
FORMALISING THE π -CALCULUS USING NOMINAL LOGIC

JESPER BENGTON AND JOACHIM PARROW

Department of Information Technology, University of Uppsala, Sweden
e-mail address: {Jesper.Bengtson,Joachim.Parrow}@it.uu.se

ABSTRACT. We formalise the pi-calculus using the nominal datatype package, based on ideas from the nominal logic by Pitts et al., and demonstrate an implementation in Isabelle/HOL. The purpose is to derive powerful induction rules for the semantics in order to conduct machine checkable proofs, closely following the intuitive arguments found in manual proofs. In this way we have covered many of the standard theorems of bisimulation equivalence and congruence, both late and early, and both strong and weak in a uniform manner. We thus provide one of the most extensive formalisations of a process calculus ever done inside a theorem prover.

A significant gain in our formulation is that agents are identified up to alpha-equivalence, thereby greatly reducing the arguments about bound names. This is a normal strategy for manual proofs about the pi-calculus, but that kind of hand waving has previously been difficult to incorporate smoothly in an interactive theorem prover. We show how the nominal logic formalism and its support in Isabelle accomplishes this and thus significantly reduces the tedium of conducting completely formal proofs. This improves on previous work using weak higher order abstract syntax since we do not need extra assumptions to filter out exotic terms and can keep all arguments within a familiar first-order logic.

1. INTRODUCTION

1.1. Motivation. As the complexity of software systems increases, the need is growing to ensure their correct operation. One way forward is to create particular theories or frameworks geared towards particular application areas. These frameworks have the right kind of abstractions built in from the beginning, meaning that proofs can be conducted at a high level. The drawback is that different areas need different such frameworks, resulting in a proliferation and even abundance of theories. A prime example can be found in the field of process calculi. It originated in work by Milner in the late 1970s [27] and was intended to provide an abstract way to reason about parallel and communicating processes. Today there are many different strands of calculi addressing specific issues. Each of them embodies a certain kind of abstraction suitable for a particular area of application.

For each such calculus a certain amount of theoretical groundwork must be laid down. Typical examples include definitions of the semantics, establishing substitutive properties,

1998 ACM Subject Classification: F.4.1.

Key words and phrases: Pi-calculus, Theorem proving, Isabelle, Nominal logic.

structures for inductive proof strategies etc. This groundwork must naturally be correct beyond doubt (if there is an error in it then all proofs conducted in that calculus will be incorrect). The idea to use formal verification of the groundwork itself is therefore natural. In this paper we shall present an improved method to accomplish this.

1.2. Theorem provers. There exist today several proof assistants, aka theorem provers: Coq [11], Isabelle [32], Agda [1], PVS [34], Nuprl [16] and HOL [23], just to name a few. These theorem provers are interactive. They have many automated tactics, and the user can provide additional proof strategies. Many are also getting better and easier to use, and so the concept of having fully machine checked proofs has recently become far more realistic. As an indication of this several major results have been proven over the last few years, including the four and five colour theorems [7, 22], Kepler’s conjecture [33] and Gödel’s incompleteness theorem [41]. Significant advances in applications related to software are summarized in the POPLmark Challenge [6], a set of benchmarks intended both for measuring progress and for stimulating discussion and collaboration in mechanizing the metatheory of programming languages. There are for example results on analysis of typing in system F and light versions of Java. The theorem prover Isabelle is also currently used to verify software in the Verisoft project [3].

We want to emphasize that these types of tools are now being transferred to industry. In [12], a group at Microsoft Research in Cambridge compiles a subset of $F\sharp$ (a Microsoft product) code to the pi-calculus and security properties are checked using ProVerif [13]. This work was later extended in [9] where a cryptographic type checker was constructed for $F\sharp$ which handles a larger set of problems. The ideas from these are now being transferred into other Microsoft products. Also, the Spec \sharp [2] programming system is integrated in the Microsoft Visual Studio environment for the .NET platform and contains an automatic theorem prover.

1.3. The π -calculus. As the basic underlying model we have chosen the π -calculus, which since its conception in the late 1980s by Milner, Parrow and Walker [30] has had a significant impact on the way formal methods handle mobile systems. The mechanism of name-passing, in combination with the paradigm of static binding, where the scope of names may be dynamically extended by means of communication to include the receiver, has turned out to be surprisingly expressive for a vast variety of programming idioms: abstract data types, lambda-calculus, i.e. functional programming, object-oriented programming, imperative programming, logic and concurrent constraint programming, and primitives for encryption/decryption. The π -calculus has influenced the development of many high-level programming languages and it has triggered a whole family of related calculi. e.g. spi [5], join [17], fusion [36], blue [14], the applied π -calculus [4] and ambients [15]. In essence, the π -calculus has now grown out of a single formalism into a general field where components of formalisms, such as operators, semantics and proof methods, can be more freely combined.

1.4. Approach. The goal of our project is to provide a library in an automated theorem prover, Isabelle/HOL [32], which allows users to do machine checked proofs on the groundwork of process calculi. The guiding principle is that the proofs should correspond very closely to the traditional manual proofs present in the literature. This means that for a person who has completed these proofs manually very little extra effort should be required

in order to let Isabelle check them. Today those proofs are reasonably well understood, but capturing them in a theorem prover has until now been a daunting task. The reason is mainly related to bound names and the desire to abstract away from α -equivalence [6].

In the literature it is not uncommon to find statements such as: “henceforth we shall not distinguish between α -equivalent terms” or “we assume bound names to always be fresh”, even though it is left unsaid exactly what this means. In [40] Sangiorgi and Walker write:

In any discussion, we assume that the bound names of any processes or actions under consideration are chosen to be different from the names free in any other entities under consideration, such as processes, actions, substitutions and sets of names.

