Languages, Logics, Types and Tools for Concurrent System Modelling

RAMŪNAS GUTKOVAS
Abstract


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
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.

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

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

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.

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


*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.


*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.


*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.


Vi kan ha modeller på olika abstraktionsnivåer: från ganska enkla modeller som är lätt 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 komplex beteende med abstrakt beteende samtidigt som korrekthetsegenskaper behålls. Till exempel, i

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 felaktigt. 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 visas.

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 våletablerat session typ system för att hantera opålitlig kommunikation. Vi undersöker i första hand samtidigt kommunicerande system. I sådana system kommunikerar 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 programming1 “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

---

1The 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 $5 + "\text{foobar}"$ that applies addition to $5$ of type integer and "$\text{foobar}$" 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 known 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 $\mathcal{A}$. 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 $\overline{a}.P \xrightarrow{a} P$.

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

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 $\text{tick}$, as follows $\text{CLOCK} = \text{tick} \cdot \text{tick} \cdot 0$. The CLOCK process performs an action $\text{tick}$ and continues as $\text{tick} \cdot 0$, and again performs an action $\text{tick}$ and continue as $0$, that is, it stops performing any actions:

$\text{CLOCK} \xrightarrow{\text{tick}} \text{tick} \cdot 0 \xrightarrow{\text{tick}} 0$.

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

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

Clearly, the notion of an action is an abstraction. In the example, the dual actions of output $\text{tick}$ and input $\text{tick}$ 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

$P \parallel Q$.

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 $P' \parallel Q$, using the notation

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

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

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

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 \( P' | Q' \) by performing the silent action \( \tau \):

\[
P | Q \xrightarrow{\tau} P' | Q'.
\]

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 \( \tau \) action of the form \( \tau.P \).

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

\texttt{COMPONENT | CLOCK.}

Let us expand the definitions to:

\[
\texttt{tick.0} | \texttt{tick.tick.0}.
\]

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

\[
\texttt{tick.0} | \texttt{tick.tick.0} \xrightarrow{\tau} \texttt{0} | \texttt{tick.0}.
\]

The first parallel component emitted the \( \texttt{tick} \) action, the second component emitted the \( \texttt{tick} \) 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 \( \texttt{COMPONENT} \), that is, \( \texttt{COMPONENT | COMPONENT | CLOCK} \), then there are two possible continuations at this point: the first, \( \texttt{0} | \texttt{tick.0} | \texttt{tick.0} \), and the second \( \texttt{tick.0} | \texttt{0} | \texttt{tick.0} \). By using the arrow notation, we depict the transitions as follows

\[
\begin{align*}
0 & \xrightarrow{\tau} \texttt{tick.0} | \texttt{tick.0} \\
\texttt{tick.0} & \xrightarrow{\tau} \texttt{0} | \texttt{tick.tick.0} \\
\texttt{tick.0} & \xrightarrow{\tau} \texttt{0} | \texttt{tick.0}
\end{align*}
\]

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 $0 | 0 | 0$:

```
  0 | \text{\texttt{tick}}.0 | \text{\texttt{tick}}.0
     \tau
```

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., $\text{\texttt{tick}}.0 | \text{\texttt{tick}}.0 | \text{\texttt{tick}}.0 | \text{\texttt{tick}}.0$, 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 $P \overset{a}{\rightarrow} P'$, then $P + Q \overset{a}{\rightarrow} P'$. 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 $Q \overset{a}{\rightarrow} Q'$ then $P + Q \overset{a}{\rightarrow} Q'$. 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

$$P | P | \cdots$$

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 $P | !P \overset{a}{\rightarrow} P'$ then $!P \overset{a}{\rightarrow} P'$. 

26
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

\[ A \overset{\text{def}}{=} P, \]

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 \( A \overset{\text{def}}{=} P \parallel A \), 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

\[ \text{CLOCK}' = !\text{tick.0}. \]

The system \( \text{CLOCK}' \parallel \text{COMPONENT}_1 \parallel \ldots \parallel \text{COMPONENT}_n \) 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] generalises CCS by introducing message passing between processes while preserving synchronous behaviour. The pi-calculus

\[ \text{Figure 2.1. The grammar of a CCS process.} \]
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 $\mathcal{N}$ 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 $\overline{ab}$ 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 $ab$ 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 $\tau$. 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 $b$,

$$\overline{ab}.P.$$ 

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

$$\overline{ab}.P \xrightarrow{\overline{ab}} P.$$ 

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 $ab$ 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 $P' = P\{b/x\}$:

