The clause for nodes simply bridges the congruences.

Given a finite family of networks $\{N_i\}_{i \in I}$, we write $\prod_{i \in I} N_i$ for the parallel composition of $N_1 \mid \cdots \mid N_n$ for non-empty $I = \{1, \dots, n\}$, otherwise $[\mathbf{0} \triangleright \mathbf{0} \mid \varepsilon]_l$.

Definition 4 (Reduction Relation). *We define the reduction relation on networks $N \to_G N'$, parameterised over an arbitrary connectivity graph $G \subseteq L \times L$, as the smallest relation satisfying the rules given in Fig. 1.*

$$\frac{i \in I \quad s \# \psi, \psi_i, R, R_i \quad (l, l_i) \in G}{[a_!(s).P \triangleright R \,|\, \psi]_l \mid \prod_{i \in I}[a_?(s).Q_i \triangleright R_i \,|\, \psi_i]_{l_i} \to_G (\nu\, s)([P \triangleright R \,|\, \psi]_l \mid \prod_{i \in I}[Q_i \triangleright R_i \,|\, \psi_i]_{l_i})}\,[\text{RInit}]$$

$$\frac{i \in I \quad (l, l_i) \in G \quad \mathsf{cnt}(s, \psi) = \mathsf{cnt}(s, \psi_i)}{[\breve{s}_!\langle e\rangle.P \triangleright R \,|\, \psi]_l \mid \prod_{i \in I}[s_?(x_i).Q_i \triangleright R_i \,|\, \psi_i]_{l_i} \to_G [P \triangleright R \,|\, \psi \,|\, s]_l \mid \prod_{i \in I}[Q_i\{e/x_i\} \triangleright R_i \,|\, \psi_i \,|\, s]_{l_i}}\,[\text{RScatter}]$$

$$\frac{i \in I \quad (l_i, l) \in G \quad \mathsf{cnt}(s, \psi) = \mathsf{cnt}(s, \psi_i) \quad e = \odot_{i \in I} e_i}{[\breve{s}_?(x).P \triangleright R \,|\, \psi]_l \mid \prod_{i \in I}[s_!\langle e_i\rangle.Q_i \triangleright R_i \,|\, \psi_i]_{l_i} \to_G [P\{e/x\} \triangleright R \,|\, \psi \,|\, s]_l \mid \prod_{i \in I}[Q_i \triangleright R_i \,|\, \psi_i \,|\, s]_{l_i}}\,[\text{RGather}]$$

$$\frac{}{[s_!\langle e\rangle.Q \triangleright R \,|\, \psi]_l \to_G [Q \triangleright R \,|\, \psi \,|\, s]_l}\,[\text{RLoss}]$$

$$\frac{i \in I \quad (\ell : Q_i) \in B_i \quad (l, l_i) \in G \quad \mathsf{cnt}(s, \psi) = \mathsf{cnt}(s, \psi_i)}{[\breve{s} \oplus \ell.P \triangleright R \,|\, \psi]_l \mid \prod_{i \in I}[s \,\&\, B_i \triangleright R_i \,|\, \psi_i]_{l_i} \to_G [P \triangleright R \,|\, \psi \,|\, s]_l \mid \prod_{i \in I}[Q_i \triangleright R_i \,|\, \psi_i \,|\, s]_{l_i}}\,[\text{RSel}]$$

$$\frac{\Vdash \varphi}{[\mathsf{if}\ \varphi\ \mathsf{then}\ P\ \mathsf{else}\ Q \triangleright R \,|\, \psi]_l \to_G [P \triangleright R \,|\, \psi]_l}\,[\text{RTrue}]$$

$$\frac{\not\Vdash \varphi}{[\mathsf{if}\ \varphi\ \mathsf{then}\ P\ \mathsf{else}\ Q \triangleright R \,|\, \psi]_l \to_G [Q \triangleright R \,|\, \psi]_l}\,[\text{RFalse}]$$

$$\frac{}{[s_?(x).P \triangleright R \,|\, \psi]_l \to_G [R \triangleright R \,|\, \varepsilon]_l}\,[\text{RRecover}]$$

$$\frac{}{[s \,\&\, \{\ell_i : Q_i\}_{i \in I} \triangleright R \,|\, \psi]_l \to_G [R \triangleright R \,|\, \varepsilon]_l}\,[\text{RRecoverSel}]$$

$$\frac{N \equiv N' \quad N' \to_G M' \quad M' \equiv M}{N \to_G M}\,[\text{RCong}]$$

$$\frac{N \to_G N'}{N \,|\, M \to_G N' \,|\, M}\,[\text{RPar}] \qquad \frac{N \to_G N'}{(\nu\, n)N \to_G (\nu\, n)N'}\,[\text{RRes}]$$

Fig. 1. Reduction rules of the broadcast calculus.

With a connectivity graph we can capture spatial distribution of nodes that is common in ad-hoc and wireless sensor networks. Note that connectivity graph is an arbitrary relation and is not required to be neither reflexive nor symmetric.

By not requiring symmetry, allows us to capture the asymmetry of communication often found in WSNs; where a typical situation is that a special node, base station, has more powerful antenna that allows it to broadcast to all the nodes in the network, whereas the more distant nodes with less powerful antennas cannot broadcast directly to the base station. We do not require reflexivity of a connectivity graph, that is, a node may not be able to communicate with itself, for generality reasons as we do not need reflexivity for our results to hold. We do not model mobility of the nodes (nodes are not able to change location), meaning that the connectivity graph G does not evolve during reductions. Without mobility we can still capture many classes of networks.

We have chosen communication to be synchronous in our system since it seems that kind of communication is more prevalent in broadcasting systems. This means that a sender synchronises with a subset of receivers. We believe that reformulating the system to introduce asynchrony, would require the definition of a queue process that is able to hold messages in an ordered fashion [11].

The rule [RINIT] establishes a session between the requesting process node and accepting process nodes. It is a broadcast communication pattern (one-to-many) where there might not be any accepting process nodes (i.e., $I = \emptyset$). The session channel is chosen fresh for all session states and recovery processes of participants. Note that the session channels s in the requesting process P are annotated as $\breve{s}$ by the well-formedness condition on processes (Definition 1). Let us call with respect to the session channel s the process P prefixed with the subject $\breve{s}$ *parent*, and with subject s *child*.

The rule [RSCATTER] states that the parent broadcasts e to the children that continue by substituting e for x_i. Both the parent and children advance their session states by emitting the corresponding session channel in their session state stores. This is the case for all reduction rules concerning interaction within the session. There is also a precondition that session partners have advanced the same number of steps in the session.

The reduction [RGATHER] is dual to [RSCATTER] in the sense that the communication is reversed (many-to-one): the parent receives from children. The rule is an abstraction of a communication pattern that consists of multiple broadcasts from each individual child. The iterated result e is collected as the product of the aggregation operation $\odot_{i \in I} e_i$ that is defined to be $e_1 \odot \ldots \odot e_n$ for nonempty $I = \{1, \ldots, n\}$ and $\mathbf{1}$ in case $I = \emptyset$. We have not assumed that the $\odot$ is commutative, thus different orderings of parallel components can lead to a different product in the reduction. Here again I might be empty. However, here the advance of parent does not correspond to a message loss, but that the node prematurely terminated the gathering operation. For example, in a real network the node could have a time limit for gathering data. When I is empty, we use the unit $\mathbf{1}$ as the product. This pattern is common in ad-hoc networks; see, for example, the RIME communication stack [8] for wireless sensor networks.

The rule [RSEL] is similar to [RSCATTER]; however, here the parent selects the branch that children should take. [RLOSS] models message loss for a child. Note that it still emits a session channel in its session stores. As noted before, message

loss for the parent is already captured by [RSCATTER] and [RPAR]. There is no corresponding dual rule to [RSEL]: the opposite communication pattern would require an assumption of a non-trivial underlying protocol where the child nodes would establish a consensus on the branch to be taken by the parent. This is unrealistic in the unreliable setting.

The communication rules [RINIT], [RSCATTER], [RGATHER], and [RSEL] adhere to the communication graph G, that is, there needs to be a connection between the parent and children locations. We don't assume that this relation is symmetric.

[RCONG], [RRES] and [RPAR] are standard, but note that [RPAR] is essential for capturing the unreliability of the medium since it can be used to exclude potential communication partners. [RTRUE] and [RFALSE] are self-explanatory.

Rules [RRECOVER] and [RRECOVERSEL] state that a child node may recover from a session, by replacing their running process with the recovery process R. The recovery is hard and drops all the sessions that the node was a participant of and clears the session state store. The purpose of these rules is to recover from the situation where the processes cannot continue due to a loss of messages that result in diverged session states. However, note that the rules do not have a condition that triggers recovery allowing nodes to act autonomously. This means that the recovery behaves nondeterministically and the nodes can recover even though the session state has not diverged. Having recovery nondeterministic we abstract over other possible sources of failure, e.g., a timeout of a message reception.

The semantics of [RGATHER] can be implemented with series of broadcasts to the parent. Thus, the calculus can be encoded in a calculus with just broadcast albeit with added level of indirection. In our previous work [13], we have such an encoding to broadcast psi-calculi [3], and we conjecture that the semantics weakly corresponds to our previous encoding.

3 Session Types

In this section, we introduce the session type system for the broadcast calculus of Section 2 and the main results of the paper: type soundness and safety. Remarkably, the types that we use are the standard binary session types as introduced by Honda et al. [10]. However, we have no session delegation.

Definition 5 (Session Type). *Let $\mathcal{B}$ be a set of* base types *ranged over by β. Then, the session types are inductively defined by the following grammar:*

$$T \quad ::= \quad {!}\beta.T \;\mid\; {?}\beta.T \;\mid\; \oplus\{\ell_i : T_i\}_{i\in I} \;\mid\; \&\{\ell_i : T_i\}_{i\in I} \;\mid\; \mathsf{end} \;\mid\; t \;\mid\; \mu t.T$$

$\mu t.T$ is a binder and binds free occurrences of t in T. We define a capture avoiding substitution on types $\{T/t\}$ in the usual way. As usual, we identify recursive types with their expansion: $\mu t.T = T\{\mu t.T/t\}$.

We define the duality operation on types $\overline{T}$ recursively as follows:

$$\overline{\mathsf{end}} = \mathsf{end} \qquad \overline{{!}\beta.T} = {?}\beta.\overline{T} \qquad \overline{\oplus\{\ell_i : T_i\}_{i\in I}} = \&\{\ell_i : \overline{T}_i\}_{i\in I} \qquad \overline{t} = t$$

$$\overline{{?}\beta.T} = {!}\beta.\overline{T} \qquad \overline{\&\{\ell_i : T_i\}_{i\in I}} = \oplus\{\ell_i : \overline{T}_i\}_{i\in I} \qquad \overline{\mu t.T} = \mu t.\overline{T}$$

We say that two types T_1 and T_2 are dual if $\overline{T}_1 = T_2$. Note $\overline{\overline{T}} = T$ for any T.

We define a transition relation on types that we will use in the typing rules to obtain the session type consistent with the session state of a node.

Definition 6 (Type Advancement). *The relation $T \to T'$ is defined inductively as follows:*

$$\frac{}{?\beta.T \to T} \qquad \frac{}{!\beta.T \to T} \qquad \frac{i \in I}{\oplus\{\ell_i : T_i\}_{i\in I} \to T_i} \qquad \frac{i \in I}{\&\{\ell_i : T_i\}_{i\in I} \to T_i}$$

We also define a type n-advancement relation $T \to^n T'$ that says T' is reached in n advancements from T, by induction, as follows:

$$\frac{}{T \to^0 T} \qquad \frac{T \to^n T'' \quad T'' \to T'}{T \to^{n+1} T'}$$

Type advancement for recursive types is obtained by expansion. We will typically use n-advancement relation to advance with regard to a session store ψ as $T \to^{\mathsf{cnt}(s,\psi)} T'$.

Definition 7 (Typing Context). *We define shared Γ and linear Δ typing contexts by mutual induction as follows:*

$$\begin{array}{rcl} \gamma & ::= & x : \beta \mid a : T \mid X : \Delta \\ \delta & ::= & s : T \mid \breve{s} : T \end{array}, \qquad \begin{array}{rcl} \Gamma & ::= & \varepsilon \mid \Gamma, \gamma \\ \Delta & ::= & \varepsilon \mid \Delta, \delta \end{array}.$$

We denote with Δ, Δ' the concatenation of contexts Δ and Δ', and similarly for shared contexts Γ, Γ'. We make use of the function subj *defined on γ that extracts the subject as follows* $\mathrm{subj}(x : \beta) = x$, $\mathrm{subj}(a : T) = a$, *and* $\mathrm{subj}(X : \Delta) = X$.

We also define a typing context for the expressions and conditions:

$$\Gamma_E \quad ::= \quad \varepsilon \mid \Gamma_E, x : \beta.$$

By abuse of notation, we denote the restriction of shared typing context Γ to expression context by Γ_E. A domain of a context Γ, and Δ are defined as follows:

$$\mathrm{dom}(\Gamma, \gamma) = \{\mathrm{subj}(\gamma)\} \cup \mathrm{dom}(\Gamma), \qquad \begin{array}{l} \mathrm{dom}(\Delta, s : T) = \{s\} \cup \mathrm{dom}(\Delta) \\ \mathrm{dom}(\Delta, \breve{s} : T) = \{s\} \cup \mathrm{dom}(\Delta) \end{array}.$$

where $\mathrm{dom}(\varepsilon) = \emptyset$ *in both cases.*

In turn, we define the notion of freshness as follows: $\kappa \# \Delta$ is defined as $s \notin \mathrm{dom}(\Delta)$ where $\kappa = s$ or $\kappa = \breve{s}$, and pronounced as κ is fresh for Δ. Similarly, we define $x \# \Gamma$, $a \# \Gamma$, and $X \# \Gamma$ as $x \notin \mathrm{dom}(\Gamma)$, $a \notin \mathrm{dom}(\Gamma)$, and $X \notin \mathrm{dom}(\Gamma)$, respectively.

In the linear context, we distinguish the endpoints as we do in the process syntax. The choice of representing typing contexts as lists and not as sets, as is perhaps more common, has technical convenience. It is important to have multiple copies of a session channel and its type for child nodes. This as we shall see trivialises the parallel typing rule to a simple concatenation of parallel contexts. We extend the notion of type advancement of Definition 6 to linear context advancement based on a session state assertion of Definition 2.

Definition 8 (Linear Context Advancement). *Let ψ be an arbitrary session state assertion of Definition 2. Then, we define linear context type advancement $\Delta \to^{\psi} \Delta'$ inductively as follows:*

$$\frac{}{\varepsilon \to^{\psi} \varepsilon} \qquad \frac{\Delta \to^{\psi} \Delta' \quad T \to^{\mathsf{cnt}(s,\psi)} T'}{\Delta, s : T \to^{\psi} \Delta', s : T'} \qquad \frac{\Delta \to^{\psi} \Delta' \quad T \to^{\mathsf{cnt}(s,\psi)} T'}{\Delta, \breve{s} : T \to^{\psi} \Delta', \breve{s} : T'}$$

To take an example of linear context advancement, let $\psi = s_1 \,|\, s_1 \,|\, s_2$ be session state and $\Delta = \varepsilon, s_1 : {}_{!}\beta_1.{}_{?}\beta_1.{}_{!}\beta_2.\mathsf{end}, s_2 : {}_{!}\beta_2.\mathsf{end}, s_3 : {}_{?}\beta_2.\mathsf{end}$ a linear context. Then, $\Delta \to^{\psi} s_1 : {}_{!}\beta_2.\mathsf{end}, s_2 : \mathsf{end}, s_3 : {}_{?}\beta_2.\mathsf{end}$.

Definition 9 (Typing Judgement). *We parameterise our type system on the typing judgements of expressions and conditions in the following way: let us assume the following to be a typing judgement on expressions, and typing judgement on conditions:*

$$\Gamma_E \succ e : \beta \quad \text{and} \quad \Gamma_E \succ \varphi : \mathsf{bool}.$$

The above judgements are arbitrary but required to satisfy the following conditions: type is preserved by substitutions (i) if $\Gamma_E, x : \beta' \succ e : \beta$ and $\Gamma_E \succ e' : \beta'$, then $\Gamma_E \succ e\{e'/x\} : \beta$; (ii) if $\Gamma_E, x : \beta \succ \varphi : \mathsf{bool}$ and $\Gamma_E \succ e : \beta$, then $\Gamma_E \succ \varphi\{e/x\} : \mathsf{bool}$; and that the aggregation operation does not change the type of resulting expression (iii) if $\Gamma_E \succ e : \beta$ and $\Gamma_E \succ e' : \beta$, then $\Gamma_E \succ e \odot e' : \beta$.

Our typing judgement is of the following form

$$\Gamma; \Delta \vdash N.$$

It is defined as the smallest relation satisfying the rules given in Fig. 2. We call the network N or process P well-typed if for some contexts Γ and Δ it holds that $\Gamma; \Delta \vdash N$ or $\Gamma; \Delta \vdash P$, respectively.

Since our contexts are lists and not sets, we employ structural rules to manage contexts. The rules of note are [TNode] and [TSRes]. Other rules are fairly standard and generalise to multiple partners quite straightforwardly (cf. [10]). [TSRes] rule asserts that under a shared context and a linear context with exactly one parent session type $\breve{s} : T$ for session channel s, and a, possibly empty, sequence of session types for the children $s : \overline{T}, \ldots, s : \overline{T}$, the network N is typed, then the network is typed by closing of the session. The parent session type is dual to all of the children session types, and furthermore N has to be an alpha-variant such that the session channel s is fresh for Δ, Γ. The number of copies of children types $s : \overline{T}$ are determined by the number of parallel components that participate in the session in N. This is ensured by the inductive invariant that rules preserve: whenever $\Gamma; \Delta \vdash P$ for a process P, then all the channels are distinct in Δ.

Note that [TSRes] uses the same type T; however, the nodes may have diverged to a different session state. The [TNode] accounts for the divergence. It states that for an (initial) context Δ that can be advanced by the session state

$$\frac{\Gamma; \Delta \vdash N}{\Gamma'; \Delta' \vdash N}[\text{TExch}] \qquad \frac{\Gamma; \Delta \vdash N \quad \text{subj}(\gamma) \# N, \Gamma}{\Gamma, \gamma; \Delta \vdash N}[\text{TShWeak}]$$

$$\frac{\Gamma; \Delta \vdash P \quad \kappa \# P, \Delta}{\Gamma; \Delta, \kappa : \mathsf{end} \vdash P}[\text{TWeak}] \qquad \frac{}{\Gamma; \varepsilon \vdash \mathbf{0}}[\text{TInact}]$$

$$\frac{\Gamma; \Delta, \check{s} : T \vdash P \quad s \# \Gamma, \Delta}{\Gamma, a : T; \Delta \vdash a_!(s).P}[\text{TReq}] \qquad \frac{\Gamma; \Delta, s : \overline{T} \vdash P \quad s \# \Gamma, \Delta}{\Gamma, a : T; \Delta \vdash a_?(s).P}[\text{TAcc}]$$

$$\frac{\Gamma, x : \beta; \Delta, \kappa : T \vdash P \quad x \# \Gamma}{\Gamma; \Delta, \kappa : {}_?\beta.T \vdash \kappa_?(x).P}[\text{TRcv}] \qquad \frac{\Gamma; \Delta, \kappa : T \vdash P \quad \Gamma_E \succ e : \beta}{\Gamma; \Delta, \kappa : {}_!\beta.T \vdash \kappa_!\langle e \rangle.P}[\text{TSnd}]$$

$$\frac{\Gamma; \Delta, \check{s} : T_j \vdash P \quad j \in I}{\Gamma; \Delta, \check{s} : \oplus\{\ell_i : T_i\}_{i \in I} \vdash \check{s} \oplus \ell_j.P}[\text{TSel}]$$

$$\frac{\Gamma; \Delta, s : T_1 \vdash P_1 \quad \cdots \quad \Gamma; \Delta, s : T_n \vdash P_n}{\Gamma; \Delta, s : \&\{\ell_1 : T_1, \ldots, \ell_n : T_n\} \vdash s \,\&\, \{\ell_1 : P_1, \ldots, \ell_n : P_n\}}[\text{TBr}]$$

$$\frac{\Gamma, X : \Delta; \Delta \vdash P \quad X \# \Gamma}{\Gamma; \Delta \vdash \mu X.P}[\text{TRec}] \qquad \frac{}{\Gamma, X : \Delta; \Delta \vdash X}[\text{TVar}]$$

$$\frac{\Gamma; \Delta \vdash P \quad \Gamma; \Delta \vdash Q \quad \Gamma_E \succ \varphi : \mathsf{bool}}{\Gamma; \Delta \vdash \mathsf{if}\, \varphi\, \mathsf{then}\, P\, \mathsf{else}\, Q}[\text{TCond}] \qquad \frac{\Gamma, a : T; \Delta \vdash N \quad a \# \Gamma}{\Gamma; \Delta \vdash (\nu\, a)N}[\text{TCRes}]$$

$$\frac{\Gamma; \Delta \vdash M \quad \Gamma; \Delta' \vdash N}{\Gamma; \Delta, \Delta' \vdash M \,|\, N}[\text{TPar}] \qquad \frac{\Delta \rightarrow^{\psi} \Delta' \quad \Gamma; \Delta' \vdash P \quad \Gamma; \varepsilon \vdash R}{\Gamma; \Delta \vdash [P \,\triangleright\, R \,|\, \psi]_l}[\text{TNode}]$$

$$\frac{\Gamma; \Delta, \check{s} : T, s : \overline{T}, \ldots, s : \overline{T} \vdash N \quad s \# \Gamma, \Delta}{\Gamma; \Delta \vdash (\nu\, s)N}[\text{TSRes}]$$

Fig. 2. Typing rules. In [TExch], Δ' and Γ' are permutations of Δ and Γ, respectively. We also assume analogous rules to [TExch] and [TShWeak] for processes.