And in [35] we can find:

... we will use the phrase “ $bn(\alpha)$ is fresh” in a definition to mean that the name in $bn(\alpha)$, if any, is different from any free name occurring in any of the agents in the definition.

This kind of reasoning does not necessarily imply that proofs conducted in this manner are incorrect, only that they are not fully formalised.

Our approach is to formulate the π -calculus using ideas from nominal logic developed by Pitts et al. [37, 21, 42]. This is a first order logic designed to work with calculi using binders. It maintains all the properties of a first order logic and introduces an explicit notion of freshness of names in the terms. Gabbay’s thesis [18] uses it to introduce FM set theory, this is the standard ZF set theory but with an extra axiom for freshness of names. Recent work by Urban and Tasson [43] extends this work using ideas from [37] and solves the problem with freshness without introducing new axioms. The techniques have been implemented into the theorem prover Isabelle/HOL, in a nominal datatype package, so that when defining nominal datatypes, Isabelle will automatically generate a type which models the datatype up to α -equivalence as well as induction principles and a recursion combinator allowing the user to create functions on nominal datatypes.

1.5. Results. Our contribution is to use the nominal package in Isabelle to describe the π -calculus. We have proved substantial portions of [30] using these techniques. More specifically, we have proven that strong equivalence and weak congruence are congruence relations for both late and early operational semantics, that all structurally congruent terms are bisimilar and that late strong equivalence, weak bisimulation and weak congruence are included in their early counterparts. To our knowledge, properties about weak equivalences of the π -calculus have never before been formally derived inside a theorem prover. Our proof method is to lift the strong operational semantics to a weak one, enabling us to port our proofs between the two semantics. Moreover, our proofs follow their pen-and-paper equivalents very closely inside a first-order environment. In other words, the extra effort to have proofs checked by a machine is not prohibitive.

1.6. Exposition. In the next section we explain some basic concepts of the nominal datatype package. We do not give a full account of it, only enough that a reader may follow the rest of our paper. In Section 3 we cover the strong late operational semantics of the π -calculus as well as the induction and case analysis rules we have created for the semantic rules. Section 4 treats strong late bisimulation, the proofs that it is preserved by all operators except input prefix and that strong equivalence is a congruence. In Section 5 we show the proof

strategies for one of our main results in depth demonstrating how closely our formalised proofs map their pen-and-paper equivalents. Section 6 handles the structural congruence rules and the proof that all structurally congruent terms are also bisimilar. We cover the weak late operational semantics in Section 7 and prove that weak bisimulation is preserved by all operators except sum and input prefix and that weak congruence is a congruence. In Section 8 we formalise the early π -calculus, both strong and weak, and prove all the results which we have for the late semantics for early. We also prove that all late bisimulation relations are a subset of their corresponding early ones. In the concluding section we compare our efforts to related work and comment on planned further work. The Isabelle source files can be found at <http://www.it.uu.se/katalog/jesperb/pi>.

2. THE PI-CALCULUS IN ISABELLE

For a more thorough presentation of the nominal datatype package in Isabelle the reader is referred to [43], but enough basic definitions will be covered here for the reader to understand the rest of this paper. A *nominal datatype* definition is like an ordinary data type but it explicitly tags the binding occurrences of names. For example, a data type for λ -calculus terms would in this way tag the name in the abstraction. The point is that the nominal package in Isabelle automatically generates induction rules where α -equivalent terms are identified, thus saving the user much tedium in large proofs.

At the heart of nominal logic is the notion of *name swapping* where names are a countably infinite set of atomic terms. If T is any term of permutation type (a term which supports permutations of its names) and a and b are names then $(a\ b) \bullet T$ denotes the term where all instances of a in T become b and vice versa. All names (even the binding and bound occurrences) are swapped in this way. A *permutation* p is a finite sequence of swappings. If $p = (a_1\ b_1) \cdots (a_n\ b_n)$ then $p \bullet T$ means applying all swappings in p to T , beginning with the last element $(a_n\ b_n)$.

Permutations are mathematically well behaved. They very rarely change the properties of a term. Most importantly, α -equivalence is preserved by permutations. The property of being preserved by permutations is often called *equivariance*. We shall mainly use equivariance on binary relations, where the definition is:

Definition 2.1. Equivariance

$$\text{eqvt } \mathcal{R} \stackrel{\text{def}}{=} \forall p\ T\ U. (T, U) \in \mathcal{R} \implies (p \bullet T, p \bullet U) \in \mathcal{R}$$

Another key concept is the notion of *support*. The definition, in general, is that the support $\text{supp } T$ of a term T is the set of names which can affect T in permutations. In other words, if p is a permutation only involving names outside the support of T then $p \bullet T = T$. Remembering that α -equivalent terms are identified we see that the support corresponds to the *free names* in calculi like the λ -calculus.

A crucial property is that the support of a term is finite. This implies that for any term it is always possible to find a name outside its support. We say that a name a is *fresh* for a term T , written $a \# T$, if a is not in the support of T .

Permutations can be used to capture α -equivalence. Let $[x].T$ stand for any operator that binds x in T .

Proposition 2.2. $[x].T = [y].U \implies (x = y \wedge T = U) \vee (x \neq y \wedge x \# U \wedge T = (x\ y) \bullet U)$

If $[x].T = [y].U$ then either x and y are equal and T and U are α -equivalent or x is not equal to y and fresh in U and T is α -equivalent to U with all occurrences of x swapped with y and vice versa. Another way to capture α -equivalence is the following:

Proposition 2.3. $c \# (x, y, T, U) \wedge [x].T = [y].U \implies (x\ c) \bullet T = (y\ c) \bullet U$

Here and in the rest of the paper we use the word “proposition” for something that Isabelle generates automatically.