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

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:

\[
\begin{align*}
P\{a/x\} & \xrightarrow{aa} P\{b/x\} \\
a(x).P & \xrightarrow{ac} P\{c/x\} \\
& \xrightarrow{ad} P\{d/x\} \\
\end{align*}
\]

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 \(\text{\textendash}\text{\textendash}b.P\) 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 \(\text{fn}(P)\). 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 \(a(b).\text{\textendash}d.\text{\textendash}0\), and \(a(c).\text{\textendash}d.\text{\textendash}0\), and \(a(e).\text{\textendash}d.\text{\textendash}0\). 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 \(a(d).\text{\textendash}d.\text{\textendash}0\) 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 \(P \parallel Q \xrightarrow{\tau} P' \parallel Q'\) whenever \(P \xrightarrow{ab} P'\) and \(Q \xrightarrow{\text{\textendash}b} Q'\). For example, \(a(x).\text{\textendash}y.\text{\textendash}0 \parallel \text{\textendash}b.\text{\textendash}0\) synchronises by performing the silent \(\tau\) action and continues as \(\text{\textendash}y.\text{\textendash}0\), because the left hand side parallel component performs the input action \(ab\) (among all the other
possible input actions) \(a(x).\overline{\alpha}y.0 \xrightarrow{ab} by.0\) where \(by.0 = (\overline{\alpha}y.0)\{b/x\}\); and the right hand side performs \(ab\) by \(\overline{\alpha}b.0 \xrightarrow{\overline{\alpha}b} 0\). 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 \(a(x).\overline{\alpha}b\) 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 \(\overline{\alpha}.P\) is captured by the \(\pi\)-calculus operator \(\overline{\alpha}w.P\); and the CCS operator \(\alpha .P\) 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 \(\text{if } a = b \text{ then } P \text{ else } Q\) 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 \(\overline{\alpha}(\nu b)b\) for \(a\) and \(b\) names, where \((\nu b)\) denotes that the name \(b\) is restricted. For example, \((\nu b)\overline{\alpha}b.P \xrightarrow{\overline{\alpha}(\nu b)b} P\). In this transition, the name \(b\) in the action \(\overline{\alpha}(\nu b)b\) also binds into \(P\).

The restricted name is seen as private. For example, in the process \((\nu b)(\overline{\alpha}a.P | b(x).Q) | b(x).R\) 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

\[(vb)(P | Q\{a/x\}) | b(x).R\]

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:

\[P | Q \xrightarrow{\sigma(vb)b} (vb)P' | Q' \] whenever \[P \xrightarrow{a} P'\] and \[Q \xrightarrow{ab} Q'\], 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 \(x\).

\[
\text{SERVER} = (vs)\overline{srv}vs.s(r).\overline{s}\text{response}.\text{SERVER}
\]

\[
\text{CLIENT} = srv(x).\overline{x}\text{request}.x(r).0
\]

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.

\[
\text{SERVER} | \text{CLIENT} | \text{CLIENT} \xrightarrow{\delta} (vs)(s(r).\overline{s}\text{response}.\text{SERVER} | \overline{x}\text{request}.s(r).0) | \text{CLIENT}
\]

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.
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

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

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 \(\text{AND}\) 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 \( a \land b \) operation, the AND server could instead be modelled as

\[
\text{AND}(x, y, z) = x(a).y(b).z(a \land b).
\]

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 \( \text{decrypt}(\text{encrypt}(M, k), k) = M \). 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**

Psi-calculi 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 \(\overline{M}(a_1, \ldots, a_n) N\) 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 \(\varphi\), a set of assertions \(A\) ranged over by \(\Psi\), and an entailment relation \(\Psi \vdash \varphi\) determining the condition \(\varphi\) that is made true by the assertion \(\Psi\).

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

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

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 \(\Psi\), 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

\[ \Psi \uparrow MN.P \xrightarrow{M/N} P \]

where \( \Psi \vdash M \leftrightarrow M' \).

The case \( \varphi_1 : P_1 \uplus \cdots \uplus \varphi_n : P_n \) 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 \( \varphi \in C \). For example, if we have negation \( \neg \) in the condition language defined in \( C \), then if-then-else \( \text{if } \varphi \text{ then } P \text{ else } Q \) can be defined as \( \text{case } \varphi : P \uplus \neg \varphi : Q \). The choice of the pi-calculus can be recovered as follows. Given that there is an always entailed condition \( \text{true} \) (i.e., for any \( \Psi \), \( \Psi \vdash \text{true} \)), then \( P + Q \) can be expressed in psi-calculi as \( \text{case } \text{true} : P \uplus \text{true} : Q \).