ψ to Δ' such that P is typed under Δ', then the node $[P \,\triangleright\, R \,|\, \psi]_l$ is typed under Δ. The recovery process R types if it does not contain sessions.

In order to state the subject reduction theorem, we need a notion of context inclusion. In the presence of exchange rule [TExch], the following definition that compares members element-wise is sufficient.

Definition 10 (Linear Context Inclusion). *Linear context inclusion $\Delta \subseteq \Delta'$ relation is defined inductively as follows*

$$\frac{\Delta \subseteq \Delta'}{\Delta, \kappa : T \subseteq \Delta', \kappa : T} \qquad \frac{\Delta, \kappa : T \subseteq \Delta' \quad \kappa \neq \kappa'}{\Delta, \kappa : T \subseteq \Delta', \kappa' : T} \qquad \frac{}{\varepsilon \subseteq \Delta}$$

Lemma 1 (Congruence Invariance). *If $N \equiv N'$, then $\Gamma; \Delta \vdash N$ if, and only if, $\Gamma; \Delta \vdash N'$.*

Proof. By induction on the derivation of $N \equiv N'$. The full proof is given in Appendix A.1.

Lemma 2 (Substitution). *Whenever* $\Gamma \succ e : \beta$ *and* $\Gamma, x : \beta; \Delta \vdash P$*, then* $\Gamma; \Delta \vdash P\{e/x\}$*.*

Proof. By induction on the depth of derivation of $\Gamma, x : \beta; \Delta \vdash P$. Since substitutions only affect variables and expressions, the result easily follows from the requirements on expression substitution of Definition 9.

Theorem 1 (Subject Reduction). *If* $\Gamma; \Delta \vdash N$ *and* $N \to_G N'$*, then there exist* $\Delta' \subseteq \Delta$ *such that* $\Gamma; \Delta' \vdash N'$*.*

Proof. By induction on the depth of derivation of $N \to_G N'$. The full proof is given in Appendix A.2.

Note that a linear context does not change in terms of types that it describes, but only that it may reduce in size. The reason is twofold. First, because [TNode] advances the type in accordance with the sessions state of a node, the same type from Δ may be chosen as the initial type in Δ' for each session channel. Second, a node may recover during the reduction (due to rules [RRecover] and [RRecoverSel]) and hence dropping all the sessions it participated in and in turn these sessions are discarded from the initial context Δ. Also note that subject reduction property is unaffected by the structure of the connectivity graph G.

The subject reduction theorem still holds if we allow recovery for any process not just input prefix, but we need to modify [TSRes] rule to possibly exclude $\breve{s}$ from the context.

Definition 11 (Error Network). *Let s-prefix be a network of the following form*

$$\begin{aligned}
\text{SCT} &= [\breve{s}_!\langle e\rangle.P_1 \,\triangleright\, R_1 \,|\, \psi_1]_{l_1}\\
\text{GTH} &= [\breve{s}_?(x).P_2 \,\triangleright\, R_2 \,|\, \psi_2]_{l_2}\\
\text{SEL} &= [\breve{s} \oplus \ell.P_3 \,\triangleright\, R_3 \,|\, \psi_3]_{l_3}\\
\text{RCV} &= [s_?(x').P_4 \,\triangleright\, R_4 \,|\, \psi_4]_{l_4}\\
\text{SND} &= [s_!\langle e'\rangle.P_5 \,\triangleright\, R_5 \,|\, \psi_5]_{l_5}\\
\text{BR} &= [s \,\&\, \{\ell_1 : Q_1, \ldots, \ell_n : Q_n\} \,\triangleright\, R_6 \,|\, \psi_6]_{l_6}
\end{aligned}$$

such that all above session stores are in the same state with regard to s*, that is, for some* m *and for* $k = 1, \ldots, 6$*, we have* $\mathsf{cnt}(s, \psi_k) = m$*.*

An invalid *s-redex is one of the following parallel compositions of s-prefixes:*

SCT | SCT	GTH | GTH	SEL | SEL	
SCT | GTH	SCT | SEL	SCT | SND	SCT | BR
GTH | SEL	GTH | RCV	GTH | BR	
SEL | RCV	SEL | SND		
RCV | SND	RCV | BR		
SND | BR.			

A network N *is called an* error network *whenever there exists an invalid s-redex* M *such that for some network* N' *the following holds*

$$N \equiv (\nu\, \widetilde{n}, s)(N' \,|\, M).$$

Equivalently, a valid network is a parallel composition of nodes of which all s-prefixes that are in the same session state, consists of at most one scatter prefix (resp. gather, selection) and many receive prefixes (resp. send, branch).

Theorem 2 (Type Safety). *Let $\Gamma; \Delta \vdash N$ and $N \rightarrow_G^* N'$ then N' is* not *an error network.*

Proof. The proof follows easily from Theorem 1 and the fact that an error network is not well-typed.

Type safety states that a well-typed network has always a possibility of communication. A well-typed network never reduces to a state where the session stores are in the same state but there is a prefix mismatch.

4 Data Aggregation in a Wireless Sensor Network

We express an aggregation protocol in a wireless sensor network using our framework. A wireless sensor network consists of spatially distributed nodes that sense environment data. In this example, we use the aggregation function max that computes the maximum of two integers, e.g., temperature of the environment.

The node called sink initiates a data collection algorithm in the network by requesting the aggregated values of its neighbouring nodes. It then disseminates the maximum of these values to the network by sending the aggregate again to its neighbours. A node that receives a request then: (1) proceeds in turn to initiate the same process as the sink by requesting aggregated values from its neighbourhood; (2) aggregates the received values with its own value and sending it to the request node; (3) receives the maximum value originating from the sink; and (4) forwards the maximum value to its neighbours.

The above interaction can be described abstractly from the initiator node perspective by the session type

$$T = {}_{?}\mathsf{int}.{}_{!}\mathsf{int}.\mathsf{end}.$$

That is, first it requests and receives an aggregate value set from its neighbours, and then proceeds by sending the maximum value to its neighbours. Nodes then participate in two sessions: a session that they accept from the initiator, and a session that they initiate themselves to start the aggregation. The former is described by the dual type $\overline{T}$ and the latter by T.

Let a set of expressions $\mathcal{E}$ be defined by the following:

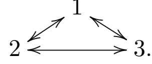
$$e, e' \quad ::= \quad x \;\mid\; \mathsf{max}(e, e') \;\mid\; n \qquad (n \in \mathbb{N}, x \in \mathcal{X}).$$

Let the set of base types $\mathcal{B}$ be $\{\mathsf{int}\}$. Then, define the expected typing judgements for expressions $\Gamma, x : \mathsf{int} \succ x : \mathsf{int}$, $\Gamma \succ n : \mathsf{int}$ for any $n \in \mathbb{N}$, and if $\Gamma \succ e : \mathsf{int}$ and $\Gamma \succ e' : \mathsf{int}$, then $\Gamma \succ \mathsf{max}(e, e') : \mathsf{int}$. The set of conditions $\mathcal{F}$ is empty. We define the aggregation opertion $e \odot e'$ as $\mathsf{max}(e, e')$, and the unit $\mathbf{1}$ as $0 \in \mathbb{N}$.

We implement sink and nodes as the following processes according to the above description:

$$\begin{aligned}\textsc{Sink} &= a_!(s).\breve{s}_?(x).\breve{s}_!\langle x\rangle.\mathbf{0}\\ \textsc{Node}_i &= a_?(s).a_!(s_c).\breve{s}_{c?}(x).s_!\langle \mathsf{max}(x, v_i)\rangle.s_?(y).\breve{s}_{c!}\langle y\rangle.\mathbf{0}\end{aligned}$$

where $v_i \in \mathbb{N}$ for $i = 2, 3$. For simplicity, let us model a three node network. Let the set of locations be $\mathcal{L} = \{1, 2, 3\}$, and the connectivity graph G be $\{(1, 2), (1, 3), (2, 1), (2, 3), (3, 1), (3, 2)\}$ that we visually depict as



The network N implementing the aggregation protocol is then the following

$$[\textsc{Sink} \triangleright \textsc{Sink} \,|\, \varepsilon]_1 \;|\; [\textsc{Node}_2 \triangleright \textsc{Node}_2 \,|\, \varepsilon]_2 \;|\; [\textsc{Node}_3 \triangleright \textsc{Node}_3 \,|\, \varepsilon]_3$$

where the recovery is simply a restart of the process. The network is well-typed:

$$a : T; \varepsilon \vdash N.$$

It is fairly easy to see that Sink is well-typed after applying [TPar] with [TReq]

$$a : T; \breve{s} : {}_?\mathsf{int}.{}_!\mathsf{int}.\mathsf{end} \vdash \breve{s}_?(x).\breve{s}_!\langle x\rangle.\mathbf{0}$$

and then type of the session channel $\breve{s}$ matches the process. Each node can be typed similarly, however, we have two sessions by applying [TAcc] and [TReq]

$$a : T; s : {}_!\mathsf{int}.{}_?\mathsf{int}.\mathsf{end}, \breve{s}_c : {}_?\mathsf{int}.{}_!\mathsf{int}.\mathsf{end} \;\vdash\; \breve{s}_{c?}(x).s_!\langle \mathsf{max}(x, v_i)\rangle.s_?(y).\breve{s}_{c!}\langle y\rangle.\mathbf{0}$$

note that the type for s is dual to that of $\breve{s}$ of sink above and that the interactions on two sessions match both of the types. This holds by easy applications of [TSnd] and [TRcv].

Thus, the operation of the network N is safe. Let us take an example of a run of the protocol from N over G where N first reduces to the following by establishing a shared session between the sink and the nodes 2 and 3

$$(\nu\, s)([\breve{s}_?(x).\breve{s}_!\langle x\rangle.\mathbf{0} \triangleright \textsc{Sink} \,|\, \varepsilon]_1 \,|\, N_2 \,|\, N_3)$$

where, for $k \in 2, 3$, $N_k = [a_!(s_c).\breve{s}_{c?}(x).s_!\langle \mathsf{max}(x, v_k)\rangle.s_?(y).\breve{s}_{c!}\langle y\rangle.\mathbf{0} \triangleright \textsc{Node}_k \,|\, \varepsilon]_k$ Then, the nodes 2 and 3 in turn request a session of their own [RInit], but since there are no other process accepting a session (neither the sink, nor the nodes themselves) they establish a session with just themselves:

$$N'_k = (\nu\, s_k)[\breve{s}_{k?}(x).s_!\langle \mathsf{max}(x, v_k)\rangle.s_?(y).\breve{s}_{k!}\langle y\rangle.\mathbf{0} \triangleright \textsc{Node}_k \,|\, \varepsilon]_k$$

for $k = 1, 2$. The nodes then again in turn proceed to gather [RGather]. However, as there are no partners they simply use the unit 0 as the result and emit s_k to their session state:

$$N'_k = (\nu\, s_k)[s_!\langle \mathsf{max}(0, v_k)\rangle.s_?(y).\breve{s}_{k!}\langle y\rangle.\mathbf{0} \triangleright \textsc{Node}_k \,|\, s_k]_k$$

for $k = 1, 2$. The sink may then proceed the gather process and we obtain the network N' where the sink is ready to disseminate the maximum value

$$(\nu\, s, s_2, s_3)([\breve{s}_!\langle \mathsf{max}(\mathsf{max}(0, v_2), \mathsf{max}(0, v_3))\rangle.\mathbf{0} \,\triangleright\, \textsc{Sink} \,|\, s]_1 \;|\; [s_?(y).\breve{s}_{2!}\langle y\rangle.\mathbf{0} \,\triangleright\, \textsc{Node}_2 \,|\, s_2 \,|\, s]_2 \;|\; [s_?(y).\breve{s}_{3!}\langle y\rangle.\mathbf{0} \,\triangleright\, \textsc{Node}_3 \,|\, s_3 \,|\, s]_3)$$

At this point the network is still well-typed $a : T; \varepsilon \vdash N'$. After applications of [TSRes] we can type with the context $\Delta = \breve{s} : T, s : \overline{T}, s : \overline{T}, \breve{s}_2 : T, \breve{s}_3 : T$. And for example the sink types with [TNode], after splitting the context Δ with [TPar], by advancing the context with the session state $\psi = s$ as $\breve{s} : {}_?\mathsf{int}.{}_!\mathsf{int}.\mathsf{end} \rightarrow^s \breve{s} : {}_!\mathsf{int}.\mathsf{end}$.

Now, suppose only node 2 receives the scatter from the sink.

$$(\nu\, s, s_2, s_3)([\mathbf{0} \triangleright \textsc{Sink} \,|s|s]_1 \;|\; [\breve{s}_{2!}\langle \mathsf{max}(\mathsf{max}(0, v_2), \mathsf{max}(0, v_3))\rangle.\mathbf{0} \triangleright \textsc{Node}_2 \,|\, s_2|s|s]_2 \;|\; [s_?(y).\breve{s}_{3!}\langle y\rangle.\mathbf{0} \,\triangleright\, \textsc{Node}_3 \,|\, s_3 \,|\, s]_3)$$

The node 2 can proceed by broadcasting its received result to ether, and after that node 3 recovers giving us one of the final states of the protocol:

$$(\nu\, s, s_2, s_3)([\mathbf{0} \triangleright \textsc{Sink} \,|s|s]_1 \;|\; [\mathbf{0} \triangleright \textsc{Node}_2 \,|\, s_2|s|s\,|s_2]_2) \;|\; [\textsc{Node}_3 \,\triangleright\, \textsc{Node}_3 \,|\, \varepsilon]_3.$$

The configuration of the network is well-typed; the session stores are consistent with the initial type T. Even with failure to receive a message the run of the protocol is successful, there are no communication errors and the communication is best effort as intended.

This simple example has a terminal state. In order to implement continuous aggregation we can simply redefine the network with recursion as usual and is typed by the same context $a : T; \varepsilon$. For example, we can redefine Sink to be $\textsc{SinkRec} = \mu X.a_!(s).\breve{s}_?(x).\breve{s}_!\langle x\rangle.X$.

Suppose that the sink is defined instead with session prefixes swapped as $\textsc{Sink}' = a_!(s).\breve{s}_!\langle v_0\rangle.\breve{s}_?(x).\mathbf{0}$. That is, it is not well-typed $a : T; \varepsilon \not\vdash \textsc{Sink}'$. Then, the network still reduces, but no communication takes place; furthermore, communication is not even possible between nodes and the sink. Let us take a minimal but sufficient example to illustrate. The network $[\textsc{Sink}' \triangleright \textsc{Sink}' \,|\, \varepsilon]_1 \,|\, [\textsc{Node} \triangleright \textsc{Node} \,|\, \varepsilon]_2$ reduces to the following, by first establishing a session between the sink and node, and node requests a session and vacuously gathers:

$$(\nu\, s, s')([\breve{s}_!\langle v_0\rangle.\breve{s}_?(x).\mathbf{0} \,\triangleright\, \textsc{Sink}' \,|\, \varepsilon]_1 \,|\, [s_!\langle \mathsf{max}(0, v_i)\rangle.s_?(y).\breve{s}'_!\langle y\rangle.\mathbf{0} \,\triangleright\, \textsc{Node} \,|\, s']_2)$$

Even though the two nodes are in the same session state with regard to session s, there is communication mismatch; this is an error network in the sense of Definition 11. The sink will scatter its value without any possibility of the node receiving it, the node will then lose its message by [RLoss]

$$(\nu\, s, s')([\breve{s}_?(x).\mathbf{0} \,\triangleright\, \textsc{Sink}' \,|\, \varepsilon]_1 \,|\, [s_?(y).\breve{s}'_!\langle y\rangle.\mathbf{0} \,\triangleright\, \textsc{Node} \,|\, s']_2)$$

and then will need to recover as both nodes are expecting a receive:

$$(\nu\, s, s')([\breve{s}_?(x).\mathbf{0} \,\triangleright\, \textsc{Sink}' \,|\, \varepsilon]_1 \,|\, [\textsc{Node} \,\triangleright\, \textsc{Node} \,|\, \varepsilon]_2).$$

Thus, sink proceeds successfully while node recovered and no communication occurs:

$$(\nu\, s, s')([\mathbf{0} \triangleright \textsc{Sink}' \,|\, \varepsilon]_1 \,|\, [\textsc{Node} \triangleright \textsc{Node} \,|\, \varepsilon]_2).$$

This illustrates the point that our type system does not only guarantee progress, which can always be achieved by assuming that messages are lost and using the recovery processes. It guarantees that progress occurs through communication.

5 Conclusions

We have introduced a process calculus with unreliable broadcast communication and a session type system that guarranties safe communication. We have captured two common operation in broadcasting system with unreliable communication: scatter and gather. Our calculus can tolerate message loss and has the ability to recover. We believe that this sufficiently captures the requirements of systems that are based on unreliable broadcast communication, such as wireless sensor networks.

Reliable broadcasting semantics were proposed in the form of multi-casting in [7]. Types for reliable gather semantics were proposed in [6] but an implementation was never proposed. This is the first time broadcasting and gather are presented in the context of unreliability and message loss. A form of recovery as exception handling was introduced in binary [5] and multipary [4] session types, that defines a complicated procedure for informing and coordinating a set of session endpoints about an exception. The recovery procedure in the above work assumes strong global synchronisation requirements, among the session endpoints. In contrast, our semantics allows for each process to autonomously recover from failure of communication. We believe that our recovery semantics are more natural and general because they can account for local session failures as well. Hüttel and Pratas [12] investigate expressiveness of a similar process calculus with scatter and gather operations.

We are the first to consider a session type system for unreliable communication. Furthermore, we are the first to introduce notions of locality and channel connectivity in session types, putting forward the requirements for developing a session type system in the presence of node distribution.

The system presented here is based on a form of binary session types. As seen in Section 4, in order to implement wireless network algorithms we need to interleave session channels. A future research direction is to develop a more robust multiparty session type system where network roles can interact with multiple roles inside a session. Furthermore, we would like to investigate more elaborate autonomous recovery mechanisms that would allow a node to continue in a session instead of restarting. We also plan to extend the system with node mobility that is important in wireless sensor networks. Node mobility would allow nodes to migrate between locations, introduce and disable nodes in a network. Because of autonomous recovery mechanism, we believe that adding node mobility to our system should not pose difficulties.

Acknowledgements Kouzapas and Gay are supported by the UK EPSRC project "From Data Types to Session Types: A Basis for Concurrency and Distribution" (EP/K034413/1). This research was supported by a Short-Term Scientific Mission grant from COST Action IC1201 (Behavioural Types for Reliable Large-Scale Software Systems).

References

1. COST Action IC1201 (BETTY) Tools (2016), `www.behavioural-types.eu/tools`
2. Ancona, D., Bono, V., Bravetti, M., Castagna, G., Campos, J., Gay, S.J., Giachino, E., Johnsen, E.B., Mascardi, V., Ng, N., Padovani, L., Deniélou, P.M., Gesbert, N., Hu, R., Martins, F., Montesi, F., Neykova, R., Vasconcelos, V.T., Yoshida, N.: Behavioral types in programming languages. Foundations and Trends in Programming Languages (2016), to appear
3. Borgström, J., Huang, S., Johansson, M., Raabjerg, P., Victor, B., Pohjola, J.Å., Parrow, J.: Broadcast psi-calculi with an application to wireless protocols. In: Barthe, G., Pardo, A., Schneider, G. (eds.) SEFM 2011. vol. 7041, pp. 74–89. Springer (2011)
4. Capecchi, S., Giachino, E., Yoshida, N.: Global escape in multiparty sessions. Mathematical Structures in Computer Science 26, 156–205 (2016)
5. Carbone, M., Honda, K., Yoshida, N.: Structured interactional exceptions in session types. In: CONCUR. LNCS, vol. 5201, pp. 402–417. Springer (2008)
6. Castagna, G., Dezani-Ciancaglini, M., Padovani, L.: On global types and multiparty session. Logical Methods in Computer Science 8(1) (2012)
7. Coppo, M., Dezani-Ciancaglini, M., Yoshida, N., Padovani, L.: Global progress for dynamically interleaved multiparty sessions. Mathematical Structures in Computer Science 26(2), 238–302 (2016)
8. Dunkels, A., Österlind, F., He, Z.: An adaptive communication architecture for wireless sensor networks. In: Proceedings of Embedded Networked Sensor Systems. pp. 335–349. SenSys '07, ACM, New York, NY, USA (2007)
9. Hennessy, M., Riely, J.: Resource access control in systems of mobile agents. Information and Computation 173(1), 82 – 120 (2002)
10. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: ESOP. pp. 122–138. Springer-Verlag, London, UK (1998)
11. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: POPL. pp. 273–284 (2008)
12. Hüttel, H., Pratas, N.: Broadcast and aggregation in BBC. In: Gay, S., Alglave, J. (eds.) PLACES. EPTCS, vol. 203, pp. 15–28 (2016)
13. Kouzapas, D., Gutkovas, R., Gay, S.J.: Session types for broadcasting. In: Donaldson, A.F., Vasconcelos, V.T. (eds.) PLACES. EPTCS, vol. 155, pp. 25–31 (2014)

A Proofs

A.1 Congruence invariance proof

Proof (of Lemma 1). We proceed by induction on the derivation of $N \equiv N'$:

case $N \mid M \equiv M \mid N$**.** Let us first show the $\Rightarrow$ direction. Assume $\Gamma; \Delta \vdash N \mid M$.

$$\frac{\Gamma; \Delta_1 \vdash N \quad \Gamma; \Delta_2 \vdash M}{\Gamma; \Delta_1, \Delta_2 \vdash N \mid M}[\text{TPar}]$$

where Δ_1, Δ_2 is a permutation of Δ obtained by the exchange rule. We use $\Gamma; \Delta_1 \vdash N$ and $\Gamma; \Delta_2 \vdash M$ to construct

$$\frac{\Gamma; \Delta_2 \vdash M \quad \Gamma; \Delta_1 \vdash N}{\Gamma; \Delta_2, \Delta_1 \vdash M \mid N}[\text{TPar}]$$