We use a version of the monadic π -calculus [30], and assume that the reader is familiar with the basic ideas of its syntax and semantics.

Definition 2.4. Defining the π -calculus in Isabelle.

Nominal declaration in Isabelle	Notation in this paper
nominal_datatype pi = PiNil	0
Tau pi	$\tau.P$
Input name "<<name>> pi"	$a(x).P$
Output name name pi	$\bar{a}b.P$
Match name name pi	$[a = b]P$
Mismatch name name pi	$[a \neq b]P$
Sum pi pi	$P + Q$
Par pi pi	$P \mid Q$
Res "<<name>> pi"	$(\nu x)P$
Bang pi	$!P$

This definition is an example of Isabelle notation, where $\ll name \gg pi$ indicates that $name$ is bound in pi . For the rest of the paper we shall use the traditional notation for π -calculus terms as specified in the previous definition.

The nominal datatype package automatically generates lemmas for reasoning about α -equivalence between processes – the ones generated from Prop. 2.2 can be found in the following proposition.

Proposition 2.5. *The most commonly used α -equivalence rules for the Input- and the Restriction case.*

$$\begin{aligned}
 \text{Input:} \quad & a(x).P = b(y).Q \implies a = b \wedge ((x = y \wedge P = Q) \vee \\
 & \quad \quad \quad (x \neq y \wedge x \# Q \wedge P = (x\ y) \bullet Q)) \\
 \text{Restriction:} \quad & (\nu x)P = (\nu y)Q \implies (x = y \wedge P = Q) \vee \\
 & \quad \quad \quad (x \neq y \wedge x \# Q \wedge P = (x\ y) \bullet Q)
 \end{aligned}$$

Most modern theorem provers automatically generate induction rules for defined datatypes. The nominal datatype package does the same for nominal datatypes but with one addition: bound names which occur in the inductive cases can be assumed to be disjoint from any finite set of names. This greatly reduces the amount of manual α -conversions.

Functions over nominal datatypes have one restriction – they may not depend on the bound names in their arguments. Since nominal types are equal up to α -equivalence two equal terms may have different bound names. When creating recursive functions over nominal datatypes in Isabelle, one has to prove that this property holds for all instantiations of the function. The nominal package provides the appropriate proof conditions.

Our only nominal function is substitution where $P\{a/b\}$ (which can be read P with a for b) is the agent obtained by replacing all free occurrences of b in P with a .

3. OPERATIONAL SEMANTICS

3.1. Definitions. We use the standard operational semantics [30]. Here transitions are of the form $P \xrightarrow{\alpha} P'$, where α is an action. A first attempt, which works well for simpler calculi like CCS, is to inductively define a set of tuples containing three elements: a process P , an action α and the α -derivative of P [8].

However, in the π -calculus the action α may bind a name, and the scope of this binding extends into P' . This observation is made already in the original presentation of the π -calculus [30] where lemmas concerning variants of transitions are spelled out. In his tutorial on the polyadic pi-calculus [28] Milner uses "commitments" rather than labelled transitions. A transition here corresponds to a pair consisting of an agent and a commitment where the latter may have binders and contains both the action and derivative process. We thus face a discrepancy between a more traditional syntax for transitions (looking like tuples of three elements) and the intended semantics (that action and derivative in reality is one construct with names that can be bound in all of it). In many presentations of the π -calculus this issue is glossed over, and if α -conversions are not defined rigorously the three-element syntax for transitions works fine. But here it poses a problem — it would require us to explicitly state the rules for changing the bound variable, and we would not be able to rely on the otherwise smooth treatment of α -variants in our framework. Therefore, in our implementation we follow [28], with a slight change of notation to avoid confusion of prefixes and commitments, and define a *residual*-datatype which contains both action and derivative. It binds the bound names of an action also in the derivative. (A similar technique is also used by Gabbay when formalising the π -calculus in FM set theory [20].)

Definition 3.1. The residual datatype

```
datatype subject = Input name
                | BoundOutput name
```

```
datatype freeRes = Output name name
                | Tau
```

```
nominal_datatype residual = BoundResidual subject "<<name>> pi"
                        | FreeResidual freeRes pi
```

In this paper we shall continue to write pairs of processes and residuals as transitions in the familiar way, and we need to distinguish between actions that bind names and those that do not. We introduce the following notation.

Definition 3.2.

- (i) $P \xrightarrow{a\langle\langle x \rangle\rangle} P'$ denotes a transition with the bound name x in the action. Note that a is of type **subject**. The residual by itself is written $a\langle\langle x \rangle\rangle \prec P'$.
- (ii) $P \xrightarrow{\alpha} P'$ denotes a transition without bound names. Note that α is of type **freeRes**. The residual by itself is written $\alpha \prec P'$.
- (iii) A transition can also be written as $P \mapsto Res$ where P is an agent and Res is a residual, for example $\tau.P \mapsto \tau \prec P$