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

\( \langle \Psi \rangle \)

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 \( T \) be the name set; assertions \( A \) be the finite sets of names, and conditions \( C \) 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

\[ R = \langle \{a\} \rangle \mid \langle \{b\} \rangle \mid a(\lambda x)x.P \mid \overline{b}.c.Q. \]

The process can behave as

\[ R \xrightarrow{\tau} \langle \{a\} \rangle \mid \langle \{b\} \rangle \mid P[x := c] \mid Q \]

due to the following facts. The shared assertion environment \( \Psi \) 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,

\[ R' = \langle \{a\} \rangle \mid a(\lambda x)x.P \mid \overline{b}.c.Q \xrightarrow{\tau} \]

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.
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 \( \tilde{x} \) and \( \tilde{N} \) 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 \((\rightarrow, \mathcal{A}, \mathcal{P})\) where \(\mathcal{A}\) is a set of actions (labels), \(\mathcal{P}\) is a set of processes (also called states), and \(\rightarrow \subseteq \mathcal{P} \times \mathcal{A} \times \mathcal{P}\) 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 \((\rightarrow, \mathcal{P})\) where \(\rightarrow \subseteq \mathcal{P} \times \mathcal{P}\). 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 \(0\):

\[
\begin{align*}
P | Q & \equiv Q | P \\
(P | Q) | R & \equiv (Q | P) | R \\
P | 0 & \equiv P
\end{align*}
\]

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

\[
(va)P | Q \equiv (va)(P | Q) \quad \text{if} \ a \notin \text{fn}(Q)
\]

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

\[
p_1 \xrightarrow{\alpha_1} p'_1 \cdots p_n \xrightarrow{\alpha_n} p'_n
\]

\[
p \xrightarrow{\alpha} p'
\]

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 \( \alpha_1 \) and continues as \( p'_1 \), and analogously for other process \( p_i \) for \( i = 2, \ldots, n \), then \( p \) performs an action \( \alpha \) 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)

\[
P, Q ::= 0 | a.P | \overline{a}.P | P | Q
\]

The rules for actions are simply the following:

\[
\frac{a.P \xrightarrow{a} P}{a.P \xrightarrow{a} P} \quad \frac{\overline{a}.P \xrightarrow{\overline{a}} P}{\overline{a}.P \xrightarrow{\overline{a}} P}
\]

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

\[
\frac{P \xrightarrow{\alpha} P'}{P | Q \xrightarrow{\alpha} P' | Q} \quad \frac{Q \xrightarrow{\overline{\alpha}} Q'}{P | Q \xrightarrow{\overline{\alpha}} P | Q'}
\]

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:

\[
\frac{P \xrightarrow{a} P' \quad Q \xrightarrow{\overline{a}} Q'}{P | Q \xrightarrow{\tau} P' | Q'} \quad \frac{P \xrightarrow{\overline{a}} P' \quad Q \xrightarrow{a} Q'}{P | Q \xrightarrow{\tau} P' | Q'}
\]

Note the rules are solely syntax-directed: they define the behaviour of a process by its structure. That is, \( P | Q \) transition with the action \( \tau \) to \( P' | Q' \) 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

\[
a.0 | (\overline{b}.0 | \overline{a}.0) \xrightarrow{\tau} 0 | (\overline{b}.0 | 0)
\]
where the first and the last parallel components synchronise:

\[
\begin{align*}
\text{a.0} & \xrightarrow{a} 0 \\
\text{b.0} & \mid \text{a.0} \xrightarrow{a} \text{b.0} \mid 0 \\
\text{a.0} & \mid (\text{b.0} \mid \text{a.0}) \xrightarrow{a} 0 \mid (\text{b.0} \mid 0)
\end{align*}
\]

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

![Tree Diagram]

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 →, which is a subset of \( P \times A \times P \) where \( P \) is a set of processes as defined by the grammar, and \( A \) 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 \( P \subseteq P \times A \times P \), one only needs to show that it satisfies the rules, and thus whenever we use the smallest relation → to derive processes the property \( P \) is implied since → ⊆ 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:

\[ a.P | \overline{a}.Q \rightarrow P | Q \]

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

\[
\frac{P \equiv P' \quad P' \rightarrow Q' \quad Q' \equiv Q}{P \rightarrow Q}
\]

\[
\frac{P \rightarrow P'}{P | Q \rightarrow P' | Q}
\]