Obviously, Δ_2, Δ_1 is a permutation of Δ. We are done with this case. The other direction is analogous.

case $(M \mid N) \mid S \equiv M \mid (N \mid S)$**.** Assume $\Gamma; \Delta \vdash (M \mid N) \mid S$. Then the derivation tree of the assumption is

$$\frac{\dfrac{\Gamma; \Delta_1 \vdash M \quad \Gamma; \Delta_2 \vdash N}{\Gamma; \Delta_1, \Delta_2 \vdash M \mid N}[\text{TPar}] \quad \Gamma; \Delta_3 \vdash S}{\Gamma; \Delta_1, \Delta_2, \Delta_3 \vdash (M \mid N) \mid S}[\text{TPar}]$$

where $\Delta_1, \Delta_2, \Delta$ is a permutation of Δ. Then, using the leafs of the above tree we obtain

$$\frac{\Gamma; \Delta_1 \vdash M \quad \dfrac{\Gamma; \Delta_2 \vdash N \quad \Gamma; \Delta_3 \vdash S}{\Gamma; \Delta_2, \Delta_3 \vdash M \mid N}[\text{TPar}]}{\Gamma; \Delta_1, \Delta_2, \Delta_3 \vdash M \mid (N \mid S)}[\text{TPar}]$$

We are done with this case. The opposite direction is analogous.

case $(\nu a)N \mid M \equiv (\nu a)(N \mid M)$**.** Assume $\Gamma; \Delta \vdash (\nu a)N \mid M$. From assumption we obtain the following tree

$$\frac{\dfrac{\Gamma, a : T; \Delta_1 \vdash N \quad a \# \Gamma}{\Gamma; \Delta_1 \vdash (\nu a)N}[\text{TCRes}] \quad \Gamma; \Delta_2 \vdash M}{\Gamma; \Delta_1, \Delta_2 \vdash (\nu a)N \mid M}[\text{TPar}]$$

We then derive the tree where we apply the weaking rule [TShWeak] since by assumption $a \# M$:

$$\frac{\dfrac{\Gamma, a : T; \Delta_1 \vdash N \quad \dfrac{\Gamma; \Delta_2 \vdash M}{\Gamma, a : T; \Delta_2 \vdash M}[\text{TShWeak}]}{\Gamma, a : T; \Delta_1, \Delta_2 \vdash N \mid M}[\text{TPar}] \quad a \# \Gamma}{\Gamma; \Delta_1, \Delta_2 \vdash (\nu a)(N \mid M)}[\text{TCRes}]$$

The other direction is similar because again by assumption we may use the weaking rule to obtain the right premise for the [TCRes] rule.

case $(\nu s)N \mid M \equiv (\nu s)(N \mid M)$**.** Assume $\Gamma; \Delta \vdash (\nu s)N \mid M$. From the assumption we obtain the following tree

$$\dfrac{\dfrac{\Gamma; \Delta_1, \breve{s} : T, s : \overline{T}, \ldots, s : \overline{T} \vdash N \quad s \# \Gamma, \Delta_1}{\Gamma; \Delta_1 \vdash (\nu s)N}\text{[TSRes]} \quad \Gamma; \Delta_2 \vdash M}{\Gamma; \Delta_1, \Delta_2 \vdash (\nu s)N \mid M}\text{[TPar]}$$

We construct the following by noting that $s \# \Gamma, \Delta_1, \Delta_2$ implies $s \# \Gamma, \Delta_1$. Note that in the tree before applying [TPar] we use the exchange rule.

$$\dfrac{\dfrac{\dfrac{\Gamma; \Delta_1, \breve{s} : T, s : \overline{T}, \ldots, s : \overline{T} \vdash N \quad \Gamma; \Delta_2 \vdash M}{\Gamma; \Delta_1, \breve{s} : T, s : \overline{T}, \ldots, s : \overline{T}, \Delta_2 \vdash N \mid M}\text{[TPar]}}{\Gamma; \Delta_1, \Delta_2, \breve{s} : T, s : \overline{T}, \ldots, s : \overline{T} \vdash N \mid M}\text{[TExch]} \quad s \# \Gamma, \Delta_1, \Delta_2}{\Gamma; \Delta_1, \Delta_2 \vdash (\nu s)(N \mid M)}\text{[TSRes]}$$

The opposite direction is similar: as $s \# M$ there is nothing to consume s on M and we can only split the context as above when applying the [TPar] rule.

case $N \equiv [\mathbf{0} \triangleright \mathbf{0} \mid \varepsilon]_l \mid N$**.** Assume $\Gamma; \Delta \vdash N$. Then, the following is a derivation tree

$$\dfrac{\dfrac{\dfrac{}{\Gamma; \varepsilon \vdash \mathbf{0}}\text{[TInact]} \quad \dfrac{}{\Gamma; \varepsilon \vdash \mathbf{0}}\text{[TInact]}}{\Gamma; \varepsilon \vdash [\mathbf{0} \triangleright \mathbf{0} \mid \varepsilon]_l}\text{[TNode]} \quad \Gamma; \Delta \vdash N}{\Gamma; \Delta \vdash [\mathbf{0} \triangleright \mathbf{0} \mid \varepsilon]_l \mid N}\text{[TPar]}$$

The other direction is analogous.

case $[P \triangleright Q \mid \psi]_l \equiv [P' \triangleright Q' \mid \psi']_l$**.** This follows from the fact that the structural congruence preserves the free names.

case $\mu X.P \equiv P\{\mu X.P/X\}$**.** Assume $\Gamma; \Delta \vdash \mu X.P$. From assumption we obtain the folllowing tree

$$\dfrac{\begin{array}{c}\dfrac{}{\Gamma, X : \Delta; \Delta \vdash X}\text{[TVar]} \\ \mathcal{D} \\ \Gamma, X : \Delta; \Delta \vdash P \quad X \# \Gamma\end{array}}{\Gamma; \Delta \vdash \mu X.P}\text{[TRec]}$$

We replace the leafs of the form [TVar] with the above tree and also substitute in the derivation tree X for $\mu X.P$ as follows

$$\dfrac{\begin{array}{c}\dfrac{\Gamma; \Delta \vdash \mu X.P}{\Gamma, X : \Delta; \Delta \vdash \mu X.P}\text{[TShWeak]} \\ \mathcal{D}\{\mu X.P/X\} \\ \\ \Gamma, X : \Delta; \Delta \vdash P\{\mu X.P/X\}\end{array}}{\Gamma; \Delta \vdash P\{\mu X.P/X\}}$$

In the last derivation step, we use a straightforward lemma: $\Gamma, X : \Delta; \Delta \vdash P$ and $X \# P$, then $\Gamma; \Delta \vdash P$. It is easy to see that the above tree satisfies the rules of Fig. 2 as needed. We also used the fact that $X \# \Gamma$ to add $X : \Delta$ to the context. The opposite direction is similar where we collapse the tree.

A.2 Subject reduction proof

Definition 12. *Define the distinct predicate inductively as follows on linear contexts:*

$$\frac{\#(\Delta) \quad \kappa \# \Delta}{\#(\Delta, \kappa : T)} \quad \frac{}{\#(\varepsilon)}$$

We make use of the following lemma that says whenever a *process* (NB: not a network) is well-typed, then all the channels are distinct in the linear context.

Lemma 3. *If $\Gamma; \Delta \vdash P$, then $\#(\Delta)$.*

Proof. By induction on the derivation of $\Gamma; \Delta \vdash P$.

Now we are ready to prove the subject reduction property of our system.

Proof (of Theorem 1). By induction on the depth of derivation of $N \to_G N'$.

case [RLoss]. Assume $[s_!\langle e\rangle.Q \;\triangleright\; R \,|\, \psi]_l \to_G [Q \;\triangleright\; R \,|\, \psi \,|\, s]_l$ and $\Gamma; \Delta \vdash [s_!\langle e\rangle.Q \triangleright R \,|\, \psi]_l$. From the assumption we derive the following tree:

$$\dfrac{\Delta \to^{\psi} \Delta', s : {}_!\beta.T \qquad \dfrac{\Gamma; \Delta', s : T \vdash Q \quad \Gamma_E \succ e : \beta}{\Gamma; \Delta', s : {}_!\beta.T \vdash s_!\langle e\rangle.Q}[\textsc{TSnd}] \qquad \Gamma; \varepsilon \vdash R}{\Gamma; \Delta \vdash [s_!\langle e\rangle.Q \triangleright R \,|\, \psi]_l}[\textsc{TNode}]$$

By Lemma 3, we know that $s \# \Delta'$, thus $\Delta \to^{\psi \,|\, s} \Delta', s : T$. Then, we can type the reduct as follows:

$$\dfrac{\Delta \to^{\psi \,|\, s} \Delta', s : T \quad \Gamma; \Delta', s : T \vdash Q \quad \Gamma; \varepsilon \vdash R}{\Gamma; \Delta \vdash [Q \triangleright R \,|\, \psi \,|\, s]_l}[\textsc{TNode}]$$

case [RScatter]. Assume $[\breve{s}_!\langle e\rangle.P \triangleright R \,|\, \psi]_l \,|\, \prod_{i\in I}[s_?(x_i).Q_i \triangleright R_i \,|\, \psi_i]_{l_i} \to_G [P \triangleright R \,|\, \psi \,|\, s]_l \,|\, \prod_{i\in I}[Q_i\{e/x_i\} \triangleright R_i \,|\, \psi_i \,|\, s]_{l_i}$ and $(l, l_i) \in G$, $\mathsf{cnt}(s, \psi) = \mathsf{cnt}(s, \psi_i)$ for all $i \in I$. Also, assume $\Gamma; \Delta \vdash [\breve{s}_!\langle e\rangle.P \;\triangleright\; R \,|\, \psi]_l \,|\, \prod_{i\in I}[s_?(x_i).Q_i \;\triangleright\; R_i \,|\, \psi_i]_{l_i}$. Without loss of generality, for convenience, suppose $I = \{1, \ldots, n\}$. We obtain the following derivation tree:

$$\dfrac{\dfrac{\Delta_0 \to^{\psi} \Delta'_0, \breve{s} : {}_!\beta.T \qquad \dfrac{\Gamma; \Delta'_0, \breve{s} : T \vdash P \quad \Gamma_E \succ e : \beta}{\Gamma; \Delta'_0, \breve{s} : {}_!\beta.T \vdash \breve{s}_!\langle e\rangle.P} \qquad \Gamma; \varepsilon \vdash R}{\Gamma; \Delta_0 \vdash [\breve{s}_!\langle e\rangle.P \triangleright R \,|\, \psi]_l}[\textsc{TNode}] \qquad \mathcal{D}}{\Gamma; \Delta_0, \Delta_1, \ldots, \Delta_n \vdash [\breve{s}_!\langle e\rangle.P \triangleright R \,|\, \psi]_l \,|\, \prod_{i\in I}[s_?(x_i).Q_i \triangleright R_i \,|\, \psi_i]_{l_i}}[\textsc{TPar}]$$

where $\Delta_0, \Delta_1, \ldots, \Delta_n$ is a permutation of Δ, and $\Delta'_0, \breve{s} : T$ is Δ_0, and $\mathcal{D}$ is as follows where we use [TPar] to split the linear contexts

$$\dfrac{\dfrac{\Delta_1 \to^{\psi} \Delta'_1, s{:}{}_?\beta.T' \qquad \dfrac{\Gamma, x_1{:}\beta; \Delta'_1, s{:}T' \vdash Q_1}{\Gamma; \Delta'_1, s{:}{}_?\beta.T' \vdash s_?(x_1).Q_1}[\textsc{TRcv}] \qquad \Gamma; \varepsilon \vdash R_1}{\Gamma; \Delta_1 \vdash [s_?(x_1).Q_1 \triangleright R_1 \,|\, \psi_1]_{l_1}}[\textsc{TN}] \qquad \mathcal{D}''}{\Gamma; \Delta_1, \ldots, \Delta_n \vdash \prod_{i\in I}[s_?(x_i).Q_i \triangleright R_i \,|\, \psi_i]_{l_i}}[\textsc{TPar}]$$

where $\mathcal{D}''$ is the following

$$\frac{\mathcal{D}'}{\Gamma; \Delta_2, \ldots, \Delta_n \vdash \prod_{i \in I \setminus \{1\}} [s_?(x_i).Q_i \,\triangleright\, R_i \,|\, \psi_i]_{l_i}}$$

We iterate [TPar] and [TNode] rules on $\mathcal{D}'$ to obtain the premises: $\Gamma, x_i : \beta; \Delta'_i, s : T \vdash Q_i$ for $i \in I$.
Observe that $\breve{s} \# \Delta'_0$ by Lemma 3. Thus, type advancement only affects s, i.e., $\Delta_0 \rightarrow^{\psi \,|\, s} \Delta'_0, \breve{s} : T$.
Then, the following is a typing derivation

$$\frac{\dfrac{\Delta \rightarrow^{\psi \,|\, s} \Delta'_0, \breve{s}:T \quad \Gamma;\Delta'_0,\breve{s}:T \vdash P \quad \Gamma;\varepsilon \vdash R}{\Gamma;\Delta_0 \vdash [P \triangleright R \,|\, \psi \,|\, s]_{l_0}} \qquad \dfrac{\mathcal{D}''}{\Gamma;\Delta_1,\ldots,\Delta_n \vdash \prod_{i \in I}[Q_i\{e/x_i\} \triangleright R_i \,|\, \psi_i \,|\, s]_{l_i}}}{\Gamma;\Delta \vdash [P \triangleright R \,|\, \psi \,|\, s]_{l_0} \,|\, \prod_{i \in I}[Q_i\{e/x_i\} \triangleright R_i \,|\, \psi_i \,|\, s]_{l_i}}\,[\text{TPar}]$$

The cases for $\mathcal{D}''$ are similar but we use the substitution Lemma 2 where appropriate.

case [RGather]. For convenience let $I = \{1, \ldots, n\}$. From an assumption we obtain the following tree:

$$\frac{\dfrac{\Delta_0 \rightarrow^{\psi} \Delta'_0, \breve{s} : {}_?\beta.T \qquad \dfrac{\Gamma, x : \beta; \Delta'_0, \breve{s} : T \vdash P}{\Gamma; \Delta'_0, \breve{s} : {}_?\beta.T \vdash \breve{s}_?(x).P}\,[\text{TRcv}] \qquad \Gamma; \varepsilon \vdash R}{\Gamma; \Delta_0 \vdash [\breve{s}_?(x).P \,\triangleright\, R \,|\, \psi]_l}\,[\text{TNode}] \qquad \mathcal{D}}{\Gamma; \Delta_0, \Delta_1, \ldots, \Delta_n \vdash [\breve{s}_?(x).P \,\triangleright\, R \,|\, \psi]_l \,|\, \prod_{i \in I}[s_!\langle e_i \rangle.Q_i \,\triangleright\, R_i \,|\, \psi_i]_{l_i}}$$

where $\mathcal{D}$ is the following

$$\frac{\dfrac{\Delta_1 \rightarrow^{\psi_1} \Delta'_1, s : {}_!\beta.T' \qquad \dfrac{\Gamma; \Delta'_1, s : T' \vdash Q_1 \quad \Gamma_E \succ e_1 : \beta}{\Gamma; \Delta'_1, s : {}_!\beta.T' \vdash s_!\langle e_1 \rangle.Q_1}\,[\text{TSnd}] \qquad \Gamma; \varepsilon \vdash R_1}{\Gamma; \Delta_1 \vdash [s_!\langle e_1 \rangle.Q_1 \,\triangleright\, R_1 \,|\, \psi_1]_{l_1}} \qquad \mathcal{D}'}{\Gamma; \Delta_1, \ldots, \Delta_n \vdash \prod_{i \in I}[s_!\langle e_i \rangle.Q_i \,\triangleright\, R_i \,|\, \psi_i]_{l_i}}$$

By Lemma 3, from $\Gamma; \Delta'_0, \breve{s} : T$ we get that $\breve{s} \# \Delta'_0$, thus $\Delta \rightarrow^{\psi \,|\, s} \Delta'_0, \breve{s} : T$ since the advancement has no affect on Δ'_0. From above we have that $\Gamma_E \succ e_i : \beta$, by the type preservation under aggregation operation (Definition 9 (iii)) we get that $\Gamma_E \succ e : \beta$ where $e = \odot_{i \in I} e_i$. By Lemma 2, from $\Gamma, x : \beta; \Delta'_0, \breve{s} : T \vdash P$ and previous fact, we get $\Gamma; \Delta_0, \breve{s} : T \vdash P\{e/x\}$. By iterating [TPar] and [TNode] rules on $\mathcal{D}'$, we get assumptions: $\Delta_i \rightarrow^{\psi_i} \Delta'_i, s : {}_!\beta.T'$, and $\Gamma; \Delta'_i, s : T' \vdash Q_i$, and $\Gamma_E \succ e_i : \beta$, and $\Gamma; \varepsilon \vdash R_i$ for $i \in I \setminus \{1\}$.
Hence, we can derive the first half:

$$\frac{\dfrac{\Delta_0 \rightarrow^{\psi \,|\, s} \Delta'_0, \breve{s} : T \quad \Gamma; \Delta_0, \breve{s} : T \vdash P\{e/x\} \quad \Gamma; \varepsilon \vdash R}{\Gamma; \Delta_0 \vdash [P\{e/x\} \,\triangleright\, R \,|\, \psi \,|\, s]_l}\,[\text{TNode}] \qquad \mathcal{D}''}{\Gamma; \Delta_0, \Delta_1, \ldots, \Delta_n \vdash [P\{e/x\} \,\triangleright\, R \,|\, \psi \,|\, s]_l \,|\, \prod_{i \in I}[Q_i \,\triangleright\, R_i \,|\, \psi_i \,|\, s]_{l_i}}\,[\text{TPar}]$$

where $\mathcal{D}''$ is

$$\frac{\dfrac{\Delta_1 \rightarrow^{\psi \,|\, s} \Delta'_1, s : T' \quad \Gamma; \Delta'_1, s : T' \vdash Q_1 \quad \Gamma; \varepsilon \vdash R_1}{\Gamma; \Delta_1 \vdash [Q_1 \,\triangleright\, R_1 \,|\, \psi_1 \,|\, s]_{l_1}} \qquad \mathcal{D}'''}{\Gamma; \Delta_1, \ldots, \Delta_n \vdash \prod_{i \in I}[Q_i \,\triangleright\, R_i \,|\, \psi_i \,|\, s]_{l_i}}\,[\text{TPar}]$$

and $\mathcal{D}'''$ is simply an iteration of the above. We obtain $\Delta_1 \rightarrow^{\psi\,|\,s} \Delta_1', s : T'$ by a similar consideration as above with Lemma 3. Other premises are straightforward.

case [RRes] We first consider the case when the restricted name is a session channel. Thus, we have

$$\frac{\Gamma; \Delta, \breve{s} : T, s : \overline{T}, \ldots, s : \overline{T} \vdash N \quad s \# \Gamma, \Delta}{\Gamma; \Delta \vdash (\nu\, s)N}[\text{TSRes}]$$

By induction hypothesis, we have that for any Δ'' there is $\Delta''' \subseteq \Delta''$ such that $\Gamma; \Delta''' \vdash N'$. In particular, $\Delta'' = \Delta, \breve{s} : T, s : \overline{T}, \ldots, s : \overline{T}$. Thus,

$$\frac{\Gamma; \Delta''' \vdash N'}{\Gamma; \Delta \vdash (\nu s)N'}[\text{TSRes}]$$

The case when the restricted name is a shared channel, follows immediately from the induction hypothesis by using the [TCRes] rule.

case [RPar] We have the tree

$$\frac{\Gamma; \Delta_1 \vdash N_1 \quad \Gamma; \Delta_2 \vdash N_2}{\Gamma; \Delta_1, \Delta_2 \vdash N_1 \,|\, N_2}[\text{TPar}]$$

By induction hypothesis, for any Γ, we have $\Gamma; \Delta_1 \vdash N_1$ and $N_1 \rightarrow_G N_1'$, and, for some Δ_1', s.t. $\Delta_1' \subseteq \Delta_1$ and $\Gamma; \Delta_1' \vdash N_1'$. Hence, we derive, as required, the following:

$$\frac{\Gamma; \Delta_1' \vdash N_1' \quad \Gamma; \Delta_2 \vdash N_2}{\Gamma; \Delta_1', \Delta_2 \vdash N_1' \,|\, N_2}[\text{TPar}]$$

since $\Delta_1', \Delta_2 \subseteq \Delta_1, \Delta_2$.

case [RRecover] We have

$$\frac{\Delta \rightarrow^{\psi} \Delta' \quad \Gamma; \Delta' \vdash s_?(x).P \quad \Gamma; \varepsilon \vdash R}{\Gamma; \Delta \vdash [s_?(x).P \triangleright R \,|\, \psi]_l}[\text{TNode}]$$

Note $\varepsilon \subseteq \Delta$, thus

$$\frac{\varepsilon \rightarrow^{\psi} \varepsilon \quad \Gamma; \varepsilon \vdash R \quad \Gamma; \varepsilon \vdash R}{\Gamma; \varepsilon \vdash [R \triangleright R \,|\, \varepsilon]_l}[\text{TNode}]$$

case [RRecoverSel] We have

$$\frac{\Delta \rightarrow^{\psi} \Delta' \quad \Gamma; \Delta' \vdash s \,\&\, \{\ell_i : Q_i\}_{i \in I} \quad \Gamma; \varepsilon \vdash R}{[s \,\&\, \{\ell_i : Q_i\}_{i \in I} \triangleright R \,|\, \psi]_l}[\text{TNode}]$$