$a(x).P \xrightarrow{a(x)} P$ Input	$\bar{a}b.P \xrightarrow{\bar{a}b} .P$ Output	$\tau.P \xrightarrow{\tau} P$ Tau
$\frac{P \mapsto Res}{[a = a]P \mapsto Res}$ Match	$\frac{P \mapsto Res \quad a \neq b}{[a \neq b]P \mapsto Res}$ Mismatch	
$\frac{P \xrightarrow{\bar{a}b} P' \quad a \neq b}{(\nu b)P \xrightarrow{\bar{a}(b)} P'}$ Open	$\frac{P \mapsto Res}{P + Q \mapsto Res}$ Sum	$\frac{P \xrightarrow{a\langle\langle x \rangle\rangle} P' \quad x \# Q}{P \mid Q \xrightarrow{a\langle\langle x \rangle\rangle} P' \mid Q}$ ParB
$\frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q}$ ParF	$\frac{P \xrightarrow{a(x)} P' \quad Q \xrightarrow{\bar{a}b} Q'}{P \mid Q \xrightarrow{\tau} P'\{b/x\} \mid Q'}$ Comm	
$\frac{P \xrightarrow{a(x)} P' \quad Q \xrightarrow{\bar{a}(y)} Q' \quad y \# P}{P \mid Q \xrightarrow{\tau} (\nu y)(P'\{y/x\} \mid Q')}$ Close	$\frac{P \xrightarrow{a\langle\langle x \rangle\rangle} P' \quad y \# (a, x)}{(\nu y)P \xrightarrow{a\langle\langle x \rangle\rangle} (\nu y)P'}$ ResB	
$\frac{P \xrightarrow{\alpha} P' \quad y \# \alpha}{(\nu y)P \xrightarrow{\alpha} (\nu y)P'}$ ResF	$\frac{P \mid !P \xrightarrow{\alpha} P'}{!P \xrightarrow{\alpha} P'}$ Replication	

Figure 1: The *Par*- and the *Res*-rule in the operational semantics of the π -calculus have been split. Symmetric versions have been elided.

As previously mentioned, functions over nominal datatypes cannot depend on bound names. This poses a slight problem, since traditionally some of the operational rules have conditions on the bound names. An example of this is the **Par** rule in the standard operational semantics which states that the transition $P \mid Q \xrightarrow{\alpha} P' \mid Q$ can occur only if $P \xrightarrow{\alpha} P'$ and $\text{bn}(\alpha) \cap \text{fn}(Q) = \emptyset$. A function such as bn does not exist in nominal logic and thus cannot be created using the nominal datatype package. An easy solution is to split the operational rules which have these types of conditions into two rules — one for the transitions with bound names, and one for the ones without. Doing this does not create extra proof obligations as most proofs have to consider bound and free transitions separately anyway. We can now define our operational semantics using inductively defined sets which will contain pairs of processes and residuals. The Semantics, including the split rules for **Par** and **Res** can be found in Fig. 1.

As mentioned previously, permutations are usually very well behaved. The following proposition is generated automatically by the nominal package.

Proposition 3.3. $P \mapsto Res \implies p \bullet P \mapsto p \bullet Res$

3.2. Induction and case analysis rules.

3.2.1. *Automatically generated rules.* Isabelle will automatically create rules for both induction and case analysis of the semantics. They are specifically tailored to allow induction over all possible transitions but can also be custom made to do induction or case analysis over specific types of processes, such as those composed by the $|$ -operator. They will have an assumption of the form $P \mapsto Res$, which is the term with which we are working, and a logical proposition $Prop$ which is what we want to prove. When applied these rules will generate a set of subgoals where every subgoal corresponds to one action that the process P could take to end up in Res – in short, $Prop$ needs to be proven for all possible transitions for the rule to hold. The rules do, however, assume that the equivalence relation used is syntactic equivalence and not α -equivalence. The nominal datatype package automatically creates induction rules for nominal datatypes as well as for inductively defined sets or predicates. The induction rule generated for the semantics is the largest possible one which does induction over all operational rules and there is currently no way to automatically generate case analysis rules for transitions of a certain form. To derive rules for the cases which do not make use of bound names is unproblematic. In fact, Isabelle will be able to derive the following case analysis rules with very little help.

Proposition 3.4. *The automatically generated case analysis rule for tau-transitions.*

$$\frac{\tau.P \xrightarrow{\alpha} P' \quad \alpha = \tau \wedge P = P' \implies Prop}{Prop}$$

Proposition 3.5. *The automatically generated case analysis rule for output transitions.*

$$\frac{\bar{a}b.P \xrightarrow{\alpha} P' \quad \alpha = \bar{a}b \wedge P = P' \implies Prop}{Prop}$$

Proposition 3.6. *The automatically generated case analysis rule for matches.*

$$\frac{[a = b]P \mapsto Res \quad a = b \wedge P \mapsto Res \implies Prop}{Prop}$$

Proposition 3.7. *The automatically generated case analysis rule for mismatches.*

$$\frac{[a \neq b]P \mapsto Res \quad a \neq b \wedge P \mapsto Res \implies Prop}{Prop}$$

Proposition 3.8. *The automatically generated case analysis rules for sums.*

$$\frac{P + Q \mapsto Res \quad P \mapsto Res \implies Prop \quad Q \mapsto Res \implies Prop}{Prop}$$

The rest of the rules generated by Isabelle for our operational semantics deal with bound names and suffer from three problems, which we now address in turn.

3.2.2. *Problems with generated bound names.* The first problem is that some semantic case analysis rules generate bound names. When the rule is applied in the context of a proof, there is no a priori guarantee that these names are fresh in this larger context. We therefore derive rules for induction and case analysis which are parameterized on a finite set of names, the “context names”, which the user can provide when applying the rule. The bound names generated by the rules are guaranteed to be fresh from the context names (just as is guaranteed for induction rules generated by the nominal package, and for the same reason: avoiding name clashes and α -conversions later in the proof). This idea stems from [43] but was developed independently of similar work in [45]. The logical framework has also been covered in [38].