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 \( a.0 | (\overline{b}.0 | \overline{a}.0) \) from the previous section:

\[
\begin{align*}
& a.0 | \overline{a}.0 \rightarrow 0 | 0 \\
& a.0 | (\overline{b}.0 | \overline{a}.0) \equiv (a.0 | \overline{a}.0) | \overline{b}.0, \quad (a.0 | \overline{a}.0) | \overline{b}.0 \rightarrow (0 | 0) | \overline{b}.0, \quad (0 | 0) | \overline{b}.0 \equiv 0 | (\overline{b}.0 | 0) \\
& a.0 | (\overline{b}.0 | \overline{a}.0) \rightarrow 0 | (\overline{b}.0 | 0)
\end{align*}
\]

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

\[
\frac{P \rightarrow P'}{(\nu a)P \rightarrow (\nu a)P'}
\]

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

\[
a(x).P | \overline{a}b.Q \rightarrow P\{b/x\} | Q
\]
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 $P \xrightarrow{a} P'$, then the process $Q$ must be able to repeat this action with a transition $Q \xleftarrow{a} Q'$ and furthermore $Q'$ must be able to continue repeating actions taken by $P'$ and so on.

Consider the following two transition systems:

$$
\begin{array}{c}
  P_1 \\
  \downarrow \text{a} \\
  P_2 \\
  \downarrow \text{b} \\
  P_4 \\
  \downarrow \text{b} \\
  P_4 \\
  \downarrow \text{b} \\
  P_5 \\
  \downarrow \text{b} \\
  P_5 \\
\end{array}
\quad
\begin{array}{c}
  Q_1 \\
  \downarrow \text{a} \\
  Q_2 \\
  \downarrow \text{b} \\
  Q_3 \\
\end{array}
$$

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

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

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

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

42
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 $P_1 \mathcal{R} Q_1$ 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 $\mathcal{R} \subseteq \sim$. So to prove that two processes are bisimilar, we need to find a bisimulation relation that includes those processes. Alternatively, bisimilarity may be defined as $P \sim Q$ if there is a bisimulation relation $\mathcal{R}$ such that $P \mathcal{R} Q$. 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 $P_1 = a.b.0 + a.b.0$, and the second as $Q_1 = a.b.0$. Thus,

$$a.b.0 + a.b.0 \sim a.b.0$$

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

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

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 $\tau$ 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 $\tau$ transitions.

The relation $\sim'$ is a weak bisimulation if $\sim'$ is symmetric and for all $P \sim' Q$:

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

Note in the above definition $\xrightarrow{\tau} \cdots \xrightarrow{\tau}$ may be an empty sequence of transitions. Thus, a process ignores the $\tau$ 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, $\tau.b.0 \approx b.0$.

---

$^6$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 \((\to, \mathcal{A}, \mathcal{P})\) (Section 2.2) where \(\to \subseteq \mathcal{P} \times \mathcal{A} \times \mathcal{P}\) is a transition relation, \(\mathcal{A}\), ranged over by \(\alpha\), is a set of actions, and \(\mathcal{P}\) is a set of processes. Then, the formulas of HML is defined by the following grammar

\[
A, B ::= \langle\alpha\rangle A \quad \text{may modality} \\
| [\alpha] A \quad \text{must modality} \\
| A \land B \quad \text{conjunction} \\
| \neg A \quad \text{negation} \\
| \text{true} \quad \text{truth constant}
\]

The meaning of formulas is given by the satisfaction relation

\[
P \models A
\]
that is defined by the following

\[ P \models (\alpha)A \quad \text{if exists } P' \text{ such that } P \xrightarrow{\alpha} P' \text{ and } P' \models A \]

\[ P \models [\alpha]A \quad \text{if for all } P' \text{ such that } P \xrightarrow{\alpha} P' \text{ then } P' \models A \]

\[ P \models A \land B \quad \text{if } P \models A \text{ and } P \models B \]

\[ P \models \neg A \quad \text{if it is not the case } P \models A \]

\[ P \models \text{true} \quad \text{is always true for any } P \]

The satisfaction relation asserts that the may modality \((\alpha)A\) requires the process \(P\) to have a transition with the action \(a\) 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 \(\alpha\). Formally, only one modality is necessary to be defined as they are inter-definable, i.e., \((\alpha)A = \neg [\alpha] \neg A\), and \([\alpha]A = \neg (\alpha) \neg A\). Other logical formulas are obtained in the standard way: \(\text{false} = \neg \text{true}\), \(A \land B = \neg (\neg A \land \neg B)\), and \(A \implies B = \neg A \lor B\).

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

\[ P \equiv Q \]

if for all formulas \(A\), \(P \models A\) if and only if \(Q \models A\).