Note $\varepsilon \subseteq \Delta$, thus

$$\frac{\varepsilon \rightarrow^{\psi} \varepsilon \quad \Gamma; \varepsilon \vdash R \quad \Gamma; \varepsilon \vdash R}{\Gamma; \varepsilon \vdash [R \triangleright R \,|\, \varepsilon]_l}[\text{TNode}]$$

case [RTrue] and [RFalse]. Straightforward.
case [RCong] Follows from Lemma 1.
case [RInit]

$$\dfrac{\dfrac{\Delta_0 \rightarrow^{\psi} \Delta' \quad \dfrac{\Gamma, a:T; \Delta', \breve{s}:T \vdash P}{\Gamma, a:T; \Delta' \vdash a_!(s).P}[\text{TReq}] \quad \Gamma, a:T; \varepsilon \vdash R}{\Gamma, a:T; \Delta_0 \vdash [a_!(s).P \triangleright R \,|\, \psi]_l}[\text{TNode}] \quad \mathcal{D}}{\Gamma, a:T; \Delta_0, \Delta_1, \ldots, \Delta_n \vdash [a_!(s).P \triangleright R \,|\, \psi]_l \mid \prod_{i\in I}[a_?(s).Q_i \triangleright R_i \,|\, \psi_i]_{l_i}}[\text{TPar}]$$

where $\mathcal{D}$ is as follows

$$\dfrac{\dfrac{\Delta_1 \rightarrow^{\psi_1} \Delta_1' \quad \dfrac{\Gamma, a:T; \Delta_1', s:\overline{T} \vdash Q_1}{\Gamma, a:T; \Delta_1' \vdash a_?(s).Q_1}[\text{TAcc}] \quad \Gamma, a:T; \varepsilon \vdash R_1}{\Gamma, a:T; \Delta_1 \vdash [a_?(s).Q_1 \triangleright R_1 \,|\, \psi_1]_{l_1}} \quad \mathcal{D}'}{\Gamma, a:T; \Delta_1, \ldots, \Delta_n \vdash \prod_{i\in I}[a_?(s).Q_i \triangleright R_i \,|\, \psi_i]_{l_i}}[\text{TPar}]$$

Note that $s \# \Delta_0, \Delta_1, \ldots, \Delta_n$ and $s \# \psi, \psi_i$. In the above we chose s accordingly. Thus, we can derive:

$$\dfrac{\dfrac{\dfrac{\Delta_0, \breve{s}:T \rightarrow^{\psi} \Delta', \breve{s}:T \quad \Gamma, a:T; \Delta', \breve{s}:T \vdash P \quad \Gamma, a:T; \varepsilon \vdash R}{\Gamma, a:T; \Delta_0, \breve{s}:T \vdash [P \triangleright R \,|\, \psi]_l}[\text{TNode}] \quad \mathcal{D}''}{\Gamma, a:T; \Delta_0, \breve{s}:T, \Delta_1, s:\overline{T}, \ldots, \Delta_n, s:\overline{T} \vdash [P \triangleright R \,|\, \psi]_l \mid \prod_{i\in I}[Q_i \triangleright R_i \,|\, \psi_i]_{l_i}}}{\Gamma, a:T; \Delta_0, \Delta_1, \ldots, \Delta_n \vdash (\nu s)([P \triangleright R \,|\, \psi]_l \mid \prod_{i\in I}[Q_i \triangleright R_i \,|\, \psi_i]_{l_i})}$$

The rest of the tree $\mathcal{D}''$ is derived analogously.
case [RSel] We have the following tree, for $j \in J$:

$$\dfrac{\dfrac{\Delta_0 \rightarrow^{\psi} \Delta_0', \breve{s}:\oplus\{\ell_j : T_j\}_{j\in J} \quad \dfrac{\Gamma; \Delta_0', \breve{s}:T_j \vdash P}{\Gamma; \Delta_0', \breve{s}:\oplus\{\ell_j : T_j\}_{j\in J} \vdash \breve{s} \oplus \ell.P}[\text{TSel}] \quad \Gamma; \varepsilon \vdash R}{\Gamma; \Delta_0 \vdash [\breve{s} \oplus \ell.P \triangleright R \,|\, \psi]_l} \quad \mathcal{D}}{\Gamma; \Delta_0, \Delta_1, \ldots, \Delta_n \vdash [\breve{s} \oplus \ell.P \triangleright R \,|\, \psi]_l \mid \prod_{i\in I}[s \,\&\, B_i \triangleright R_i \,|\, \psi_i]_{l_i}}$$

where $\mathcal{D}$ is as follows where $J_1 = \{1, \ldots, n\}$ for some n.

$$\dfrac{\dfrac{\mathcal{D}'' \quad \dfrac{\Gamma; \Delta_1', s:T_1 \vdash Q_1 \quad \cdots \quad \Gamma; \Delta_1', s:T_n \vdash Q_n}{\Gamma; \Delta_1', s:\&\{\ell_j : T_j\}_{j\in J_1} \vdash s \,\&\, \{\ell_j : Q_j\}_{j\in J_1}}[\text{TBr}] \quad \Gamma; \varepsilon \vdash R_1}{\Gamma; \Delta_1 \vdash [s \,\&\, B_1 \triangleright R_1 \,|\, \psi_1]_{l_1}} \quad \mathcal{D}'}{\Gamma; \Delta_1, \ldots, \Delta_n \vdash \prod_{i\in I}[s \,\&\, B_i \triangleright R_i \,|\, \psi_i]_{l_i}}$$

where $\mathcal{D}''$ is the following

$$\Delta_1 \rightarrow^{\psi_1} \Delta_1', s:\&\{\ell_j : T_j\}_{j\in J_1}$$

We iterate [TPar], [TNode] and [TBr] to obtain the rest of assumptions in $\mathcal{D}'$, namely, for all $i \in I$, that $\Delta_i \rightarrow^{\psi_i} \Delta_i', s:\&\{\ell_j : T_j\}_{j\in J_i}$, and $\Gamma; \Delta_j', s : T_j \vdash Q_j$, for all $j \in J_i$, and $\Gamma; \varepsilon \vdash R_i$.

By Lemma 3, we have $s \# \Delta_0'$, thus

$$\dfrac{\dfrac{\Delta_0 \rightarrow^{\psi \,|\, s} \Delta_0', \breve{s} : T_j \quad \Delta_0', \breve{s} : T_j \vdash P \quad \Gamma; \varepsilon \vdash R}{\Gamma; \Delta_0 \vdash [P \triangleright R \,|\, \psi \,|\, s]_l} \quad \mathcal{D}''}{\Gamma; \Delta_0, \Delta_1, \ldots, \Delta_n \vdash [P \triangleright R \,|\, \psi \,|\, s]_l \,|\, \prod_{i \in I} [Q_i \triangleright R_i \,|\, \psi_i \,|\, s]_{l_i}} \text{[TPAR]}$$

where $(\ell : Q_i) \in B_i$; we use the same idea to build the rest of the tree $\mathcal{D}''$. Let us demonstrate this for the second parallel compotnent Q_1:

$$\dfrac{\Delta_1 \rightarrow^{\psi_1 \,|\, s} \Delta_1', s : T_k \quad \Gamma; \Delta_1', s : T_k \vdash Q_1 \quad \Gamma; \varepsilon \vdash R_1}{\Gamma; \Delta_1 \vdash [Q_1 \triangleright R_1 \,|\, \psi_1 \,|\, s]_{l_1}} \text{[TNODE]}$$

where $(\ell : T_k) \in \{\ell_j : T_j\}_{j \in J_1}$ and $(\ell : Q_1) \in \{\ell_j : Q_j\}_{j \in J_1}$. The other cases are similar.

Paper IV

The Psi-Calculi Workbench: A Generic Tool for Applied Process Calculi

JOHANNES BORGSTRÖM, RAMŪNAS GUTKOVAS, IOANA RODHE,
and BJÖRN VICTOR, Uppsala University

Psi-calculi is a parametric framework for extensions of the pi-calculus with arbitrary data and logic. All instances of the framework inherit machine-checked proofs of the metatheory such as compositionality and bisimulation congruence. We present a generic analysis tool for psi-calculus instances, enabling symbolic execution and (bi)simulation checking for both unicast and broadcast communication. The tool also provides a library for implementing new psi-calculus instances. We provide examples from traditional communication protocols and wireless sensor networks. We also describe the theoretical foundations of the tool, including an improved symbolic operational semantics, with additional support for scoped broadcast communication.

Categories and Subject Descriptors: C.2.2 [**Computer-Communication Networks**]: Network Protocols—*Protocol verification*; D.2.2 [**Software Engineering**]: Design Tools and Techniques; I.1.4 [**Symbolic and Algebraic Manipulation**]: Applications

General Terms: Design, Theory, Verification

Additional Key Words and Phrases: Wireless sensor networks, process calculi, symbolic semantics

ACM Reference Format:
Johannes Borgström, Ramūnas Gutkovas, Ioana Rodhe, and Björn Victor. 2015. The psi-calculi workbench: A generic tool for applied process calculi. ACM Trans. Embedd. Comput. Syst. 14, 1, Article 9 (January 2015), 25 pages.
DOI: http://dx.doi.org/10.1145/2682570

1. INTRODUCTION

The development of concurrent systems is greatly helped by the use of precise and formal models of the system. There are many different formalisms for concurrent systems, often in specialized versions for particular application areas. For each formalism, tool support is necessary for constructing and reasoning about models of nontrivial systems. This article describes such tool support for a generic semantic framework for process calculi with mobility. Thus, instead of developing a separate tool for each separate process calculus, we develop one single generic tool for a whole family of process calculi.

Psi-calculi [Bengtson et al. 2011] is a parametric semantic framework based on the pi-calculus [Milner et al. 1992a], adding the possibility to tailor the data language and logic for each application. The framework provides a variety of features, such as lexically scoped local names for resources, communication channels as data, both unicast and broadcast communication [Borgström et al. 2011], and both first- and higher-order communication [Parrow et al. 2013].

This work has been supported by the ProFun project.
Authors' address: Uppsala University, Dept. of IT, Box 337, 751 05 Uppsala, Sweden; email: {johannes.borgstrom, ramunas.gutkovas, bjorn.victor}@it.uu.se, ioana.rodhe@foi.se.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2015 ACM 1539-9087/2015/01-ART9 $15.00
DOI: http://dx.doi.org/10.1145/2682570

Many of the different extensions of the pi-calculus, including the spi-calculus [Abadi and Gordon 1997], the fusion calculus [Wischik and Gardner 2005], the concurrent constraint pi-calculus [Buscemi and Montanari 2007], and the polyadic synchronization pi-calculus [Carbone and Maffeis 2003], can be directly represented as instances of the psi-calculi framework. A major advantage is that all meta-theoretical results, including algebraic laws and congruence properties of bisimilarity, apply to any valid instantiation of the framework. Additionally, most of these results have been proved with certainty, using the Nominal Isabelle theorem prover [Urban and Tasson 2005]. These features of psi-calculi save a lot of effort for anyone using it—psi-calculi is a reusable framework.

This article describes the Psi-Calculi Workbench (PWB), a generic tool for implementing psi-calculus instances and for analyzing processes in the resulting instances. While there are several other tools, specialized for particular process calculi and particular application areas, our tool is generic and reusable. It has a wider scope than previous works and also allows experimentation with new process calculi with a relatively low effort. Like psi-calculi, our tool is parametric: it provides functionality for bisimulation equivalence checking and symbolic simulation (or execution) of processes in any psi instance and a base library for implementing new psi-calculi instances. PWB thus has two types of users: the user analyzing systems in an existing instance of the framework and the instance implementor.

We illustrate both uses of the tool in three steps: in Section 2, we introduce the framework of psi-calculi semiformally, relating an instance corresponding to the pi-calculus and showing symbolic simulation of agents. After describing the design of PWB and how to implement an instance in Section 3, we show how to add data and computation in Section 4 by modeling the traditional alternating bit protocol for reliable communication. In Section 5, we model a data aggregation protocol for wireless sensor networks, incorporating specialized data structures and logics and both unicast and broadcast communication. Section 6 extends the previous example with a dynamic topology.

In Section 7, we describe the symbolic semantics implemented in PWB. The symbolic operational semantics of Section 7.1 simplifies previous symbolic semantics for psi-calculi [Johansson et al. 2012] and adds rules for wireless (synchronous and unreliable) broadcast [Borgström et al. 2011]. To our knowledge, this is the first symbolic semantics for lexically scoped broadcast communication.

In Section 8, we discuss related work. An abridged version of this article was published as Borgström et al. [2013].

2. INTRODUCING PSI-CALCULI

In this section, we introduce the psi-calculi parametric semantic framework semiformally and defer some precise definitions and the operational semantics to Section 7. For a more extensive treatment of psi-calculi, including motivations of the requisites and examples of other instances, see Bengtson et al. [2011], Borgström et al. [2011], and Johansson et al. [2010, 2012]. We show more complex examples in Sections 4, 5, and 6.

A psi-calculus instance is specified by three data types: the (data) terms $\mathbf{T}$, ranged over by M, N; the conditions $\mathbf{C}$, ranged over by φ; and the assertions $\mathbf{A}$, ranged over by Ψ. The terms, conditions, and assertions can be any sets where the elements may contain names (from the set $\mathcal{N}$ of names) and name permutations are admitted (so-called *nominal sets* [Pitts 2003]). In particular, every element X has a finite set of free names $\mathsf{n}(X) \subseteq \mathcal{N}$, and we write $a\#X$ for $a \notin \mathsf{n}(X)$.

Terms are used both as communication channels and for the data sent and received in communication. They can be structured, and so permit standard constructs as lists and sets, numbers, and Booleans, as well as more advanced structures. Assertions are used to model "facts" about terms and relations between them, for instance, by giving values to variables or by constraining their values. The minimal assertion is the unit, written **1**, and assertions are composed by the $\otimes$ operator. Conditions are used to perform tests on terms. Their outcome depends on the current assertion environment, through an entailment relation (Ψ entails φ, written $\Psi \vdash \varphi$), which is also part of the psi instance specification.

In the **Pi** instance, corresponding to the polyadic pi-calculus, terms are simply names $a, b, c \ldots$ and the conditions are equality tests on names. (Name equality is used in the match construct $[a = b]P$, which behaves as P if $a = b$ holds.) In the pi-calculus, there are no assertions, but the psi-calculi framework requires at least the trivial unit assertion. Later examples will show how assertions can be exploited for modeling advanced features.

Given the psi-calculus parameters **T**, **C**, **A**, the *agents*, ranged over by P, Q, ..., are of the following forms:

$\overline{M}\,\tilde{N}\,.\,P$	Output prefix
$\underline{M}(\tilde{x})\,.\,P$	Input prefix
$\overline{\overline{M}}!\tilde{N}\,.\,P$	Broadcast output prefix
$\underline{\underline{M}}?(\tilde{x})\,.\,P$	Broadcast input prefix
$\mathbf{case}\ \varphi_1 : P_1\ [] \cdots []\ \varphi_n : P_n$	Case
$(\nu a)P$	Restriction
$P \mid Q$	Parallel
$!P$	Replication
$(\!\|\Psi\|\!)$	Assertion
$A\langle\widetilde{M}\rangle$	Invocation

We write $\widetilde{M}$ for the tuple $M_1, \ldots M_n$. The output and input prefixes denote polyadic (unicast) output and input, while the broadcast prefixes denote (synchronous) broadcast output and input, which is unreliable (as in wireless systems) in the sense that transmissions might not be received. The case construct can act as any P_i such that the corresponding condition φ_i is true; the other cases are discarded. Restriction binds a in P and input prefixes bind $\tilde{x}$ in the suffix; we identify alpha-equivalent agents. The Invocation form invokes a process A, defined by the form $A(\tilde{y}) \Leftarrow P$; the behavior is that of $P\{\widetilde{M}/\tilde{y}\}$.

In the **Pi** instance, the output and input prefixes are the usual $\overline{a}\,\tilde{x}\,.\,P$ and $\underline{a}(\tilde{x})\,.\,P$; the match construct $[a = b]P$ corresponds to $\mathbf{case}\ a = b : P$. If we have a condition **true** that is always true, we can model nondeterministic choice (traditionally written $P + Q$) as $\mathbf{case\ true} : P\ []\ \mathbf{true} : Q$.

The semantics for psi-calculi is defined by a labeled transition relation written $\Psi \rhd P \xrightarrow{\alpha} P'$, meaning that in environment Ψ agent P can do an action α to become P'. In the pi-calculus instance, the environment Ψ is always the trivial **1**, but in general, it represents the assertions of the environment, including parallel agents.

The semantics is defined only for well-formed agents. An occurrence of a subterm in an agent is *guarded* if it is a proper subterm of a prefix form. An agent is *well formed* if in $\underline{M}(\widetilde{x}).P$ and $\underline{\underline{M}}?(\widetilde{x}).P$ it holds that $\widetilde{x}$ is a sequence without duplicates, that in $\mathbf{case}\ \varphi_1 : P_1\ [] \cdots []\ \varphi_n : P_n$ the agents P_i have no unguarded assertions, and that in a replication $!P$ the agent P has no unguarded assertions or broadcast input prefixes. For process definitions, a similar requirement as for replication applies.

The actions are input $\underline{M}\ (\tilde{x})$, denoting the reception of data bound to $\tilde{x}$ over the channel denoted by M, and output $\overline{M}\ (\nu\tilde{x})\tilde{N}$, denoting the sending of $\tilde{N}$ over M and additionally opening the scopes of the names $\tilde{x}$, the corresponding broadcast actions $\underline{M}?(\tilde{x})$ and $\overline{M}!\,(\nu\tilde{x})\tilde{N}$, and the silent action τ, which is the result of communication between an input and an output. When $\widetilde{x}$ is empty, we often omit $(\nu\widetilde{x})$ and $(\widetilde{x})$.

The connectivity predicates used for communication are also defined by the instantiation. The conditions include the *channel equivalence* predicate $M \dot{\leftrightarrow} N$, which is used to define which terms denote the same unicast channel, and the broadcast connectivity predicates $M \dot{\prec} K$ and $K \dot{\succ} M$ for sending and receiving on broadcast channels: a term M can be used to send a broadcast message on the channel K only if $M \dot{\prec} K$ in the current assertion environment, and similarly for broadcast reception (see Section 5 for an example).

As an example, the **Pi** agent

$$\overline{b}\,c.Q \mid \underline{b}(a).\mathbf{case}\ a = b : \underline{a}(z).R$$

has transitions labeled $\overline{b}c$ and $\underline{b}\ (x)$ for all names x and τ. The input prefix can generate infinitely many input actions (here one for each x). To avoid this infinite branching, we use a *symbolic* semantics in the tool (see Section 7.1), where the actual values are abstracted by variables. Instead, each transition has a *transition constraint*, which must be satisfied for the corresponding nonsymbolic transitions to be possible. Formally, these transitions are written $P \xrightarrow[C]{\alpha} P'$, where C is a transition constraint.

The input transitions of the previous agent can be represented by a single transition in the symbolic semantics. For simplicity, we show the first two transitions of the input prefix subagent:

$$P = \underline{b}(a)\,.\,\mathbf{case}\ a = b : \underline{a}(z).R \xrightarrow[\{\!|\mathbf{1} \vdash b \dot{\leftrightarrow} w|\!\}]{\underline{w}\ (a)} \mathbf{case}\ a = b : \underline{a}(z).R \xrightarrow[\{\!|\mathbf{1} \vdash a \dot{\leftrightarrow} v|\!\} \wedge \{\!|\mathbf{1} \vdash a = b|\!\}]{\underline{v}\ (z)} R,$$

where w and v are fresh (see Section 7 for the formal semantics). The constraint of the first transition intuitively says that the channel w is equivalent to b (there may not always be such a w!); for the second transition, a similar constraint appears in addition to the condition of the case construct.

We can use the PWB to simulate the transitions of P. The tool uses an ASCII representation of agents, where nonalphanumeric terms and conditions must be in double quotes, ν is written new, output objects are written between angular brackets, and the overline in outputs is written by a preceding single quote. For example, $\overline{b}\,\mathrm{f}(a, c)\,.\,(\nu x)Q$ is written 'b< "f(a,c)">.(new x)Q.

The first transition of agent P is as follows:

```
--|gna(a)|-->

  Source:
    b(a). case "a = b" : a(x). R<>
  Constraint:
    {| "b = gna" |}
  Solution:
    ([gna := b], 1)
  Derivative:
    case "a = b" : a(x). R<>
```

When printing the constraint, the trivial $\mathbf{1} \vdash$ is elided. The "gna" here represents a fresh name, corresponding to w: the subject of the symbolic input action.

The derivative **case** "a = b" : a(x).R does not have a nonsymbolic transition since a is not the same name as b, but the symbolic semantics does have a transition *under the constraint* that a = b:

```
--|gnb(x)|-->

   Source:
     case "a = b" : a(x). R<>
   Constraint:
     {| "a = gnb" |} ∧  {| "a = b" |}
   Solution:
     ([b := a, gnb := a], 1)
   Derivative:
     R<>
```

The constraint {|"a = b"|} can be solved by substituting a for b, as stated by the Solution line. The solution is generated by a constraint solver module in the PWB, which for the pi-calculus instance performs name unification (see Section 2), similar to earlier tools for pi-related calculi (e.g., MWB). After applying the solution to the agent, there is a corresponding nonsymbolic transition.

In addition to symbolic execution, the PWB also includes a symbolic checker that computes a minimal sufficient constraint for one agent to be (bi)similar to another, plus a witnessing relation. The two agents are nonsymbolically related after applying a solution to the constraint (if there is one).

3. IMPLEMENTATION

The Psi-Calculi Workbench (PWB) is implemented in the Standard ML programming language and compiles under the Poly/ML compiler [PolyML 2013] version 5.4. PWB is open source and freely available online from Gutkovas and Borgström [2013].

PWB is a modular implementation of psi-calculi and can be viewed both as a modeling tool and as a library for building tools for particular instances of psi-calculi. Used as a modeling tool, the user interacts with a command interpreter that provides commands for process definitions (manually or from files), manipulation of the process environment, stepping through symbolic (strong and weak) transitions of a process, and symbolic bisimilarity checking (strong and weak). Examples of such use are given in Sections 4 and 5. Next we describe the implementation of PWB and the modules that need to be provided when creating an instance of psi-calculi.