As an example a derived rule for case analysis of the parallel operator is shown in the following proposition where the parameter \mathcal{C} represents a set of context names and can be instantiated with any nominal datatype:

Lemma 3.9. *The derived case analysis rule for the parallel operator with no bound names in the transition.*

$$\begin{array}{c}
 P \mid Q \xrightarrow{\alpha} R \\
 \forall P'. P \xrightarrow{\alpha} P' \wedge R = P' \mid Q \implies Prop \\
 \forall Q'. Q \xrightarrow{\alpha} Q' \wedge R = P \mid Q' \implies Prop \\
 \forall P' Q' a x b. P \xrightarrow{a(x)} P' \wedge Q \xrightarrow{\bar{a}b} Q' \wedge \alpha = \tau \wedge \\
 \quad R = P' \{b/x\} \mid Q' \wedge x \# \mathcal{C} \implies Prop \\
 \forall P' Q' a x b. P \xrightarrow{\bar{a}b} P' \wedge Q \xrightarrow{a(x)} Q' \wedge \alpha = \tau \wedge \\
 \quad R = P' \mid Q' \{b/x\} \wedge x \# \mathcal{C} \implies Prop \\
 \forall P' Q' a x y. P \xrightarrow{a(x)} P' \wedge Q \xrightarrow{\bar{a}(y)} Q' \wedge y \# P \wedge \alpha = \tau \wedge \\
 \quad R = (\nu y)(P' \{y/x\} \mid Q') \wedge x \# \mathcal{C} \wedge y \# \mathcal{C} \implies Prop \\
 \forall P' Q' a x y. P \xrightarrow{\bar{a}(y)} P' \wedge Q \xrightarrow{a(x)} Q' \wedge y \# Q \wedge \alpha = \tau \wedge \\
 \quad R = (\nu y)(P' \mid Q' \{y/x\}) \wedge x \# \mathcal{C} \wedge y \# \mathcal{C} \implies Prop \\
 \hline
 Prop
 \end{array}$$

Each all-quantified term corresponds to a possible transition by the process $P \mid Q$. The two semantic rules which introduce bound names are the *Comm*- and the *Close* rules. The rule can be instantiated with an arbitrary term \mathcal{C} and these bound names will be set fresh for that term.

3.2.3. *Problems with equivalence checks on terms.* The second problem is that in case analysis, equivalence checks between terms always appear. If these terms contain bound names, such as $(\nu x)P = (\nu y)Q$, then normal unification is not possible. As seen in Prop. 2.2 and 2.3, every such equivalence check produces either two cases which both have to be proven or one case with several permutation and freshness conditions. As an example, a rule for case analysis on the ν -operator with no bound names in the action can be found in the following proposition:

Proposition 3.10. *The automatically generated case analysis rule for the ν -operator, based on Prop. 2.2, where no bound name occurs in the action.*

$$\frac{\forall Q Q' \beta y. Q \xrightarrow{\beta} Q' \wedge y \# \beta \wedge (\nu x)P = (\nu y)Q \wedge \alpha = \beta \wedge P' = (\nu y)Q' \implies Prop}{(\nu x) P \xrightarrow{\alpha} P' \implies Prop} Prop$$

The conjunct $(\nu x)P = (\nu y)Q$ poses a problem as we have to show *Prop* for all cases such that the equivalence holds. We can reason about this equality using either Prop. 2.2 or Prop. 2.3 but neither of these rules are convenient to work with. Prop. 2.2 causes a case explosion which forces us to prove the same thing several times for different permutations on terms and Prop. 2.3 introduces extra permutations which makes the proof more cumbersome to work with. We therefore use the following derived lemma in place of the original case analysis rule:

Lemma 3.11. *Case analysis rule derived from Prop. 3.10.*

$$\frac{\forall P''. P \xrightarrow{\alpha} P'' \wedge x \# \alpha \wedge P' = (\nu x)P'' \implies Prop}{(\nu x) P \xrightarrow{\alpha} P' \implies Prop} Prop$$

The main idea of the proof is to find a P'' which suitably depends on the universally quantified terms in the second assumption of the original proposition.

The other rule which require this treatment is the case analysis rule for the parallel operator where the transition contains a bound name.

Lemma 3.12. *Case analysis rule for the parallel operator with a bound name in the transition.*

$$\frac{\forall P'. P \xrightarrow{a \ll x \gg} P' \wedge x \# Q \wedge R = P' \mid Q \implies Prop \quad \forall Q'. Q \xrightarrow{a \ll x \gg} Q' \wedge x \# P \wedge R = P \mid Q' \implies Prop}{P \mid Q \xrightarrow{a \ll x \gg} R \implies Prop} Prop$$

3.2.4. Problems with multiple bound names in terms. The third problem arises when several bound names occur in the term that you want to do case analysis on. We have already shown how we can ensure that any newly generated bound names are disjoint from any context we might be interested in. The problem here is that since multiple bound names are present before case analysis starts, any properties regarding them are fixed in the environment and if we have a name clash, we have to do manual α -conversions. There are two rules that suffer from this problem, where the simplest one is the one for input-prefix. To solve this problem, we derive the following case analysis rule.

Lemma 3.13. *The derived case analysis rule for the input-prefix.*

$$\frac{a(x).P \xrightarrow{b(y)} P' \quad a = b \wedge P' = (x y) \bullet P \implies Prop}{a(x).P \xrightarrow{b(y)} P' \implies Prop} Prop$$

The other rule which requires this treatment is the restriction case where a bound name appears in the transition.

Lemma 3.14. *The derived case analysis rule for restriction with a bound name in the transition.*

$$\frac{\begin{array}{c} (\nu x)P \xrightarrow{a \ll y \gg} P' \\ x \neq y \\ \forall b P''. P \xrightarrow{\bar{b}x} P'' \wedge b \neq x \wedge a = \text{BoundOutput } b \wedge P' = (x y) \bullet P'' \implies \text{Prop} \\ \forall P''. P \xrightarrow{a \ll y \gg} P'' \wedge x \# a \wedge x \neq y \wedge P' = (\nu x)P'' \implies \text{Prop} \end{array}}{\text{Prop}}$$

In this rule we require x and y to be disjoint. The two applicable rules from the semantics have conflicting requirements on the bound names – one requires them to be the same, and the other requires them to be disjoint. To keep the generality of the lemma, we keep the bound names disjoint and in the **Open** case permute the names in the derivative. As we shall see later, we will always be in a context where we can guarantee that x and y are separate when applying this rule.