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

\[ a \cdot b \cdot 0 \equiv a \cdot b \cdot 0 + a \cdot b \cdot 0 \]

as a formula \(A\) does not exist such that \(a \cdot b \cdot 0 \models A\) and \((a \cdot b \cdot 0 + a \cdot b \cdot 0) \not\models A\). However, the following are not logically equivalent

\[ a \cdot b \cdot 0 + a \cdot c \cdot 0 \not\equiv a \cdot (b \cdot 0 + c \cdot 0) \]

since we can find a distinguishing formula, namely \((a)((b)\text{true} \land (c)\text{true})\), such that

\[ a \cdot (b \cdot 0 + c \cdot 0) \models (a)((b)\text{true} \land (c)\text{true}) \]

but

\[ a \cdot b \cdot 0 + a \cdot c \cdot 0 \not\models (a)((b)\text{true} \land (c)\text{true}) \].

We say that a labelled transition system is image-finite if for all processes \(P\) and actions \(\alpha\) 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 \(P \equiv Q\). That is, the two relation coincide:

\[ \sim = \equiv \]

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

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

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:

$$(vs)(P | Q) | R$$

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 $s$:

$$(vs)(s(x).P' | \bar{s}m.Q') | R \rightarrow (vs)(P\{m/x\} | Q) | R$$

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

$$(vs)(\bar{s}m.P' | \bar{s}n.Q') | R.$$ 

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 $\text{int}.T$, written $s : \text{int}.T$. 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 $x : \text{int}$ and $P'$ is well-typed with respect to $T$. The partner process $\bar{s}m.Q'$ is well-typed with $\bar{\text{int}}.T'$ where $m : \text{int}$ 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 $\text{int}.T$ and $\bar{\text{int}}.T'$ 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 $\mathcal{L}$ be a set of labels, ranged over by $\ell$. 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)

\[
P, Q ::= a(s).P \mid \bar{a}(s).P \mid s(x).P \mid \bar{s}m.P \mid P \mid Q \mid \textbf{if } \varphi \textbf{ then } P \textbf{ else } Q \mid A \mid (vs)P \mid (va)P \mid 0 \mid s \oplus \ell.P \mid s \& \{\ell_1 : P_1, \ldots, \ell_n : P_n\}
\]

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

$$a(s).P \mid \bar{a}(s).Q \rightarrow (vs)(P \mid Q)$$
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 $\ell$ via session $s$. The process $A$ is just a process constant (Section 2.1.1).

Let $\mathcal{B}$ 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$^7$

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

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.

$$\text{dual}(\beta.T) = \bar{\beta}.\text{dual}(T)$$
$$\text{dual}(\bar{\beta}.T) = \beta.\text{dual}(T)$$
$$\text{dual}(\text{end}) = \text{end}$$
$$\text{dual}(\oplus \{ \ell_1 : T_1, \ldots, \ell_n : T_n \}) = \& \{ \ell_1 : \text{dual}(T_1), \ldots, \ell_n : \text{dual}(T_n) \}$$
$$\text{dual}(\& \{ \ell_1 : T_1, \ldots, \ell_n : T_n \}) = \oplus \{ \ell_1 : \text{dual}(T_1), \ldots, \ell_n : \text{dual}(T_n) \}$$
$$\text{dual}(\mu t.T) = \mu t.\text{dual}(T)$$
$$\text{dual}(t) = t$$

$^7$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 $s : \text{dual}(T)$, 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 $\Gamma$ is a list of type assignments to the shared channels, and $\Delta$ 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 $\Delta'$ 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 typesafety.

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 $\alpha$ 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. $\forall n \in \mathcal{N} . A(n)$ as $\land_{n \in \mathcal{N}} A(n)$ 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 $M(\lambda x_1, \ldots, x_n)X.P$ 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 $x_1, \ldots, x_n$. We do this by introducing the operation $\text{VARS}(X)$ to return a set of sets of pattern variables. The pattern matching is also parametrised with the operation $\text{MATCH}(M, (x_1, \ldots, x_n), X)$ to return a set of list of terms that are assigned to the pattern variables $x_1, \ldots, x_n$ by matching the pattern $X$ against the term $M$. The $\text{MATCH}$ 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, $a(\lambda m, k)\text{enc}(m, k)X.P$ is allowed in psi-calculi where the pattern $\text{enc}(m, k)$ 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 $\text{enc}(m, k)$, and thus the only allowed input form is $a(\lambda m)\text{enc}(m, k)P$ 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 $\text{SORT}$ 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 $\text{MATCH}$ 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
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.

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


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”.)