3.1. Psi-Calculus Instantiation

PWB implements a number of helper libraries for the instance implementor. We show the architecture of PWB in Figure 1. In this figure, dependencies between components go from right to left: each component may depend only on components that are above it or to its left. All components build on the supporting library that provides the basic data structures and core algorithms for psi-calculi. The instance implementor provides definitions for the parameters of an instance, constraint solvers, and parsing and pretty-printing code. These user-implemented components are then called by the different algorithms implemented by the tool and by the command interpreter. Not all components are required to be implemented: for instance, the bisimulation constraint solver is only needed for bisimilarity checking.

The parameters of an instance consist of the types name, term, condition, and assertion, and three classes of functions: those defining the logics, the substitutions, and the connectivity. As an example of the types, here are the declarations for the pi-calculus

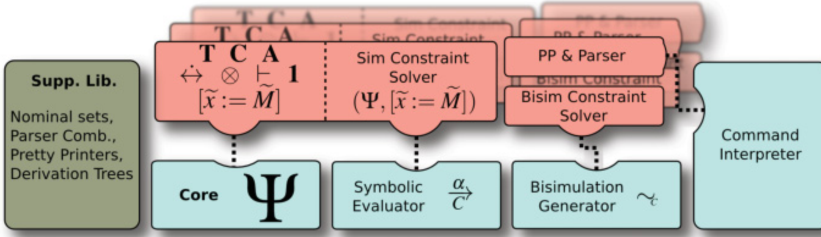


Fig. 1. Psi-Calculi Workbench architecture.

instance mentioned in Section 2. All SML code presented is written by the instance implementor.

```
type term            = name
datatype condition = Eq of term * term | T
datatype assertion = Unit
```

We need three functions to define the logic of the instance: entailment (entails, or $\vdash$), which describes which conditions are true given an assertion; a composition operator (compose, or $\otimes$), which composes two assertions; and a unit assertion (unit, or 1). We require that assertion composition form a commutative monoid (modulo entailment) and that all functions are equivariant, meaning that they treat all names equally. The bisimulation algorithm and the weak symbolic semantics also require weakening to hold, meaning that $\Psi \vdash \varphi$ implies $\Psi \otimes \Psi' \vdash \varphi$ for all Ψ'.

```
val entails : assertion * condition -> bool
val compose : assertion * assertion -> assertion
val unit    : assertion
```

We also need equivariant substitution functions, substituting terms for names in each of term, condition, and assertion.

```
type subst = (name * term) list
val substT : subst -> term      -> term
val substC : subst -> condition -> condition
val substA : subst -> assertion -> assertion
```

Finally, we have three equivariant functions that describe the connectivity of the calculus: chaneq (for unicast connectivity), brTransmit, and brReceive (for broadcast). Typically, these functions are simple injections into the condition's type (e.g., **fun** chaneq (M,N) = ChanEq (M,N), where ChanEq is a data constructor of condition), leaving the definition of connectivity to either the entailment relation or the constraint solver.

Channel equivalence chaneq is required to be commutative and transitive (for every Ψ). brTransmit is broadcast output connectivity $\dot{\prec}$ and brReceive is broadcast input connectivity $\dot{\succ}$; these functions are exemplified in Section 5. If Ψ entails $M \dot{\prec} K$ or Ψ entails $K \dot{\succ} M$, then we require all names that occur in K to also occur in M.

```
val chaneq     : term * term -> condition
val brTransmit : term * term -> condition
val brReceive  : term * term -> condition
```

All of the previous functions are further required to commute with substitution, in the sense that $f(X\sigma) = f(X)\sigma$.

The user also needs to implement parsers for each of the data types that are called by the parser for process terms.

3.2. Symbolic Execution

PWB provides symbolic execution of processes by the sstep command. This is a useful tool to explore the properties of a process, or indeed the model itself. Here values input by the process are represented by variables, and constraints are collected along the derivation of a transition. The constraints show under which conditions transitions are possible, deferring instantiation of variables as long as possible. Both strong and weak (ignoring τ-transitions) symbolic semantics are available (presented in Section 7).

In psi-calculi, parallel contexts that contain an assertion, such as $(\!|x = 3|\!)$, can enable additional transitions. Therefore, a solution (σ, Ψ) to a constraint consists of a substitution σ (representing earlier inputs) and an assertion Ψ (representing the parallel context). Intuitively, every solution (σ, Ψ) solves true, there is no solution to false, every solution to both C and C' is a solution to $C \wedge C'$, and the solutions to $(\nu\tilde{a})\{\!|\Psi' \vdash \varphi|\!\}$ are the pairs (σ, Ψ) where $\Psi \otimes \Psi'\sigma \vdash \varphi\sigma$ and the names in $\tilde{a}$ do not occur in Ψ, σ.

The instance implementor may provide a constraint solver for the transition constraints. The solver should return either a string describing the unsatisfiability of a constraint or a solution consisting of a substitution and an assertion. Since transition constraints are simply a conjunction of atomic constraints, a simple unification-based solver often suffices. The type of the solver is the following:

```
val solve : constraint -> (string, (name * term) list * assertion) either
```

As an example, the solver for the pi-calculus instance of Section 2 performs unification, implemented by the transition relation next. The nodes in the transition system are either a pair (C, σ) or the failed state $\varnothing$:

$$
\begin{array}{rcl}
(\nu\tilde{a})\{\!|\mathbf{1} \vdash \mathrm{T}|\!\} \wedge C, \sigma & \rightarrow & C, \sigma \\
(\nu\tilde{a})\{\!|\mathbf{1} \vdash a = a|\!\} \wedge C, \sigma & \rightarrow & C, \sigma \\
(\nu\tilde{a})\{\!|\mathbf{1} \vdash a = b|\!\} \wedge C, \sigma & \rightarrow & \varnothing \quad \text{if } a \neq b \wedge (a \in \tilde{a} \vee b \in \tilde{a}) \\
(\nu\tilde{a})\{\!|\mathbf{1} \vdash a = b|\!\} \wedge C, \sigma & \rightarrow & C[b := a], \sigma[b := a] \quad \text{otherwise.}
\end{array}
$$

3.3. Symbolic (Bi)simulation

PWB can also be used to check simulation relations on processes. As an example, the command P~ Q attempts to construct a bisimulation relation relating agents P and Q. To this end, we implement a symbolic bisimulation algorithm based on Johansson et al. [2012] (with some corrections and optimizations). This algorithm takes two processes and yields a constraint in an extended constraint language; the two processes are bisimilar under all solutions to the constraint. A simple variation of the algorithm is used for simulation checking.

The language for bisimulation constraints additionally includes conjunction, disjunction, and implication, as well as constraints for term equality $\{\!|M = N|\!\}$, freshness $\{\!|a\#X|\!\}$ (with the intuition "a is not free in X"), and static implication. In order to simplify the development of a constraint solver for this richer language, PWB contains an SMT solver library with suitable helper functions. Unless the assertion language is trivial (only the unit assertion), most of the additional effort in extending a solver for transition constraints to one for bisimulation constraints lies in properly treating static implication constraints.

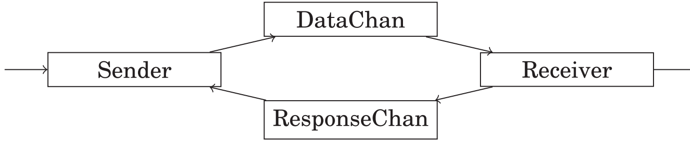


Fig. 2. Alternating Bit Protocol scheme.

4. THE ALTERNATING BIT PROTOCOL

In this section, we describe the modeling in PWB of the classical Alternating Bit Protocol. This demonstrates the use of PWB to define a tailor-made process calculus for a particular problem or problem domain. We also give an example of symbolic weak transition generation in PWB.

4.1. Introduction to the Alternating Bit Protocol

The Alternating Bit Protocol (ABP) [Bartlett et al. 1969] is a simple network protocol for reliable data transmission through lossy channels. Reliable here means that all data fragments are received exactly once and in the right order at the receiver. Consider a sender Sender, a receiver Receiver, and two communication channels between them: DataChan, over which data fragments are sent, and ResponseChan, over which acknowledgments are sent. We show this situation in Figure 2: the arrows denote the direction of the data being transmitted. ABP assumes reliable error detection, but no error correction.

To ensure that Receiver receives every fragment despite lossy communication channels, Sender repeatedly sends the same fragment until it receives a corresponding acknowledgment, at which point the sender starts transmitting the next fragment. Since the receiver should not accept the same fragment twice, a protocol is needed for distinguishing between packets. In ABP, each data packet has a one-bit flag attached to it. The flag 0 is attached to the first packet sent; the acknowledgment of the receiver for this packet will also have flag bit 0. When Sender receives an acknowledgment with flag 0, it knows that Receiver has correctly received the fragment, and Sender will then start sending the next packet with flag bit 1, and so on. Thus, sequences of sent or received packages and respective acknowledgments with the same flag bit all refer to the same data fragment.

4.2. A Psi-Calculus Instance For ABP

To define a psi-calculus instance where ABP can be expressed, we start with the data terms. Since the behavior of the protocol does not depend on the data being transmitted, we simply represent each fragment as a name. However, the protocol itself needs some data values and structures.

In the set of terms, we include the channels DataChan and ResponseChan and the value ERR to signify that an error has been detected. We also have 0 and 1 bits and a negation operation $\sim\cdot$ on them with the expected equalities $\sim 0 = 1$ and $\sim 1 = 0$.

Our account of ABP is untyped, so these term constructors yield terms that are not intended to be part of the model, such as ~ERR. Such spurious terms yield the invalid value $\bot$. In summary, we define the data terms $\mathbf{T}$ as follows:

<table>
<tr><th>Notation</th><th>SML</th><th>PWB</th></tr>
<tr><td>$Val \triangleq \{\mathrm{ERR}, 0, 1\}$
$\mathbf{T} \triangleq Val \cup \{\bot\} \cup \mathcal{N}$
$\cup \{\sim M : M \in \mathbf{T}\}$</td><td>datatype term
= Error | Zero | One | Bottom
| Name of name | Neg of term</td><td>M ::= ERR | 0 | 1 | _|_
| Name | ~M</td></tr>
</table>

Here and in subsequent displays, the column *Notation* is the mathematical notation, *SML* is the code written by the instance implementor, and PWB is the ASCII syntax used in the tool by the user of the instance.

Next we define the conditions. In the protocol, we need to compare the sender's or receiver's bit with a transmitted bit and to see whether an error occurred while transmitting data. To do this, we use equality $=$ on values.

We add a condition True that always holds and a False condition that never holds. Lastly, we include a channel equivalence condition for unicast communication (ABP does not use broadcast, so we let the broadcast connectivity predicates yield False):

Notation	*SML*	PWB
$\mathbf{C} \triangleq \{\text{True}, \text{False}\}$ $\cup\ \{M = N : M, N \in \mathbf{T}\}$ $\cup\ \{M \dot{\leftrightarrow} N : M, N \in \mathbf{T}\}$	`datatype condition` `= CTrue	CFalse` `| Equal of term * term` `| ChEq of term * term`	φ ::= True | False | M = M | M <-> M

We do not need assertions to model the ABP, so we let $\mathbf{A} = \{\text{Unit}\}$ as in Section 2.

As the last step, we define the substitution functions on terms and conditions. They are standard capture avoiding substitutions, followed by normalization with respect to a term rewriting system given later. We use rewriting after substitutions in order to accurately detect loops of τ-transitions when computing weak transitions. This also significantly simplifies the constraint solver, since the normal forms are simpler to handle than arbitrary terms.

To follow, we give the rewrite system for terms for reduction context $\mathcal{R} ::= []|{\sim}\mathcal{R}$. It evaluates the ${\sim}\cdot$ operator, cancels out double negation of variables, and identifies the spurious terms. In particular, the term ${\sim}{\sim}$ERR is spurious and is rewritten to $\bot$:

$$
\begin{array}{rcl} {\sim}\text{ERR} & \to & \bot \\ {\sim}\bot & \to & \bot \end{array}
\qquad
\begin{array}{rcl} {\sim}0 & \to & 1 \\ {\sim}1 & \to & 0 \end{array}
\qquad
{\sim}{\sim}x \ \to\ x \ \text{ if } x \in \mathcal{N}.
$$

The following is the term-rewriting system for the conditions. Equalities involving spurious terms $\bot$ are rewritten to False. Note that we only consider equality conditions where the constituent terms are already in normal form; this suffices since the substitution function on conditions is defined in terms of substitution function on terms:

$$
\begin{array}{rcl} {\sim}x = {\sim}y & \to & x = y \\ {\sim}x = x & \to & \text{False} \\ x = {\sim}x & \to & \text{False} \end{array}
\qquad
\begin{array}{rcl} M = N & \to & \text{True if } M = N \text{ and } \{M, N\} \subseteq \mathit{Val} \cup \mathcal{N} \\ M = N & \to & \text{False if } M \neq N \text{ and } \{M, N\} \subseteq \mathit{Val} \\ M = N & \to & \text{False if } \bot \in \{M, N\}. \end{array}
$$

Finally, we need to define entailment. For conditions in normal form, we define

$$
\text{Unit} \vdash a \dot{\leftrightarrow} b \text{ iff } a = b \qquad \text{Unit} \vdash M = N \text{ iff } M = N \qquad \text{Unit} \vdash \text{True},
$$

and otherwise we let $\text{Unit} \vdash \varphi$ iff $\varphi \to^{+} \varphi' \not\to$ and $\text{Unit} \vdash \varphi'$

4.3. Constraint Solver for ABP Transition Constraints

The ABP constraint solver is a standard unification algorithm defined as a transition system. The design is greatly simplified by the fact that the conditions in the constraints are in normal form.

The following is the unification transition system. The first two rules are trivial. The rules concerning the channel equivalence $\dot{\leftrightarrow}$ condition are the classic unification on names as seen in the pi-calculus solver. The last rules concern the equality condition $=$. Because the terms are in the normal form, we know that one of the sides is a name, and thus we do elimination, or swapping in order to allow elimination. To follow, $\tilde{a}\#X$

denotes that names $\tilde{a}$ don't occur freely in X; we omit $\mathbf{1} \vdash$ in front of every condition:

$$
\begin{array}{rcll}
(\nu\tilde{a})\{\!|\mathrm{True}|\!\} \wedge C, \sigma & \to & C, \sigma & \\
(\nu\tilde{a})\{\!|\mathrm{False}|\!\} \wedge C, \sigma & \to & \varnothing & \\
(\nu\tilde{a})\{\!|a \dot{\leftrightarrow} b|\!\} \wedge C, \sigma & \to & C, \sigma & \text{if } a = b \text{ and } a, b \in \mathcal{N} \\
(\nu\tilde{a})\{\!|a \dot{\leftrightarrow} b|\!\} \wedge C, \sigma & \to & C[b := a], \sigma[b := a] & \text{if } \tilde{a}\#a, b \text{ and } a \neq b \\
(\nu\tilde{a})\{\!|a \dot{\leftrightarrow} b|\!\} \wedge C, \sigma & \to & \varnothing & \text{otherwise} \\
(\nu\tilde{a})\{\!|a = M|\!\} \wedge C, \sigma & \to & C[a := M], \sigma[a := M] & \text{if } \tilde{a}\#a, M \text{ and } a \in \mathcal{N} \\
(\nu\tilde{a})\{\!|M = N|\!\} \wedge C, \sigma & \to & (\nu\tilde{a})\{\!|N = M|\!\} \wedge C, \sigma & \text{otherwise.}
\end{array}
$$

4.4. The ABP as a Process

Here we present the process modeling the ABP in the ABP psi-calculus instance defined earlier. We give the definition in PWB syntax, which is used by the user of the psi instance.

We model the components Sender and Receiver of ABP shown in Figure 2 as psi-calculus processes. The behavior of components DataChan and ResponseChan are captured implicitly in our model. For composing the system, components have input and output channels inp and out, respectively. The Receiver and Sender each have one additional channel for output o, respectively, input i to the application that uses the protocol.

The sender is modeled as follows: first it inputs data on input channel i and then recursively outputs the data together with the current bit b on the channel out. Then the sender receives the acknowledgment bit on input channel inp: if it matches b, the sender flips b and returns to waiting for data; otherwise (if the bit did not match or an error occurred), the sender attempts to send the data and b until it receives an acknowledgment with flag b:

```
Sender(i,inp,out,b) <= i(data).SenderSend<i,inp,out,data,b>;

SenderSend(i,inp,out,data,b) <= 'out<data, b>. inp(ackBit).
     case "b = ackBit"    : Sender<i,inp,out,"~b">
       [] "b = ~ackBit"   : SenderSend<i,inp,out,data,b>
       [] "ERR = ackBit"  : SenderSend<i,inp,out,data,b> ;
```

The receiver works in a dual fashion:

```
Receiver(o, inp, out, b) <= inp(data, bit).
     case "b = bit"    : 'o<data>.'out<b>.Receiver<o,inp,out,"~b">
       [] "b = ~bit"   : 'out<"~bit"> . Receiver<o,inp,out,b>
       [] "ERR = bit"  : 'out<"~b"> . Receiver<o,inp,out,b> ;
```

An error might occur at any time on each of the channels. This kind of unreliable process is modeled implicitly by treating names (representing bits) as variables. Since transmitted names are variables, the constraint solver may enable any **case** clause in either Sender or Receiver by finding a suitable term to substitute them for.

Hiding the internal channels, the ABP system can be described as follows:

```
ABP(i,o,sb,rb) <= (new RcSn, SnRc) (
     Sender<i,RcSn,SnRc,sb> | Receiver<o,SnRc,RcSn,rb>);
```

4.5. A Sample Weak Transition

When studying the ABP, it is interesting to see when the protocol communicates with the outside system, ignoring τ-transitions. We here show such a "weak" transition,

where the sender receives data and transmits it to the receiver via the data channel. We use the wsstep command on ABP<i,o,sb,rb> to obtain the following transition, among others:

```
1  ==|gen2(data1)|==>
2    Source:
3      ABP<i, o, sb, rb>
4    Constraint:
5      (new RcSn, SnRc){| "i <-> gen2" |} ∧
6      (new RcSn, SnRc){| "SnRc <-> SnRc" |} ∧
7      (new RcSn, SnRc){| "RcSn <-> RcSn" |} ∧
8      (new RcSn, SnRc){| "rb = ~sb" |}
9    Solution:
10     ([rb := "~sb", gen2 := i], 1)
11   Derivative:
12     (new SnRc, RcSn)(
13       (case
14         False : Sender<i, RcSn, SnRc, "~sb"> []
15         True  : SenderSend<i, RcSn, SnRc, data1, sb> []
16         False : SenderSend<i, RcSn, SnRc, data1, sb>
17       ) |
18         (Receiver<o, SnRc, RcSn, rb>)
19     )
```

After the transition, the sender (lines 13–16) is in a state where it has received an acknowledgment bit that does not match its own bit (constraint on line 8), reducing the condition "b = ~ackBit" (at this state it is "sb = ~rb") of SenderSend to true (on line 15).

This transition is among the seven transitions produced by PWB. Since there is always a possibility that both sender and receiver will detect an error ERR, there are infinitely many weak transitions following a cycle between them. The occurrence of such cycles are detected (modulo alpha-equivalence) by the wsstep command. Since the terms occurring in agents are in normal form, wsstep terminates on ABP.

We have shown the development of a tailor-made psi-calculus instance in PWB. (The full code listing is available online [Gutkovas and Borgström 2013].) Doing so, we have expressed bits and bit operations directly, and we have shown that it is possible and useful to use computation in the substitution functions, which departs from traditional calculi. We have also shown the symbolic simulation of a weak transition, which is useful for applications.

5. DATA COLLECTION IN A WIRELESS SENSOR NETWORK

In this example, we study a data collection protocol for wireless sensor networks (WSNs) by modeling it in a custom psi-calculus that we implement in PWB.

A wireless sensor network consists of numerous sensor nodes that sense environmental data. A special node, called the sink, is used to collect data from the network. Collection often uses multihop communication, building a routing tree rooted at the sink [Madden et al. 2002]. As wireless communication is unreliable, different trees may be built in each protocol run.

We present a simple algorithm to build a routing tree: the sink starts the tree building by broadcasting a special init message containing its identifier *Sink*. When a node n first receives an init message, it sets its parent $parent_n$ to the sender of the message and broadcasts a new init message containing its own identifier to continue building the next level of the tree. After the building of a tree is complete, each node sends a data message containing its data to its parent. Moreover, each node forwards received data messages to its parent, ensuring that it eventually reaches the sink.

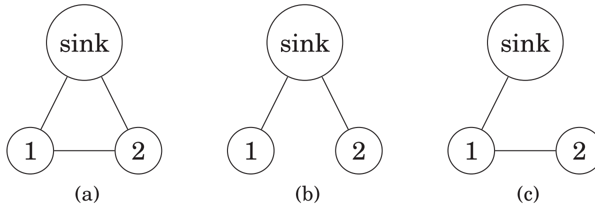


Fig. 3. A simple topology with a sink and two sensor nodes where (a) shows the connectivity and (b) and (c) show some possible routing trees.