3.2.5. *Induction.* The remaining operator is the $!$ -operator which requires an induction rule rather than a case analysis rule as it is the only operator which occurs in the premise of its inference rule, as can be seen in Fig. 1. As in Lemma 3.9, \mathcal{C} is a parameter representing the names with which new bound names may not clash.

Lemma 3.15. *The derived induction rule for the $!$ -operator.*

$$\frac{\begin{array}{l} !P \mapsto \text{Res} \\ (1) \forall a x P' \mathcal{C}. P \xrightarrow{a \ll x \gg} P' \wedge x \# P \wedge x \# \mathcal{C} \implies \text{Prop } \mathcal{C} (P \mid !P) (a \ll x \gg \prec P' \mid !P) \\ (2) \forall \alpha P' \mathcal{C}. P \xrightarrow{\alpha} P' \implies \text{Prop } \mathcal{C} (P \mid !P) (\alpha \prec P' \mid !P) \\ (3) \forall a x P' \mathcal{C}. !P \xrightarrow{a \ll x \gg} P' \wedge x \# P \wedge x \# \mathcal{C} \wedge \text{Prop } \mathcal{C} (!P) (a \ll x \gg \prec P') \implies \\ \text{Prop } \mathcal{C} (P \mid !P) (a \ll x \gg \prec P \mid P') \\ (4) \forall \alpha P' \mathcal{C}. !P \xrightarrow{\alpha} P' \wedge \text{Prop } \mathcal{C} (!P) (\alpha \prec P') \implies \text{Prop } \mathcal{C} (P \mid !P) (\alpha \prec P \mid P'') \\ (5) \forall a x b P' P'' \mathcal{C}. P \xrightarrow{a(x)} P' \wedge !P \xrightarrow{\bar{a}b} P'' \wedge \text{Prop } \mathcal{C} (!P) (\bar{a}b \prec P'') \wedge x \# \mathcal{C} \implies \\ \text{Prop } \mathcal{C} (P \mid !P) (\tau \prec P' \{b/x\} \mid P'') \\ (6) \forall a x b P' P'' \mathcal{C}. P \xrightarrow{\bar{a}b} P' \wedge !P \xrightarrow{a(x)} P'' \wedge \text{Prop } \mathcal{C} (!P) (a(x) \prec P'') \wedge x \# \mathcal{C} \implies \\ \text{Prop } \mathcal{C} (P \mid !P) (\tau \prec P' \mid P'' \{b/x\}) \\ (7) \forall a x y P' P'' \mathcal{C}. P \xrightarrow{a(x)} P' \wedge !P \xrightarrow{\bar{a}(y)} P'' \wedge \text{Prop } \mathcal{C} (!P) (\bar{a}(y) \prec P'') \wedge x \# \mathcal{C} \wedge \\ y \# P \wedge y \# \mathcal{C} \implies \text{Prop } \mathcal{C} (P \mid !P) (\tau \prec (\nu y)(P' \{y/x\} \mid P'')) \\ (8) \forall a x y P' P'' \mathcal{C}. P \xrightarrow{\bar{a}(y)} P' \wedge !P \xrightarrow{a(x)} P'' \wedge \text{Prop } \mathcal{C} (!P) (a(x) \prec P'') \wedge x \# \mathcal{C} \wedge \\ y \# P \wedge y \# \mathcal{C} \implies \text{Prop } \mathcal{C} (P \mid !P) (\tau \prec (\nu y)(P' \mid P'' \{y/x\})) \end{array}}{\text{Prop } \mathcal{C} (!P) \text{ Res}}$$

Each numbered line corresponds to one way that an action can be inferred from a replication. Line (1) and (2) cover the case where a single process makes an action, line (3) and (4) perform the inductive step where a process in the smaller chain of replicated processes makes an action. Line (5) and (6) handle communication and line (7) and (8) handle scope extrusion.

The derived lemma is an induction rule in that it has the induction hypothesis *Prop* occurring on the left hand side of the implications in the inductive rules where $!$ occurs. A simpler rule which only makes use of the inference rule for $!$ is available, but the proofs we

are interested in would have to make use of the rules for the $|$ -operator to reason about all possible transitions that a process of the form $!P$ could do. This induction rule combines the two in one rule.

4. STRONG BISIMULATION

4.1. Simulation. Intuitively, two processes are said to be bisimilar if they can mimic each other step by step. Traditionally, a bisimulation is a symmetric binary relation \mathcal{R} such that for all processes P and Q in \mathcal{R} , if P can do an action, then Q can mimic that action and their corresponding derivatives are in \mathcal{R} .

When defining bisimulation between two processes in the π -calculus, extra care has to be taken with respect to bound names in actions. Consider the following processes:

$$\begin{aligned} P &\stackrel{\text{def}}{=} a(u).(vb)\bar{b}x.0 \\ Q &\stackrel{\text{def}}{=} a(x).0 \end{aligned}$$

Clearly P and Q should be bisimilar since they both can do only one input action along a channel a and then nothing more. But since x occurs free in P , P cannot be α -converted into $a(x).(vb)\bar{b}x$. However, since processes have finite support, there exists a name w which is fresh in both P and Q and after α -converting both processes, bisimulation is possible. Hence, when reasoning about bisimulation, we must restrict attention to the bound names of actions which are fresh for both P and Q . One of our main contributions is how this is achieved without running into a multitude of α -conversions. Our formal definition of bisimulation equivalence uses the following notion, where \mathcal{R} is a binary relation on agents.