5.1. Psi-Calculus Instance for WSN Data Collection

We first define and implement a custom Psi-calculus instance suitable for modeling the tree building and data collection protocol described earlier. We use structured channels of two kinds: broadcast channels $\mathsf{init}(M)$ and unicast channels $\mathsf{data}(M)$. The broadcast connectivity between nodes is given by an undirected topology graph. We first assume a static topology top; the topology in Figure 3(a) would be represented by top $= \{(0,1),(0,2),(1,2)\}$, where the sink has id 0. The corresponding psi-calculi parameters are defined as follows:

Notation	*SML*	PWB
$\mathbf{T} \triangleq \{\mathsf{init}(M), \mathsf{data}(N) : M, N \in \mathbf{T}\} \cup \mathcal{N} \cup \mathbb{N}$ $\mathbf{C} \triangleq \{M \dot{\prec} N, M \dot{\succ} N, M \dot{\leftrightarrow} N : M, N \in \mathbf{T}\}$ $\mathbf{A} \triangleq \{\mathsf{top}\}$ $\mathbf{1} \triangleq \mathsf{top}$	`datatype term` ` = Init of term	Data of term` `	Name of name	Int of int` `datatype condition` ` = OutputConn of term*term` `	InputConn of term*term` `	ChEq of term*term` `datatype assertion = Unit` `val unit = Unit`	M ::= init(M) | data(M) | *Name* | *N* φ ::= M < M | M > M | M <−> M Ψ ::= 1 N ::= $[0-9]^+$

Since we consider a static topology, we implement assertions as a unit type. A broadcast output prefix with subject $\mathsf{init}(i)$ can broadcast on the broadcast channel $\mathsf{init}(i)$, while an input prefix with the same subject can receive from any connected broadcast channel as given by the topology. Two unicast prefixes may communicate if and only if their subjects are the same name. Thus, we define $\vdash$ as follows:

$$\begin{array}{ll}\Psi \vdash \mathsf{init}(M) \dot{\prec} \mathsf{init}(N) & \text{iff } M = N \in \mathbb{N}\\ \Psi \vdash \mathsf{init}(M) \dot{\succ} \mathsf{init}(N) & \text{iff } M, N \in \mathbb{N} \text{ and either } (M,N) \in \Psi \text{ or } (N,M) \in \Psi\\ \Psi \vdash \mathsf{data}(a) \dot{\leftrightarrow} \mathsf{data}(b) & \text{iff } a = b \in \mathcal{N}.\end{array}$$

5.2. Constraint Solver for Symbolic Transitions

We describe the implementation of the transition constraint solver. We write $\varnothing$ for no solution. Transition constraints are conjunctions of conditions. The constraints are solved in two phases, corresponding to the unicast connectivity constraints and the broadcast connectivity constraints, respectively. To simplify the solver, we treat all free names in the processes as distinct (cf. distinctions [Milner et al. 1992b]). For unicast constraints, the solver thus fails (returning $\varnothing$) if the constraint is not satisfied:

$$\begin{array}{ll}(\nu\tilde{a})\{\!|\mathsf{data}(a) \dot{\leftrightarrow} \mathsf{data}(b)|\!\} \wedge C \;\rightarrow\; C & \text{if } a = b\\ (\nu\tilde{a})\{\!|\mathsf{data}(a) \dot{\leftrightarrow} \mathsf{data}(b)|\!\} \wedge C \;\rightarrow\; \varnothing & \text{otherwise.}\end{array}$$

The constraint solver then checks for broadcast connectivity in the given topology. Let O be the output constraints $\{\!|\mathsf{init}(n) \dot{\prec} a|\!\}$ and I the input constraints $\{\!|a \dot{\succ} \mathsf{init}(n)|\!\}$. We distinguish four different cases:

(1) If $I = \emptyset$ and $O = \{\{\!|\mathsf{init}(n) \dot{\prec} a|\!\}\}$, then the solution is $[a := \mathsf{init}(n)]$.
(2) If $I \neq \emptyset$ and $O = \{\{\!|\mathsf{init}(n) \dot{\prec} a|\!\}\}$, and we have $(n, m) \in \mathsf{top}$ for every constraint $\{\!|a \dot{\succ} \mathsf{init}(m)|\!\}$ in I, then the solution is $[a := \mathsf{init}(n)]$. Otherwise, the constraint is unsatisfiable, that is, $\varnothing$.
(3) If $I \neq \emptyset$ and $O = \emptyset$, then the constraint solver finds n such that for every $\{\!|a \dot{\succ} \mathsf{init}(m)|\!\} \in I$, we have $(n, m) \in \mathsf{top}$. For each such n, $[a := \mathsf{init}(n)]$ is a possible solution.
(4) If $I = \emptyset$ and $O = \emptyset$, then the broadcast part of the constraint is trivially true.

5.3. Tree Building Model

Once the instance is implemented, we can define processes modeling the tree building algorithm in PWB syntax. The sink broadcasts its own channel and then goes into data collection mode; that is, it listens on its unicast channel repeatedly:

```
Sink(nodeId, bsChan) <=
     '"init(nodeId)"!<bsChan> .
     ! "data(bsChan)"(x) ;
```

A node listens on its broadcast channel for a channel of a parent to which it will send data. Then, similarly to the sink, it broadcasts its own unicast channel on which it expects data to receive in order to forward it to the parent. After completing the broadcast, it sends its data to the parent and goes into mode of forwarding data:

```
Node(nodeId, nodeChan, datum) <=
     "init(nodeId)"?(pChan) .
     '"init(nodeId)"!<nodeChan> .
     '"data(pChan)"<datum> .
     NodeForwardData<nodeChan, pChan> ;

NodeForwardData(nodeChan, pChan) <=
     ! "data(nodeChan)"(x). '"data(pChan)"<x> ;
```

5.4. Example Strong Transitions

We here study the (symbolic) transition system generated by a small WSN with a sink and two sensor nodes. Each node has a unique channel for response messages:

```
System3(d1,d2) <=
     (new chanS) Sink<0,chanS>        |
     (new chan1)  Node<1, chan1, d1>  |
     (new chan2)  Node<2, chan2, d2>
```

We will show a possible transition sequence in PWB, using the topology shown in Figure 3(a). To follow, we only consider transitions labeled with broadcast output and unicast communication actions.

The following initial transition is obtained by executing the symbolic simulator of PWB on System3<d1,d2>. The resulting system is in a configuration where both sensor nodes have obtained the parent's channel, in this case the sink's. The nodes would then be able to communicate their data to the sink. The unicast channel connectivity corresponds to the routing tree shown in Figure 3(b). It is one of seven possible initial transitions produced by PWB, of which three represent broadcast reception from the environment and the other three situations where not all nodes receive the broadcast message. The transition label gna!(new bsChan)bsChan represents the channel with a fresh name gna. The generated constraint requires $\{\!|\mathrm{init}(0) \dot{\prec} \mathrm{gna}|\!\} \wedge \{\!|\mathrm{gna} \dot{\succ} \mathrm{init}(1)|\!\} \wedge \{\!|\mathrm{gna} \dot{\succ} \mathrm{init}(2)|\!\}$, meaning node 0 is output connected to some channel gna that is input

connected to nodes 1 and 2. The constraint solver finds a solution to the constraint, which substitutes init(0) for gna:

```
--|gna!(new bsChan)bsChan|-->
Source:
  System3<d1, d2>
Constraint:
  (new chan1, chan2, chanS){| "init(0)<gna" |} ∧
  (new chanS, chan2, chan1){| "gna>init(1)" |} ∧
  (new chanS, chan1, chan2){| "gna>init(2)" |}
Solution:
  ([gna := "init(0)"], 1)
Derivative:
  (!("data(chanS)"(x))) |
    (((new chan1)(
      '"init(1)"!<chan1>.
        '"data(chanS)"<d1>.
          NodeForwardData<chan1, chanS>
    )) |
      ((new chan2)(
        '"init(2)"!<chan2>.
          '"data(chanS)"<d2>.
            NodeForwardData<chan2, chanS>
      )))
```

In the derivative, the Sink successfully communicated its unicast channel chanS to both nodes.

From this point, the system can evolve in two symmetrical ways: either of the nodes broadcasts an init message, but since no node in the (closed) system is listening on a broadcast channel, the message is not received. The following transition is for node 1:

```
--|gna!(new chan1)chan1|-->
Source:
   The same as the above derivative
Constraint:
  (new chan2, chan1){| "init(1)<gna" |}
Solution:
  ([gna := "init(1)"], 1)
Derivative:
  (!("data(chanS)"(x))) |
    (('"data(chanS)"<d1>.
      NodeForwardData<chan1, chanS>) |
     ((new chan2)(
       '"init(2)"!<chan2>.
         '"data(chanS)"<d2>.
           NodeForwardData<chan2, chanS>
      )))
```

The system is now in the state where node 1 can send data to the sink. By following the analogous transition for node 2, we get the system where both nodes are ready to communicate the data:

```
--|gna!(new chan2)chan2|-->
Source:
   The same as the above derivative
Constraint:
  (new chan2){| "init(2)<gna" |}
Solution:
  ([gna := "init(2)"], 1)
Derivative:
  (!("data(chanS)"(x))) |
```

```
(('"data(chanS)"<d1>.
  NodeForwardData<chan1, chanS>) |
 ('"data(chanS)"<d2>.
   NodeForwardData<chan2, chanS>))
```

We have demonstrated the use of advanced features in PWB such as the use of structured channels with different modes of communication (point to point vs. broadcast). The broadcast connectivity graph (topology) was formalized as an assertion; this allows us to potentially extend the model, for instance, to dynamic or localized connectivity. We used the symbolic execution to simulate strong symbolic transitions of the system. All of this shows the versatility and utility of PWB for use in modeling and studying WSN algorithms.

6. DYNAMIC TOPOLOGY IN WIRELESS SENSOR NETWORK

We here extend the example of Section 5 with dynamic topology. We first allow adding edges to the connectivity graph and then add the dual operation of removing edges.

Let the parameters be as in the example in Section 5 except for the assertions, which are now a finite set of tuples representing edges in a topology:

Notation	*SML*	PWB
$\mathbf{A} \triangleq \mathcal{P}_{\text{fin}}(\mathbf{T} \times \mathbf{T})$	`datatype assertion` ` = Top of (term*term)list`	$\Psi ::= \epsilon$
$\mathbf{1} \triangleq \emptyset$	`val unit = Top []`	$\mid \ (\mathrm{M},\mathrm{N})(,(\mathrm{M},\mathrm{N}))^*$

The entailment relation is left unchanged, and the constraint solver for the unicast constraints is the same. To enable broadcast connectivity, if the necessary edge is not present, the solver simply attempts to add it to the solution (as is common in process calculi models for WSNs [Ghassemi et al. 2008; Godskesen 2010]). For example, the solution of the constraint of the first transition in Section 5.4 with an empty topology is ([gna := "init(0)"], "(0,2),(0,1)").

In the following, we add the ability for agents to also remove edges from the environment. In the assertions, we model edges as binary toggles, so if the same edge occurs twice, this is equivalent to it not appearing at all (i.e., $\{(M,N)\} \otimes \{(M,N)\} \simeq \mathbf{1}$). The parameters are extended by adding conditions corresponding to whether an edge is present or not, and the assertions are finite multisets:

Notation	*SML*	PWB

$\mathbf{C} \triangleq \cdots \cup \{\mathsf{conn}(M,N), \mathsf{disconn}(M,N) : M,N \in \mathbf{T}\}$	`datatype condition = ...`
	` | Conn of term*term`
	` | Disconn of term*term`
		$\varphi ::= \ldots \mid \mathrm{conn(M,N)} \mid \mathrm{disconn(M,N)}$

$\mathbf{A} \triangleq \mathbf{T} \times \mathbf{T} \rightarrow_{\text{fin}} \mathbb{N}$	`datatype assertion` ` = Top of (term*term)list`	$\Psi ::= \epsilon \mid (\mathrm{M},\mathrm{N})(,(\mathrm{M},\mathrm{N}))^*$
$\mathbf{1} \triangleq \emptyset$	`val unit = Top []`	

An odd number of edge tuples in the environment denotes that the edge is present; an even number denotes absence. Thus, adding a tuple to the environment might add or remove an edge. We capture this with the following entailment definition:

$$
\begin{array}{ll}
\Psi \vdash \mathsf{conn}(M,N) & \text{iff } M,N \in \mathbb{N} \text{ and } |\Psi(M,N)| + |\Psi(N,M)| \text{ is odd} \\
\Psi \vdash \mathsf{disconn}(M,N) & \text{iff } M,N \in \mathbb{N} \text{ and } \Psi \nvdash \mathsf{conn}(M,N) \\
\Psi \vdash \mathsf{init}(M) \dot{\succ} \mathsf{init}(N) & \text{iff } \mathsf{conn}(M,N).
\end{array}
$$

For the protocol in Section 5, we may reuse the same constraint solver, keeping in mind that it does not handle the case where a disconn condition guards a broadcast input. We can also express the alteration of the topology with the following two agents:

```
Connect(a,b) <=
case "conn(a,b)"    :*tau*.0
  [] "disconn(a,b)":*tau*.(|"(a,b)"|) ;
```

```
Disconnect(a,b) <=
case "conn(a,b)"    :*tau*.(|"(a,b)"|)
  [] "disconn(a,b)":*tau*.0 ;
```

The agent Disconnect<1, 2> $|(\!|$"(1,2)"$|\!)$ has two transitions: first $(\!|$"(1,2)"$|\!)|$ $(\!|$"(1,2)"$|\!)$ with trivially solvable constraint $\{\!|$"(1,2)" |– "conn(1,2)"$|\!\}$, and second 0|(|"(1,2)"|) with the solution ([ ], "(1,2)"). In both transitions, the environment was extended with an extra tuple $(1, 2)$, effectively removing an edge from the topology. Intuitively, the agents Connect and Disconnect can be used to set and unset bits in a global table.

7. SYMBOLIC SEMANTICS

In this section, we describe a symbolic operational semantics for broadcast psi-calculi that is sound (Theorem 7.11) and complete (Theorem 7.12) with respect to the concrete broadcast semantics [Borgström et al. 2011, 2013]. This semantics is the one that is implemented in the PWB, and it extends, simplifies, and corrects the original symbolic semantics [Johansson et al. 2012].

7.1. Symbolic Operational Semantics

As we have seen, transitions in the symbolic operational semantics are of the form $P \xrightarrow[C]{\alpha} Q$, where C is a constraint that needs to be satisfied for the transition to be enabled. Each PWB instance implements a solver, which computes solutions for the transition constraints of that instance.

Definition 7.1 (*Constraints and Solutions*). A *solution* is a pair (σ, Ψ) where σ is a substitution sequence of terms for names, and Ψ is an assertion. The *transition constraints*, ranged over by C, C_t, and their corresponding solutions $\mathrm{sol}(C)$ are defined by:

	Constraint	Solutions
$C, C' ::=$	**true**	$\{(\sigma, \Psi) \,:\, \sigma$ is a subst. sequence $\wedge\ \Psi \in \mathbf{A}\}$
$\mid$	**false**	$\emptyset$
$\mid$	$(\nu a)C$	$\{(\sigma, \Psi) \,:\, b\#\sigma, \Psi, C \wedge (\sigma, \Psi) \in \mathrm{sol}((a\ b) \cdot C)\}$
$\mid$	$\{\!\mid \Psi' \vdash \varphi \mid\!\}$	$\{(\sigma, \Psi) \,:\, \Psi'\sigma \otimes \Psi \vdash \varphi\sigma\}$
$\mid$	$\exists x.C$	$\{(\sigma, \Psi) \,:\, y\#\sigma, \Psi, C \wedge ([y := M]\sigma, \Psi) \in \mathrm{sol}((x\ y) \cdot C)\}$
$\mid$	$a \in \mathsf{n}(M)$	$\{(\sigma, \Psi) \,:\, a \in \mathsf{n}(M\sigma)\}$
$\mid$	$C \wedge C'$	$\mathrm{sol}(C) \cap \mathrm{sol}(C')$

$(a\ b)\cdot C$ stands for the simultaneous replacement of a for b and b for a in C ("swapping"). In $(\nu a)C$, a is binding into C; and in $\exists x.C$, x is binding into C. We write $\exists^b x.C$ for $(\nu b)\exists x.(b \in \mathsf{n}(x) \wedge C)$; the only uses of $\exists$ and $\cdot \in \mathsf{n}(\cdot)$ will be in this restricted form (which is itself only used in Rule sBrClose in Table I). We adopt the notation $(\sigma, \Psi) \models C$ to say that $(\sigma, \Psi) \in \mathrm{sol}(C)$, and write $C \leftrightarrow D$ to say that $\mathrm{sol}(C) = \mathrm{sol}(D)$.

A transition constraint C defines a set of solutions $\mathrm{sol}(C)$, namely, those where the formula becomes true by applying the substitution and adding the assertion. For example, the transition constraint $\{\!\mid \mathbf{1} \vdash x = 3 \mid\!\}$ has solutions $([x := 3], \mathbf{1})$ and $([\,],\ x = 3)$, where $[\,]$ is the identity substitution.

Restriction distributes over logical conjunction, and logical conjunction has **true** as unit and is associative. We thus consider constraints modulo the equations to follow.

LEMMA 7.2. $(\nu a)(C_1 \wedge C_2) \leftrightarrow (\nu a)C_1 \wedge (\nu a)C_2$ *and* $C_1 \wedge (C_2 \wedge C_3) \leftrightarrow (C_1 \wedge C_2) \wedge C_3$ *and* $C \wedge \mathbf{true} \leftrightarrow C$.

Table I. Symbolic Transition Rules for Broadcast Communication. A symmetric version of sBrCom is elided. In sBrOpen, the expression $\nu\tilde{a} \cup \{b\}$ means the sequence $\tilde{a}$ with b inserted anywhere.

$$\textsc{sBrOut}\ \frac{x\#\Psi, M, \widetilde{N}, P}{\overline{M}\,\widetilde{N}\,.\,P \xrightarrow[\mathbf{1}\vdash M \dot{\prec} x]{\overline{x}!\,\widetilde{N}} P} \qquad \textsc{sBrIn}\ \frac{x, \widetilde{y}\#\Psi, M, P \quad x\#\widetilde{y}}{\underline{M}(\widetilde{y})\,.\,P \xrightarrow[\mathbf{1}\vdash x \dot{\succ} M]{\underline{x}?(\widetilde{y})} P}$$

$$\textsc{sBrMerge}\ \frac{P \xrightarrow[C_1]{\underline{x}?(\widetilde{y})} P' \qquad Q \xrightarrow[C_2]{\underline{x}?(\widetilde{y})} Q'}{P \mid Q \xrightarrow[(\mathcal{F}(Q)\otimes C_1)\wedge(\mathcal{F}(P)\otimes C_2)]{\underline{x}?(\widetilde{y})} P' \mid Q'}$$

$$\textsc{sBrCom}\ \frac{P \xrightarrow[C_1]{\overline{x}!\,(\nu\widetilde{a})\widetilde{N}} P' \qquad Q \xrightarrow[C_2]{\underline{x}?(\widetilde{y})} Q'}{P \mid Q \xrightarrow[(\mathcal{F}(Q)\otimes C_1)\wedge(\mathcal{F}(P)\otimes C_2)]{\overline{x}!\,(\nu\widetilde{a})\widetilde{N}} P' \mid Q'[\widetilde{y} := \widetilde{N}]}\ \begin{array}{l}|\widetilde{y}| = |\widetilde{N}| \\ \widetilde{a}\#Q\end{array}$$

$$\textsc{sBrOpen}\ \frac{P \xrightarrow[C]{\overline{y}!\,(\nu\widetilde{a})\widetilde{N}} P'}{(\nu b)P \xrightarrow[(\nu b)C]{\overline{y}!\,(\nu\widetilde{a}\cup\{b\})\widetilde{N}} P'}\ \begin{array}{l} b \in \mathsf{n}(\widetilde{N}) \\ b\#\widetilde{a}, y\end{array} \qquad \textsc{sBrClose}\ \frac{P \xrightarrow[C]{\overline{x}!\,(\nu\widetilde{a})\widetilde{N}} P'}{(\nu b)P \xrightarrow[\exists^b x.C]{\tau} (\nu b)(\nu\widetilde{a})P'}$$

The concept of *frame of an agent* $\mathcal{F}(P)$ is used in the semantics: intuitively, it is the top-level assertions of an agent, including the top-level binders. Frames are of the form $F ::= \Psi \mid (\nu\, a)\Psi$, where a is bound in $(\nu\, a)\Psi$. The frame of a process denotes its contribution to parallel agents. For example, the frame $\mathcal{F}((\nu a)((\!|\Psi_1|\!) \mid \overline{M}\,\widetilde{N}\,.\,(\!|\Psi_3|\!) \mid (\!|\Psi_2|\!)))$ is $(\nu a)(\Psi_1 \otimes \Psi_2)$. Note that Ψ_3 is not included in the frame, since it occurs under a prefix. In order to define the symbolic operational semantics, we need a way to add the frame of a parallel process to the current transition constraint.

Definition 7.3 (*Adding Frames to Constraints*). We define $F \otimes C$ as follows:

$$\begin{array}{rcll} F \otimes (\nu a)C & = & (\nu a)(F \otimes C) & \text{where } a\#F \\ (\nu\widetilde{a})\Psi \otimes \{\!|\Psi' \vdash \varphi|\!\} & = & (\nu\widetilde{a})\{\!|\Psi \otimes \Psi' \vdash \varphi|\!\} & \text{where } \widetilde{a}\#\Psi', \varphi \\ (\nu\widetilde{a})\Psi \otimes \exists x.C & = & (\nu\widetilde{a})\exists x.(\Psi \otimes C) & \text{where } \widetilde{a}\#C \text{ and } x\#\widetilde{a}, \Psi \\ F \otimes (C \wedge D) & = & (F \otimes C) \wedge (F \otimes D) & \\ F \otimes C & = & C & \text{otherwise.} \end{array}$$

For the symbolic semantics to be able to pick out the original channel to be used to send a message, we require partial injectivity of channel connectivity in its left argument: we require that for all names a, the function $x \mapsto (x \dot{\leftrightarrow} a)$ is injective.

A process P is said to be assertion guarded if every occurrence of a $(\!|\Psi|\!)$ in P is a subterm of an input or an output. We require that processes are well formed: P is well formed if in every subterm of P of the form **case** $\widetilde{\varphi} : \widetilde{Q}$ every Q_i is assertion guarded, and in every subterm of P of the form $!Q$ we have that Q is assertion guarded.