Definition 4.1. The agent P can simulate the agent Q preserving \mathcal{R} , written $P \rightsquigarrow_{\mathcal{R}} Q$, if

$$\begin{aligned} (\forall a \ x \ Q'. \quad & Q \xrightarrow{a \ll x \gg} Q' \wedge x \# P \implies \\ & \exists P'. P \xrightarrow{a \ll x \gg} P' \wedge \text{derivative}(a, x, P', Q', \mathcal{R})) \wedge \\ (\forall \alpha \ Q'. \quad & Q \xrightarrow{\alpha} Q' \implies \exists P'. P \xrightarrow{\alpha} P' \wedge (P', Q') \in \mathcal{R}) \end{aligned}$$

where

$$\begin{aligned} \text{derivative}(a, x, P', Q', \mathcal{R}) &\stackrel{\text{def}}{=} \\ \text{case } a \text{ of } & \text{Input } _ \implies \forall u. (P'\{u/x\}, Q'\{u/x\}) \in \mathcal{R} \\ & | \text{BoundOutput } _ \implies (P', Q') \in \mathcal{R} \end{aligned}$$

Note that the argument a in derivative is of type **subject** as described in Def. 3.1. Thus, the requirement is that if Q has an action then P has the same action, and the derivatives P' and Q' are in \mathcal{R} .

Equivariance also needs to be established for simulations. More specifically, we need to prove the following lemma:

Lemma 4.2. *If $P \rightsquigarrow_{\mathcal{R}} Q$, \mathcal{R} is a subset of \mathcal{R}' and \mathcal{R}' is equivariant then $p \bullet P \rightsquigarrow_{\mathcal{R}'} p \bullet Q$.*

Proof. By Def. 4.1. The intuition is to apply the inverse permutation of p to cancel it out. The inverse can be applied using Lemma 3.3 and the assumption $\text{eqvt } \mathcal{R}'$. \square

The traditional way to define strong bisimulation equivalence is to say that \mathcal{R} is a *bisimulation* if it is symmetric and that for all agents P, Q it holds that $(P, Q) \in \mathcal{R} \rightarrow P \rightsquigarrow_{\mathcal{R}} Q$; the strong bisimulation equivalence is then the union of all strong bisimulations. As we shall see in a moment, an alternative definition using direct coinduction, similar to the approach in [25], yields shorter proofs. Our main improvement, however, is in the treatment of the bound name x . In Def. 4.1 it is by definition ensured not to be among the free names in P , but when we use it within a complex proof we will run into a massive case analysis on whether x is equal to other names used in the proof. In the same way as in Lemma 3.9 we bypass this tedium and derive the following introduction rule for an arbitrary nominal data term \mathcal{C} . This term is provided by the user to ensure that the bound name is distinct from any name occurring so far in the proof.

Lemma 4.3. *An introduction rule for simulation avoiding name clashes.*

$$\frac{\begin{array}{c} \text{eqvt } \mathcal{R} \\ \forall a \ x \ Q'. \ Q \xrightarrow{a \ll x \gg} Q' \wedge x \# \mathcal{C} \implies \\ \exists P'. \ P \xrightarrow{a \ll x \gg} P' \wedge \text{derivative}(a, \ x, \ P', \ Q', \ \mathcal{R}) \\ \forall \alpha \ Q'. \ Q \xrightarrow{\alpha} Q' \implies \exists P'. \ P \xrightarrow{\alpha} P' \wedge (P', Q') \in \mathcal{R} \end{array}}{P \rightsquigarrow_{\mathcal{R}} Q}$$

This is used extensively in our proofs. We can in this way make sure that whenever bound names appear in our proof context, these bound names do not clash with other names which would force us to do α -conversions. The amount of α -conversions we have to do manually is reduced to the instances where they would be required in a manual proof.

Note that we need an extra requirement that our simulation relation is equivariant. The reason is that if the relation is not closed under permutations, we cannot α -convert our processes. Fortunately, all relations of interest turn out to be equivariant and the proofs trivial.

4.2. Preservation properties. Our simulations are parametrised on an arbitrary relation \mathcal{R} . We exploit this by providing, for each operator, a set of constraints on \mathcal{R} such that the operator preserves $\rightsquigarrow_{\mathcal{R}}$. This set of constraints should be kept as small as possible as they will have to be proven when we prove preservation properties of bisimulation. In this section we show all proofs that are needed to show that a relation is preserved by all operators.

We first establish lemmas for reflexivity and transitivity.

Lemma 4.4. $\text{Id} \subseteq \mathcal{R} \implies P \rightsquigarrow_{\mathcal{R}} P$

Proof. By the definition of simulation. □

Lemma 4.5. $P \rightsquigarrow_{\mathcal{R}} Q \wedge Q \rightsquigarrow_{\mathcal{R}'} R \wedge \text{eqvt } \mathcal{R}'' \wedge \mathcal{R}' \circ \mathcal{R} \subseteq \mathcal{R}'' \implies P \rightsquigarrow_{\mathcal{R}''} R$

Proof. By Lemma 4.3 and setting \mathcal{C} to (P, Q) to make the bound names which occur in the transitions disjoint from P and Q . We would otherwise have to do manual α -conversions when traversing the simulation chain. □

We can now move on to our preservation lemmas.