We let the subject (or channel) of an action α be $\mathrm{subj}(\underline{x}?(\widetilde{y})) = \mathrm{subj}(\underline{x}(\widetilde{y})) = \mathrm{subj}(\overline{x}!\,(\nu\widetilde{a})\widetilde{N}) = \mathrm{subj}(\overline{x}\ (\nu\widetilde{a})\widetilde{N}) = x$ and $\mathrm{subj}(\tau) = \emptyset$. We also define the bound names (i.e., the private names) of a label as $\mathrm{bn}(\underline{x}?(\widetilde{y})) = \mathrm{bn}(\underline{x}(\widetilde{y})) = \widetilde{y}$ and $\mathrm{bn}(\overline{x}!\,(\nu\widetilde{a})\widetilde{N}) = \mathrm{bn}(\overline{x}\ (\nu\widetilde{a})\widetilde{N}) = \widetilde{a}$ and $\mathrm{bn}(\tau) = \emptyset$.

The structured symbolic operational semantics preserves well formedness and is defined in Tables I, II, and III. We first describe the broadcast rules in Table I. First consider the sBrOut rule: $\overline{M}\,\widetilde{N}.P \xrightarrow[\{\!|\mathbf{1}\vdash M \dot{\prec} y|\!\}]{\overline{y}!\,\widetilde{N}} P$. The solutions to its transition constraint are those that enable the subject M of the output prefix to broadcast on the fresh

Table II. Revised Symbolic Transition Rules for Binary Communication. The symmetric version of sCom is elided. In sCom, we assume that $\widetilde{c_1}\#y, \widetilde{c_2}, \Psi_2, M_2$ and $\widetilde{c_2}\#z, \Psi_1, M_1$ and let $C_{\mathrm{com}} = ((\nu\widetilde{c_1}\widetilde{c_2})\{\!|\Psi_1 \otimes \Psi_2 \vdash M_1 \dot{\leftrightarrow} M_2|\!\}) \wedge (((\nu\widetilde{c_2})\Psi_2) \otimes C_1) \wedge (((\nu\widetilde{c_1})\Psi_1) \otimes C_2)$. In sOpen, the expression $\nu\tilde{a} \cup \{b\}$ means the sequence $\tilde{a}$ with b inserted anywhere.

$$\text{sOut}\ \frac{y\#M, \widetilde{N}, P}{\overline{M}\,\widetilde{N}\,.\,P \xrightarrow[\{\!|\mathbf{1}\vdash M \dot{\leftrightarrow} y|\!\}]{\overline{y}\widetilde{N}} P} \qquad \text{sIn}\ \frac{y\#M, P, \widetilde{x}}{\underline{M}(\widetilde{x})\,.\,P \xrightarrow[\{\!|\mathbf{1}\vdash M \dot{\leftrightarrow} y|\!\}]{\underline{y}(\widetilde{x})} P}$$

$$\text{sCom}\ \frac{P \xrightarrow[(\nu\widetilde{c_1})\{\!|\Psi_1 \vdash M_1 \dot{\leftrightarrow} y|\!\} \wedge C_1]{\overline{y}\ (\nu\widetilde{a})\widetilde{N}} P' \qquad Q \xrightarrow[(\nu\widetilde{c_2})\{\!|\Psi_2 \vdash M_2 \dot{\leftrightarrow} z|\!\} \wedge C_2]{\underline{z}(\widetilde{x})} Q'}{P \mid Q \xrightarrow[C_{\mathrm{com}}]{\tau} (\nu\widetilde{a})(P' \mid Q'[\widetilde{x} := \widetilde{N}])}\ \begin{array}{l}|\widetilde{x}| = |\widetilde{N}| \\ \widetilde{a}\#Q\end{array} \qquad \text{sOpen}\ \frac{P \xrightarrow[C]{\overline{y}\ (\nu\widetilde{a})\widetilde{N}} P'}{(\nu b)P \xrightarrow[(\nu b)C]{\overline{y}\ (\nu\widetilde{a}\cup\{b\})\widetilde{N}} P'}\ \begin{array}{l}b \in \mathsf{n}(\widetilde{N}) \\ b\#\widetilde{a}, y\end{array}$$

Table III. Revised Symbolic Transition Rules Common to Broadcast and Binary Communication. A symmetric version of sPar is elided.

$$\text{sCase}\ \frac{P_i \xrightarrow[C]{\alpha} P'}{\mathbf{case}\ \widetilde{\varphi} : \widetilde{P} \xrightarrow[C \wedge \{\!|\mathbf{1}\vdash\varphi_i|\!\}]{\alpha} P'}\ \mathrm{subj}(\alpha)\#\varphi_i \qquad \text{sRep}\ \frac{P \mid !P \xrightarrow[C]{\alpha} P'}{!P \xrightarrow[C]{\alpha} P'}$$

$$\text{sPar}\ \frac{P \xrightarrow[C]{\alpha} P'}{P \mid Q \xrightarrow[\mathcal{F}(Q)\otimes C]{\alpha} P' \mid Q}\ \begin{array}{l}\mathrm{bn}(\alpha)\#Q \\ \alpha = \tau \vee \mathrm{subj}(\alpha)\#Q\end{array} \qquad \text{sScope}\ \frac{P \xrightarrow[C]{\alpha} P'}{(\nu b)P \xrightarrow[(\nu b)C]{\alpha} (\nu b)P'}\ b\#\alpha$$

$$\text{sInv}\ \frac{P[\widetilde{x} := \widetilde{M}] \xrightarrow[C]{\alpha} P'}{\mathrm{A}\langle\widetilde{M}\rangle \xrightarrow[C]{\alpha} P'}\ \begin{array}{l}\mathrm{A}\langle\widetilde{x}\rangle \Leftarrow P \\ |\widetilde{x}| = |\widetilde{M}|\end{array}$$

channel variable y. Similarly, in sBrIn, we can receive a broadcast from any channel x that the subject M of the input prefix can listen to. In sBrMerge, two inputs with the same labels are merged into one. In sBrCom, a broadcast of P is received by Q, substituting the message $\widetilde{N}$ for the input variables $\widetilde{y}$. The names $\widetilde{a}$ are restricted in P, so they must be fresh for Q. In both sBrMerge and sBrCom, each transition constraint is extended with the frame of the other process. In sBrOpen, the scope of the new name b that occurs in the message $\widetilde{N}$ is opened; we remember in the transition constraint that b is fresh. In sBrClose, a broadcast that has reached its lexical scope turns into an internal τ action. The scoping of the new names $\widetilde{a}$ is re-established.

The other symbolic rules in Tables II and III are similar to the broadcast rules, with two exceptions. In Rule sCase in Table III, we add the constraint that φ_i must hold to the transition constraint. In Rule sCom in Table II, we partially deconstruct the transition constraints of the input and the output transition, picking out the first conjunct. We then recombine the remainder of the transition constraints, adding the constraint that their channels are equivalent (i.e., $\Psi_1 \otimes \Psi_2 \vdash M_1 \dot{\leftrightarrow} M_2$), yielding C_{com}. Here the partial injectivity of $\dot{\leftrightarrow}$ is used to guarantee that M_1 is the channel that originated the transition.

7.2. Comparison with the Original Symbolic Operational Semantics

The symbolic semantics used in this article differs from the original semantics [Johansson et al. 2012] in four significant ways:

(1) support for broadcast communication (Table I);
(2) support for polyadic communication (sending of multiple message terms at once);

(3) no dependence on an external assertion environment ($\Psi \rhd$ later); and
(4) a new sCom rule, for reasons explained later.

The original version of the communication rule was as follows (omitting its freshness side conditions). To follow, the assertion environment "$\ldots \Psi \rhd$" collects the assertions of the context of the current process and can be ignored:

$$\textsc{Old-sCom}\ \frac{\Psi_Q \otimes \Psi \ \rhd\ P \xrightarrow[(\nu\widetilde{b}_P)\{\!|\Psi_1 \vdash M_1 \dot{\leftrightarrow} y|\!\}\wedge C_1]{\overline{y}\ (\nu\widetilde{a})\widetilde{N}} P' \qquad \Psi_P \otimes \Psi \ \rhd\ Q \xrightarrow[(\nu\widetilde{b}_Q)\{\!|\Psi_2 \vdash M_2 \dot{\leftrightarrow} z|\!\}\wedge C_2]{\underline{z}(\widetilde{x})} Q'}{\Psi \ \rhd\ P \mid Q \xrightarrow[C_{\text{old}}]{\tau} (\nu\widetilde{a})(P' \mid Q'[\widetilde{x} := \widetilde{N}])} \quad \begin{array}{c} |\widetilde{x}| = |\widetilde{N}| \\ \mathcal{F}(P) = (\nu\widetilde{b}_P)\Psi_P \\ \mathcal{F}(Q) = (\nu\widetilde{b}_Q)\Psi_Q \\ \Psi_1 = \Psi_2 = \Psi \otimes \Psi_P \otimes \Psi_Q \end{array}$$

In order to derive a transition with Old-sCom, we need to compute the frames of P and Q, equate the bound names in the frames with the ones appearing in the transition constraints such that $\mathcal{F}(P) = (\nu\widetilde{b}_P)\Psi_P$ and $\mathcal{F}(Q) = (\nu\widetilde{b}_Q)\Psi_Q$, and then check that $\Psi_1 = \Psi_2 = \Psi \otimes \Psi_P \otimes \Psi_Q$. However, these equalities fail in certain situations where we would expect them to hold.

Example 7.4. This example shows issues related to restrictions under process constructors **case** and replication (!). We use replication as an example; the issue when using **case** is analogous. Consider the process $P = !(\nu\, b)\overline{c}\, b.Q$ in the pi-calculus instance. In the original semantics, the symbolic output transition of P has the constraint $(\nu\, b)\{\!|\mathbf{1} \vdash c \dot{\leftrightarrow} x|\!\}$ since the frame of $(\nu\, b)\overline{c}\, b.Q$ (which is $(\nu\, b)\mathbf{1}$) is used in the derivation. When attempting to derive a communication between P and the process $\underline{c}(x).R$, the side condition $\mathcal{F}(P) = (\nu\widetilde{b}_P)\Psi_P$ of Old-sCom is impossible to satisfy: $\mathcal{F}(P) = (\nu\varepsilon)\mathbf{1}$ while the transition constraint of P is $(\nu\, b)\{\!|\mathbf{1} \vdash c \dot{\leftrightarrow} x|\!\}$, and the number of bound names thus differs.

A similar issue, related to the ordering of restrictions in the frame, applies when an inactive parallel process has top-level restrictions.

Example 7.5. Let $P = (\nu\, b)\overline{c}\, b.Q \mid (\nu\, a)\underline{c}(x).R$. In the original semantics, the symbolic output transition of P has the constraint $(\nu\, a)(\nu\, b)\{\!|\mathbf{1} \vdash c \dot{\leftrightarrow} x|\!\}$ but $\mathcal{F}(P) = (\nu\, b)(\nu\, a)\mathbf{1}$ where the order of the bound names is different.

Both these issues could be avoided if the binders of frames were so-called set+ binders [Huffman and Urban 2010] where order does not matter and redundant binders are ignored. However, such a notion of binders is not available in the version of Nominal Isabelle [Urban and Tasson 2005] that is used for the formalization of psi-calculi [Bengtson and Parrow 2009].

Example 7.6. This example show issues related to situations where assertion composition is noncommutative. Let the assertions be tuples $\widetilde{a}$ of names, composed using concatenation $\widetilde{a};\widetilde{b}$. Consider the premises of Old-sCom: in the original semantics, Ψ_1 will have a prefix $\Psi_Q; \Psi$ and Ψ_2 will have a prefix $\Psi_P; \Psi$. Since concatenation is noncommutative, the side condition $\Psi_1 = \Psi_2 = \Psi \otimes \Psi_P \otimes \Psi_Q$ of Old-Com cannot hold if Ψ_P and Ψ_Q are nonempty and $\mathrm{n}(\Psi_P) \neq \mathrm{n}(\Psi_Q)$. This makes it impossible for the two processes $(\!|a|\!) \mid c$ and $(\!|b|\!) \mid \overline{c}$ to communicate using Old-sCom.

These examples show that Rule Old-sCom makes too strong assumptions on the syntactic form of the constraints of the transitions in its premise. The original symbolic semantics still corresponds to the concrete semantics [Bengtson et al. 2011] in certain instances, such as when communicating processes do not contain restrictions and

Table IV. General Requirements on Substitution

$X[x := x]$	$=$	X	
$x[x := M]$	$=$	M	
$X[x := M]$	$=$	X	if $x\#X$
$X[x := L][y := M]$	$=$	$X[y := M][x := L]$	if $x\#y$, M and $y\#L$
$X[\tilde{x} := \tilde{T}]$	$=$	$((\tilde{y}\ \tilde{x}) \cdot X)[\tilde{y} := \tilde{T}]$	if $\tilde{y}\#X, \tilde{x}$

Table V. Requirements for Specific Data Types

$\mathrm{n}(M\sigma)$	$\supseteq$	$\mathrm{n}(M) \setminus \mathrm{n}(\sigma)$			
$\mathrm{n}(M[\tilde{a} := \tilde{L}])$	$\supseteq$	$\mathrm{n}(\tilde{L})$ when $\mathrm{n}(M) \supseteq \tilde{a}$	$\Psi \otimes \mathbf{1}$	$\simeq_{\mathcal{N}}$	Ψ
$(M \dot{\prec} N)\sigma$	$=$	$M\sigma \dot{\prec} N\sigma$	$\Psi \otimes \Psi'$	$\simeq_{\mathcal{N}}$	$\Psi' \otimes \Psi$
$(N \dot{\succ} M)\sigma$	$=$	$N\sigma \dot{\succ} M\sigma$	$\Psi_1 \otimes (\Psi_2 \otimes \Psi_3)$	$\simeq_{\mathcal{N}}$	$(\Psi_1 \otimes \Psi_2) \otimes \Psi_3$
$(M \dot{\leftrightarrow} N)\sigma$	$=$	$M\sigma \dot{\leftrightarrow} N\sigma$	$(\Psi \otimes \Psi')\sigma$	$\simeq_{\mathcal{N}}$	$\Psi\sigma \otimes \Psi'\sigma$
$\mathbf{1}\sigma$	$=$	$\mathbf{1}$	$\Psi \otimes \Psi_1$	$\simeq_{\mathcal{N}}$	$\Psi \otimes \Psi_2$ when $\Psi_1 \simeq_{\mathcal{N}} \Psi_2$

assertion composition satisfies the commutative monoid laws (not only modulo assertion equivalence). In contrast to OLD-sCOM, Rule sCOM in Table II does not make any assumptions about the number of bound names or on the structure of the assertion, and the corresponding broadcast rules sBRCOM and sBRMERGE in Table I do not make any assumptions at all about the form of their constraints.

7.3. Correctness of the Symbolic Operational Semantics

The proofs for the soundness and completeness of the symbolic semantics with respect to the concrete broadcast semantics [Borgström et al. 2011] mainly follow [Johansson et al. 2012]. The main exception is that their counterpart of Lemma 7.10, which describes the shape of transition constraints, does not hold in all cases, as seen in Examples 7.4, 7.5, and 7.6. We here instead prove a weaker result by considering assertions and frames modulo redundant restrictions (cf. Example 7.4), restriction ordering (cf. Example 7.5), and commutative monoid laws for assertion composition (cf. Example 7.6).

As for technical preliminaries, we assume the general properties of substitution in Table IV, and the homomorphism and name preservation laws in Table V. As an example, the standard notion of substitution in (nominal) term algebras satisfies all of these properties. We write $\Psi \simeq_{\mathcal{N}} \Psi'$ iff $\mathrm{n}(\Psi) = \mathrm{n}(\Psi')$, and for all φ, it holds that $\Psi \vdash \varphi$ iff $\Psi' \vdash \varphi$. We then assume the equivalences in Table V. As an example, they are satisfied when assertions are finite sets of equations on terms, with standard substitution.

The main difference to the original proofs is the introduction of an auxiliary relation on frames (Definition 7.7) in order to accurately describe the shape of transition constraints (Lemma 7.10) such that they can always be decomposed in Rule sCOM, unlike the case for OLD-sCOM.

Definition 7.7 (*AC-equivalence*). Associative/commutative equivalence (AC equivalence) of assertions is the smallest equivalence relation such that

(1) $\mathbf{1} \otimes \Psi \equiv_{\mathrm{AC}} \Psi$; and
(2) $\Psi_1 \otimes \Psi_2 \equiv_{\mathrm{AC}} \Psi_2 \otimes \Psi_1$; and
(3) $\Psi_1 \otimes (\Psi_2 \otimes \Psi_3) \equiv_{\mathrm{AC}} (\Psi_1 \otimes \Psi_2) \otimes \Psi_3$; and
(4) $\Psi_1 \equiv_{\mathrm{AC}} \Psi_2 \Rightarrow \Psi \otimes \Psi_1 \equiv_{\mathrm{AC}} \Psi \otimes \Psi_2$.

Frames $(\nu\widetilde{a})\Psi_1$ and $(\nu\widetilde{c})\Psi_2$ are AC equivalent, written $(\nu\widetilde{a})\Psi_1 \equiv_{\mathrm{AC}} (\nu\widetilde{c})\Psi_2$, if $\Psi_1 \equiv_{\mathrm{AC}} \Psi_2$ and $\{\widetilde{a}\} \cap \mathrm{n}(\Psi_1) = \{\widetilde{c}\} \cap \mathrm{n}(\Psi_2)$.

LEMMA 7.8. *AC equivalence is an equivalence relation on frames, and whenever* $F_1 \equiv_{\mathrm{AC}} F_2$, *we also have* $\mathrm{n}(F_1) = \mathrm{n}(F_2)$ *and* $(\nu a)F_1 \equiv_{\mathrm{AC}} (\nu a)F_2$ *and* $G \otimes F_1 \equiv_{\mathrm{AC}} G \otimes F_2$.

PROOF. Straightforward from the definitions, using the laws in Table V. □

As an example, guarded processes have frames that are AC equivalent to the unit frame **1**.

LEMMA 7.9. *If P is assertion guarded, then $\mathcal{F}(P) \equiv_{\mathrm{AC}} \mathbf{1}$.*

PROOF. By induction on P. □

The following lemma characterizes the shape of the constraints of point-to-point input and output transitions. The first conjunct in the constraint is always a channel equivalence constraint (between the object M of the original prefix and the transition object variable y) that must hold under a frame $(\nu\widetilde{c})\Psi$ that is AC equivalent to that of the original process P. The lemma is used in the proof of Theorem 7.12 to show that the precondition on the shape of the transitions in Rule sCOM always holds.

LEMMA 7.10 (FORM OF CONSTRAINT). *Let $\alpha = \overline{y}\,(\nu\widetilde{a})\widetilde{N}$ or $\alpha = \underline{y}(\widetilde{x})$. If $P \xrightarrow[C]{\alpha} P'$ and $y\#P$, then there exist $\widetilde{c}, \Psi, M$, and D such that $\mathcal{F}(P) \equiv_{\mathrm{AC}} (\nu\widetilde{c})\Psi$ and $y\#\widetilde{c}, \Psi, M, D$ and $C = (\nu\widetilde{c})\{\!|\Psi \vdash M \dot{\leftrightarrow} y|\!\} \wedge D$.*

PROOF. By induction on the derivation of $P \xrightarrow[C]{\alpha} P'$. A base case is as follows.

sOUT. In this case, the transition is derived by

$$\text{sOUT}\ \frac{}{\overline{K}\widetilde{N}\,.\,P \xrightarrow[\{\!|\mathbf{1}\vdash K\dot{\leftrightarrow}y|\!\}]{\overline{y}\widetilde{N}} P.}\ y\#K, \widetilde{N}, P$$

Here $\widetilde{c} = \epsilon$, $\Psi = \mathbf{1}$, $M = K$, and $D = \mathbf{true}$, where $\mathcal{F}(\overline{K}\,\widetilde{N}\,.\,P) = \mathbf{1}$.

The cases that require the use of AC equivalence are the following.

sCASE. In this case, the transition is derived by

$$\text{sCASE}\ \frac{P_i \xrightarrow[C]{\alpha} P'}{\mathbf{case}\ \widetilde{\varphi} : \widetilde{P} \xrightarrow[C\wedge\{\!|\mathbf{1}\vdash\varphi_i|\!\}]{\alpha} P'.}\ \mathrm{bn}(\alpha)\#\varphi_i$$

By induction, we get $M, D', \Psi, \widetilde{c}$ such that $C = (\nu\widetilde{c})\{\!|\Psi \vdash M \dot{\leftrightarrow} y|\!\} \wedge D'$ with $y\#D'$ and $\mathcal{F}(P_i) \equiv_{\mathrm{AC}} (\nu\widetilde{c})\Psi$. Let $D = D' \wedge \{\!|\mathbf{1} \vdash \varphi_i|\!\}$; since $y\#\mathbf{case}\ \widetilde{\varphi} : \widetilde{P}$, we also have that $y\#D$. By well formedness, P_i is guarded, so by Lemma 7.9, $\mathcal{F}(P_i) \equiv_{\mathrm{AC}} \mathbf{1}$. By transitivity, $\mathcal{F}(P) = \mathbf{1} \equiv_{\mathrm{AC}} (\nu\widetilde{c})\Psi$.

sPAR. In this case, the transition is derived by

$$\text{sPAR}\ \frac{P \xrightarrow[C]{\alpha} P'}{P \mid Q \xrightarrow[\mathcal{F}(Q)\otimes C]{\alpha} P' \mid Q}\ \begin{array}{c}\mathrm{bn}(\alpha)\#Q\\ \alpha = \tau \vee \mathrm{subj}(\alpha)\#Q.\end{array}$$

By induction, there are $M, D', \Psi, \widetilde{c}$ such that $C = (\nu\widetilde{c})\{\!|\Psi \vdash M \dot{\leftrightarrow} y|\!\} \wedge D'$ with $y\#D'$ and $\mathcal{F}(P) \equiv_{\mathrm{AC}} (\nu\widetilde{c})\Psi$. Let $D = \mathcal{F}(Q) \otimes D'$; since $y\#P|Q$, we also have that $y\#D$. By Lemma 7.8, $\mathcal{F}(P \mid Q) \equiv_{\mathrm{AC}} ((\nu\widetilde{c})\Psi) \otimes \mathcal{F}(Q) \equiv_{\mathrm{AC}} \mathcal{F}(Q) \otimes (\nu\widetilde{c})\Psi$.

sSCOPE. In this case, the transition is derived by

$$\text{sSCOPE}\ \frac{P \xrightarrow[C]{\alpha} P'}{(\nu b)P \xrightarrow[(\nu b)C]{\alpha} (\nu b)P'}\ b\#\alpha.$$

Table VI. Concrete Semantics. Symmetric versions of cBrCom, cCom, and cPar are elided. In Rules cBrCom and cBrMerge and cCom, we assume that $\mathcal{F}(P) = (\nu\widetilde{b}_P)\Psi_P$ and $\mathcal{F}(Q) = (\nu\widetilde{b}_Q)\Psi_Q$, where $\widetilde{b}_P$ is fresh for P, $\widetilde{b}_Q$, Q, and Ψ, and that $\widetilde{b}_Q$ is fresh for Q, $\widetilde{b}_P$, P, and Ψ. In Rule cPar, we assume that $\mathcal{F}(Q) = (\nu\widetilde{b}_Q)\Psi_Q$, where $\widetilde{b}_Q$ is fresh for Ψ, P, and α. In Rules cOpen and cBrOpen, the expression $\tilde{a}\cup\{b\}$ means the sequence $\tilde{a}$ with b inserted anywhere.

$$\textsc{cBrOut}\ \frac{\Psi \vdash M \dot{\prec} K}{\Psi \rhd \overline{M}!\widetilde{N}.P \xrightarrow{\overline{K}!\widetilde{N}} P} \qquad \textsc{cBrIn}\ \frac{\Psi \vdash K \dot{\succ} M \qquad |\widetilde{x}| = |\widetilde{N}|}{\Psi \rhd \underline{M}?(\widetilde{x}).P \xrightarrow{\underline{K}?\widetilde{N}} P[\widetilde{x} := \widetilde{N}]}$$

$$\textsc{cBrMerge}\ \frac{\Psi_Q \otimes \Psi \rhd P \xrightarrow{\underline{K}?\widetilde{N}} P' \qquad \Psi_P \otimes \Psi \rhd Q \xrightarrow{\underline{K}?\widetilde{N}} Q'}{\Psi \rhd P \mid Q \xrightarrow{\underline{K}?\widetilde{N}} P' \mid Q'}$$

$$\textsc{cBrCom}\ \frac{\Psi_Q \otimes \Psi \rhd P \xrightarrow{\overline{K}!(\nu\widetilde{a})\widetilde{N}} P' \qquad \Psi_P \otimes \Psi \rhd Q \xrightarrow{\underline{K}?\widetilde{N}} Q'}{\Psi \rhd P \mid Q \xrightarrow{\overline{K}!(\nu\widetilde{a})\widetilde{N}} P' \mid Q'}\ \begin{array}{c}\widetilde{b}_P\widetilde{b}_Q\#K \\ \widetilde{a}\#Q\end{array}$$

$$\textsc{cBrOpen}\ \frac{\Psi \rhd P \xrightarrow{\overline{K}!(\nu\widetilde{a})\widetilde{N}} P'}{\Psi \rhd (\nu b)P \xrightarrow{\overline{K}!(\nu\widetilde{a}\cup\{b\})\widetilde{N}} P'}\ \begin{array}{c}b\#\widetilde{a}, \Psi, K \\ b \in \mathrm{n}(N)\end{array} \qquad \textsc{cBrClose}\ \frac{\Psi \rhd P \xrightarrow{\overline{K}!(\nu\widetilde{a})\widetilde{N}} P'}{\Psi \rhd (\nu b)P \xrightarrow{\tau} (\nu b)(\nu\widetilde{a})P'}\ \begin{array}{c}b \in \mathrm{n}(K) \\ b\#\Psi\end{array}$$

$$\textsc{cOut}\ \frac{\Psi \vdash M \dot{\leftrightarrow} K}{\Psi \rhd \overline{M}\,\widetilde{N}.P \xrightarrow{\overline{K}\widetilde{N}} P} \qquad \textsc{cIn}\ \frac{\Psi \vdash M \dot{\leftrightarrow} K \qquad |\widetilde{x}| = |\widetilde{N}|}{\Psi \rhd \underline{M}(\widetilde{x}).P \xrightarrow{\underline{K}\,\widetilde{N}} P[\widetilde{x} := \widetilde{N}]}$$

$$\textsc{cCom}\ \frac{\Psi \otimes \Psi_P \otimes \Psi_Q \vdash M \dot{\leftrightarrow} K \qquad \Psi_Q \otimes \Psi \rhd P \xrightarrow{\overline{M}(\nu\widetilde{a})\widetilde{N}} P' \qquad \Psi_P \otimes \Psi \rhd Q \xrightarrow{\underline{K}\,\widetilde{N}} Q'x}{\Psi \rhd P \mid Q \xrightarrow{\tau} (\nu\widetilde{a})(P' \mid Q')}\ \begin{array}{c}\widetilde{b}_P\#M \\ \widetilde{b}_Q\#K \\ \widetilde{a}\#Q\end{array}$$

$$\textsc{cOpen}\ \frac{\Psi \rhd P \xrightarrow{\overline{M}(\nu\widetilde{a})N} P'}{\Psi \rhd (\nu b)P \xrightarrow{\overline{M}(\nu\widetilde{a}\cup\{b\})N} P'}\ \begin{array}{c}b\#\widetilde{a}, \Psi, M \\ b \in \mathrm{n}(N)\end{array} \qquad \textsc{cCase}\ \frac{\Psi \rhd P_i \xrightarrow{\alpha} P' \qquad \Psi \vdash \varphi_i}{\Psi \rhd \mathbf{case}\,\widetilde{\varphi} : \widetilde{P} \xrightarrow{\alpha} P'}$$

$$\textsc{cRep}\ \frac{\Psi \rhd P \mid !P \xrightarrow{\alpha} P'}{\Psi \rhd !P \xrightarrow{\alpha} P'} \qquad \textsc{cPar}\ \frac{\Psi_Q \otimes \Psi \rhd P \xrightarrow{\alpha} P'}{\Psi \rhd P \mid Q \xrightarrow{\alpha} P' \mid Q}\ \mathrm{bn}(\alpha)\#Q$$

$$\textsc{cScope}\ \frac{\Psi \rhd P \xrightarrow{\alpha} P'}{\Psi \rhd (\nu b)P \xrightarrow{\alpha} (\nu b)P'}\ b\#\alpha, \Psi \qquad \textsc{cInv}\ \frac{\Psi \rhd P[\widetilde{x} := \widetilde{M}] \xrightarrow{\alpha} P'}{\Psi \rhd \mathrm{A}\langle\widetilde{M}\rangle \xrightarrow{\alpha} P'}\ \begin{array}{c}\mathrm{A}\langle\widetilde{x}\rangle \Leftarrow P \\ |\widetilde{x}| = |\widetilde{M}|\end{array}$$

By induction, there exist $\widetilde{c}$, Ψ, M, and D' such that $C = (\nu\widetilde{c})(\Psi \vdash M \dot{\leftrightarrow} y) \wedge D'$ with $y\#M, D$ and $\mathcal{F}(P) \equiv_{\mathrm{AC}} (\nu\widetilde{c})\Psi$. Let $D = (\nu b)D'$; a fortiori $y\#(\nu b)D$. By Lemma 7.8, $\mathcal{F}((\nu b)P) \equiv_{\mathrm{AC}} (\nu b)(\nu\widetilde{c})\Psi$.

sOpen. As sScope.

sRep. In this case, the transition is derived by

$$\textsc{sRep}\ \frac{P \mid !P \xrightarrow[C]{\alpha} P'}{!P \xrightarrow[C]{\alpha} P'}\ .$$

By induction, there exist $\widetilde{c}$, Ψ, M, and D such that $C = (\nu\widetilde{c})(\Psi \vdash M \dot{\leftrightarrow} y) \wedge D$ with $y\#M, D$ and $\mathcal{F}(P|!P) \equiv_{\mathrm{AC}} (\nu\widetilde{c})\Psi$. By well formedness, P is guarded, so by Lemma 7.9, $\mathcal{F}(P|!P) \equiv_{\mathrm{AC}} \mathbf{1}$. By transitivity, $\mathcal{F}(!P) = \mathbf{1} \equiv_{\mathrm{AC}} (\nu\widetilde{c})\Psi$. □

We prove soundness and completeness of the symbolic semantics of this article with respect to a polyadic version of the concrete semantics of broadcast psi-calculi [Borgström et al. 2011], which we show in Table VI.

The soundness theorem and its proof follow [Johansson et al. 2012], apart from the weaker preconditions of Rule sCom (compared to Old-sCom) and the new cases for broadcast actions.

THEOREM 7.11 (SOUNDNESS OF SYMBOLIC TRANSITIONS). *If* $P \xrightarrow[C]{\alpha} P'$ *and* $(\sigma, \Psi) \models C$ *and* $\mathrm{bn}(\alpha)\#\sigma$, *then* $\Psi \rhd P\sigma \xrightarrow{\alpha\sigma} P'\sigma$.

PROOF. By induction on the inference of $P \xrightarrow[C]{\alpha} P'$. □

The proof of the completeness theorem follows [Johansson et al. 2012], apart from new cases for the broadcast rules. In the cCOM case of the proof, Lemma 7.10 is used to show that the symbolic transitions obtained from the induction hypothesis are of the right form to apply Rule sCOM.

THEOREM 7.12 (COMPLETENESS OF SYMBOLIC TRANSITIONS).

—*If* $\Psi \rhd P\sigma \xrightarrow{\tau} P'$, *then* $\exists C, Q\,.\; P \xrightarrow[C]{\tau} Q$, $Q\sigma = P'$, *and* $(\sigma, \Psi) \models C$.

—*If* $\Psi \rhd P\sigma \xrightarrow{\alpha} P'$, $\alpha \neq \tau$, $y\#P, \mathrm{bn}(\alpha), \sigma$, *and* $\mathrm{bn}(\alpha)\#\sigma, P$, *then* $\exists C, \alpha', Q.\; P \xrightarrow[C]{\alpha'} Q$, $Q\sigma = P'$, $\mathrm{subj}(\alpha') = y$, $\alpha'\sigma' = \alpha$, *and* $(\sigma', \Psi) \models C$ *where* $\sigma' = \sigma[y := \mathrm{subj}(\alpha)]$.

PROOF. By induction on the inference of $\Psi \rhd P\sigma \xrightarrow{\alpha} P'\sigma$. □

8. RELATED WORK

Our previous work [Borgström et al. 2011] presented the broadcast extension of psi-calculi and a model of a routing protocol for ad hoc networks. In the present article, we have given a corresponding symbolic semantics and several new example models.

The precursors of the PWB are the Concurrency Workbench [Cleaveland et al. 1993] for CCS and the Mobility Workbench [Victor and Moller 1994] for pi-calculus. The tool mCRL2 [Cranen et al. 2013] for ACP admits higher-order sorted free algebras and equational logics. PAT3 [Liu et al. 2011] includes a CSP♯ [Sun et al. 2009] module where actions are built over types like Booleans and integers are extended with C♯ like programs. ProVerif [Blanchet 2011] is a verification tool for the applied pi-calculus [Abadi and Fournet 2001], an extension of the pi-calculus that is specialized for security protocol verification. The tool is parametric in a term language equipped with equations and unidirectional rewrite rules but works in a fixed logic (predicate logic with equality). ProVerif does not include a symbolic simulator or a general bisimulation checker.

Our symbolic semantics and bisimulation generation algorithm (slight variations of our previous work [Johansson et al. 2012]) are to a large extent based on the pioneering work by Hennessy and Lin [1995] for value-passing CCS, later specialized for the pi-calculus by Boreale and De Nicola [1996] and independently by Lin [1996, 2000].

9. FUTURE WORK

It would be interesting to investigate other notions of bisimulation for wireless communication [Merro 2007], including machine-checked proofs of their meta-theoretical properties. We have performed initial work [Åman Pohjola et al. 2013] on modeling discrete time and are considering extensions to other quantitative aspects of wireless networks, including probabilities, distance, and energy.

ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed in the ACM Digital Library.

ACKNOWLEDGMENTS

We thank the anonymous referees of ACSD 2013 and TECS for their comments.

REFERENCES

Martín Abadi and Cédric Fournet. 2001. Mobile values, new names, and secure communication. In *Proc. of POPL'01*. ACM Press, New York, NY, 104–115. DOI:http://dx.doi.org/10.1145/373243.360213

Martín Abadi and Andrew D. Gordon. 1997. A calculus for cryptographic protocols: The spi calculus. In *Proceedings of the 4th ACM Conference on Computer and Communications Security*. ACM, 36–47. DOI:http://dx.doi.org/10.1145/266420.266432

Johannes Åman Pohjola, Johannes Borgström, Joachim Parrow, Palle Raabjerg, and Ioana Rodhe. 2013. *Negative Premises in Applied Process Calculi*. Technical Report 2013-014. Dept. of Information Technology, Uppsala University.

K. A. Bartlett, R. A. Scantlebury, and P. T. Wilkinson. 1969. A note on reliable full-duplex transmission over half-duplex links. *Commun. ACM* 12, 5 (May 1969), 260–261. DOI:http://dx.doi.org/10.1145/362946.362970

Jesper Bengtson, Magnus Johansson, Joachim Parrow, and Björn Victor. 2011. Psi-calculi: A framework for mobile processes with nominal data and logic. *Logical Methods Comput. Sci.* 7, 1 (2011), Article 11, 44 pages. DOI:http://dx.doi.org/10.2168/LMCS-7(1:11)2011

Jesper Bengtson and Joachim Parrow. 2009. Psi-calculi in Isabelle. In *Proc. of TPHOLs'09 (LNCS)*. LCNS, Springer, 99–114. DOI:http://dx.doi.org/10.1007/978-3-642-03359-9_9

Bruno Blanchet. 2011. Using Horn clauses for analyzing security protocols. In *Formal Models and Techniques for Analyzing Security Protocols*, Véronique Cortier and Steve Kremer (Eds.). Vol. 5. IOS Press, 86–111. DOI:http://dx.doi.org/10.3233/978-1-60750-714-7-86

Michele Boreale and Rocco De Nicola. 1996. A symbolic semantics for the π-calculus. *Information and Computation* 126, 1 (1996), 34–52. DOI:http://dx.doi.org/10.1006/inco.1996.0032

Johannes Borgström, Ramūnas Gutkovas, Ioana Rodhe, and Björn Victor. 2013. A parametric tool for applied process calculi. In *Proc. of ACSD'13*. IEEE, Los Alamitos, CA, 187–192. DOI:http://dx.doi.org/10.1109/ACSD.2013.22

Johannes Borgström, Shuqin Huang, Magnus Johansson, Palle Raabjerg, Björn Victor, Johannes Åman Pohjola, and Joachim Parrow. 2011. Broadcast psi-calculi with an application to wireless protocols. In *Proc. of SEFM'11*. Springer, 74–89. DOI:http://dx.doi.org/10.1007/978-3-642-24690-6_7

Johannes Borgström, Shuqin Huang, Magnus Johansson, Palle Raabjerg, Björn Victor, Johannes Åman Pohjola, and Joachim Parrow. 2013. Broadcast psi-calculi with an application to wireless protocols. In *Software and Systems Modeling* (2013). Springer Berlin Heidelberg, 1–16. DOI:http://dx.doi.org/10.1007/s10270-013-0375-z.

Maria Grazia Buscemi and Ugo Montanari. 2007. CC-Pi: A constraint-based language for specifying service level agreements. In *Proc. of ESOP'07*, Rocco De Nicola (Ed.). LNCS, Vol. 4421. Springer, 18–32.

Marco Carbone and Sergio Maffeis. 2003. On the expressive power of polyadic synchronisation in π-calculus. *Nordic Journal of Computing* 10, 2 (2003), 70–98.

Rance Cleaveland, Joachim Parrow, and Bernhard Steffen. 1993. The concurrency workbench: A semantics-based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems* 15, 1 (1993), 36–72. DOI:http://dx.doi.org/10.1145/151646.151648

Sjoerd Cranen, Jan Friso Groote, Jeroen J. A. Keiren, Frank P. M. Stappers, Erik P. Vink, Wieger Wesselink, and Tim A. C. Willemse. 2013. An overview of the mCRL2 toolset and its recent advances. In *Proc. of TACAS'13 (LNCS)*, Vol. 7795. Springer, 199–213. DOI:http://dx.doi.org/10.1007/978-3-642-36742-7_15

Fatemeh Ghassemi, Willem Fokkink, and Ali Movaghar. 2008. Restricted broadcast process theory. In *Proc. of SEFM'08.* 345–354. DOI:http://dx.doi.org/10.1109/SEFM.2008.25

Jens Chr. Godskesen. 2010. Observables for mobile and wireless broadcasting systems. In *Coordination Models and Languages*, Dave Clarke and Gul Agha (Eds.). LNCS, Vol. 6116. Springer, 1–15. DOI:http://dx.doi.org/10.1007/978-3-642-13414-2_1

Ramūnas Gutkovas and Johannes Borgström. 2013. The Psi-Calculi Workbench. Retrieved from http://www.it.uu.se/research/group/mobility/applied/psiworkbench.

Matthew Hennessy and Huimin Lin. 1995. Symbolic bisimulations. *Theoret. Comput. Sci.* 138, 2 (1995), 353–389. DOI:http://dx.doi.org/10.1016/0304-3975(94)00172-F

Brian Huffman and Christian Urban. 2010. A new foundation for nominal Isabelle. In *Proc. of ITP'10*. Springer, 35–50. DOI:http://dx.doi.org/10.1007/978-3-642-14052-5_5

Magnus Johansson, Jesper Bengtson, Joachim Parrow, and Björn Victor. 2010. Weak equivalences in psi-calculi. In *Proc. of LICS 2010*. IEEE, 322–331. DOI:http://dx.doi.org/10.1109/LICS.2010.30

Magnus Johansson, Björn Victor, and Joachim Parrow. 2012. Computing strong and weak bisimulations for psi-calculi. *J. Logic Algebraic Program* 81, 3 (2012), 162–180. DOI:http://dx.doi.org/10.1016/j.jlap.2012.01.001

Huimin Lin. 1996. Symbolic transition graph with assignment. In *Proc. of CONCUR'96*. LNCS, Vol. 1119. Springer, 50–65. DOI:http://dx.doi.org/10.1007/3-540-61604-7_47

Huimin Lin. 2000. Computing bisimulations for finite-control pi-calculus. *J. Comput. Sci. Technol.* 15, 1 (2000), 1–9. DOI:http://dx.doi.org/10.1007/BF02951922

Yang Liu, Jun Sun, and Jin Song Dong. 2011. PAT 3: An extensible architecture for building multi-domain model checkers. In *Proc. of ISSRE'11*. IEEE, Los Alamitos, CA, 190–199. DOI:http://dx.doi.org/10.1109/ISSRE.2011.19

Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. 2002. TAG: A Tiny AGgregation service for ad-hoc sensor networks. *SIGOPS Oper. Syst. Rev.* 36, SI (Dec. 2002), 131–146. DOI:http://dx.doi.org/10.1145/844128.844142

Massimo Merro. 2007. An observational theory for mobile ad hoc networks. *Electronical Notes in Theoretical Computer Science* 173 (April 2007), 275–293. DOI:http://dx.doi.org/10.1016/j.entcs.2007.02.039

Robin Milner, Joachim Parrow, and David Walker. 1992a. A calculus of mobile processes, I. *Inf. Comput.* 100, 1 (1992), 1–40. DOI:http://dx.doi.org/10.1016/0890-5401(92)90008-4

Robin Milner, Joachim Parrow, and David Walker. 1992b. A calculus of mobile processes, II. *Inf. Comput.* 100, 1 (1992), 41–77. DOI:http://dx.doi.org/10.1016/0890-5401(92)90009-5

Joachim Parrow, Johannes Borgström, Palle Raabjerg, and Johannes Åman Pohjola. 2013. Higher-order psi-calculi. *Math. Struct. Comput. Sci.* FirstView (June 2013), 1–37. DOI:http://dx.doi.org/10.1017/S0960129513000170

Andrew M. Pitts. 2003. Nominal logic, a first order theory of names and binding. *Inf. Comput.* 186, 2 (2003), 165–193. DOI:http://dx.doi.org/10.1016/S0890-5401(03)00138-X

PolyML. 2013. Poly/ML. Retrieved from http://www.polyml.org.

Jun Sun, Yang Liu, Jin Song Dong, and Chunqing Chen. 2009. Integrating specification and programs for system modeling and verification. In *Proc. TASE'09*. IEEE, 127–135. DOI:http://dx.doi.org/10.1109/TASE.2009.32

Christian Urban and Christine Tasson. 2005. Nominal techniques in Isabelle/HOL. In *CADE*, Robert Nieuwenhuis (Ed.). LNCS, Vol. 3632. Springer, 38–53. DOI:http://dx.doi.org/10.1007/11532231_4

Björn Victor and Faron Moller. 1994. The mobility workbench—A tool for the π-calculus. In *Proc. of CAV'94*, David Dill (Ed.). LCNS, Vol. 818. Springer, 428–440. DOI:http://dx.doi.org/10.1007/3-540-58179-0_73

Lucian Wischik and Philippa Gardner. 2005. Explicit fusions. *Theoret. Comput. Sci.* 304, 3 (2005), 606–630. DOI:http://dx.doi.org/10.1016/j.tcs.2005.03.017

Received November 2013; revised March 2014; accepted September 2014