Lemma 4.6. $(P, Q) \in \mathcal{R} \implies \tau.P \rightsquigarrow_{\mathcal{R}} \tau.Q$

Proof. By Def. 4.1 and Prop. 3.4. □

Lemma 4.7. $(P, Q) \in \mathcal{R} \implies \bar{a}b.P \rightsquigarrow_{\mathcal{R}} \bar{a}b.Q$

Proof. By Def. 4.1 and Prop. 3.5. □

In order for a relation to be preserved by the input prefix it needs to be closed under substitutions. We write \mathcal{R}^s for the closure of the relation \mathcal{R} under all substitutions.

Definition 4.8. $P \mathcal{R}^s Q \stackrel{\text{def}}{=} \forall \sigma. (P\sigma) \mathcal{R} (Q\sigma)$ where σ is a chain of substitutions.

Lemma 4.9. $(P, Q) \in \mathcal{R}^s \wedge \text{eqvt } \mathcal{R} \implies a(x).P \rightsquigarrow_{\mathcal{R}} a(x).Q$

Proof. By Lemma 4.3 and setting \mathcal{C} to (x, P) . Lemma 3.13 can then be used to finish the proof. □

Lemma 4.10.

$$\frac{P \rightsquigarrow_{\mathcal{R}} Q}{[a = b]P \rightsquigarrow_{\mathcal{R}} [a = b]Q}$$

Proof. By Def. 4.1 and Prop. 3.6. □

Lemma 4.11.

$$\frac{P \rightsquigarrow_{\mathcal{R}} Q}{[a \neq b]P \rightsquigarrow_{\mathcal{R}} [a \neq b]Q}$$

Proof. By Def. 4.1 and Prop. 3.7. □

Lemma 4.12.

$$\frac{\frac{P \rightsquigarrow_{\mathcal{R}} Q}{\text{Id} \subseteq \mathcal{R}}}{P + S \rightsquigarrow_{\mathcal{R}} Q + S}$$

Proof. By Def. 4.1, Prop. 3.8 and Lemma 4.4. □

The remaining preservation lemmas do not require that the relation reasoned about in the assumptions are the same as in the conclusions. It suffices to require them to be related by a set of constraints. The reason for this will be clarified when we cover bisimulation, suffice here to say that it makes the lemmas more general.

Lemma 4.13.

$$\frac{\begin{array}{l} P \rightsquigarrow_{\mathcal{R}} Q \quad (P, Q) \in \mathcal{R} \quad \text{Id} \subseteq \mathcal{R} \\ \forall P Q S. (P, Q) \in \mathcal{R} \implies (P \mid S, Q \mid S) \in \mathcal{R}' \\ \forall P Q x. (P, Q) \in \mathcal{R}' \implies ((\nu x)P, (\nu x)Q) \in \mathcal{R}' \end{array}}{P \mid S \rightsquigarrow_{\mathcal{R}'} Q \mid S}$$

Proof. By the definition of \rightsquigarrow . Lemma 3.12 is used to prove the cases where bound names occur in the transition and Lemma 3.9 is used otherwise. When using Lemma 3.9 \mathcal{C} is set to (P, S) . This proof will be covered more extensively in Section 5. □

Lemma 4.14.

$$\frac{P \rightsquigarrow_{\mathcal{R}} Q \quad \text{eqvt } \mathcal{R} \quad \text{eqvt } \mathcal{R}' \quad \mathcal{R} \subseteq \mathcal{R}' \quad \forall P Q x. (P, Q) \in \mathcal{R} \implies ((\nu x)P, (\nu x)Q) \in \mathcal{R}'}{(\nu x)P \rightsquigarrow_{\mathcal{R}'} (\nu x)Q}$$

Proof. By Lemma 4.3 and setting \mathcal{C} to (x, P) . Lemma 3.14 and 3.10 are then used for the case analysis. \mathcal{R} has to be equivariant since the **Open** case introduces permutations which need to be applied to the relation. \square

The remaining preservation lemma we need is for the $!$ -operator. For this proof we are going to need a recursively defined relation. This follows from the fact that the $!$ -operator is the only operator which occurs on the left hand side of the semantic rules and the proof needs to be done on the depth of inference and not the size of the term.

Definition 4.15.

$$\begin{aligned} \text{Rep } \mathcal{R} &\stackrel{\text{def}}{=} (P, Q) \in \mathcal{R} \implies (!P, !Q) \in \text{Rep } \mathcal{R} \\ &(P, Q) \in \mathcal{R} \wedge (S, T) \in \text{Rep } \mathcal{R} \implies (P \mid S, Q \mid T) \in \text{Rep } \mathcal{R} \\ &(P, Q) \in \text{Rep } \mathcal{R} \implies ((\nu x)P, (\nu x)Q) \in \text{Rep } \mathcal{R} \end{aligned}$$

Lemma 4.16.

$$\frac{(P, Q) \in \mathcal{R} \quad \text{eqvt } \mathcal{R} \quad \forall P Q. (P, Q) \in \mathcal{R} \implies P \rightsquigarrow_{\mathcal{R}} Q}{!P \rightsquigarrow_{\text{Rep } \mathcal{R}} !Q}$$

Proof. The trick here is to include the fact that $(!P, !Q) \in \text{Rep } \mathcal{R}$ in the induction hypothesis. We use Lemma 3.15 to do induction over the transitions made by the process $!Q$. We know that the processes in the relation \mathcal{R} simulate each other and the induction hypothesis generates the simulations by the nested replications. This proof is the most extensive of the preservation proofs due to its many cases and the need for induction. \square

4.3. Strong Bisimulation. Strong bisimulation equivalence can be described using coinduction, i.e. the greatest fixed point derived from a monotonic function.

Definition 4.17. Strong bisimulation equivalence, \sim , is the largest relation satisfying:

$$P \sim Q \implies P \rightsquigarrow_{\sim} Q \wedge Q \rightsquigarrow_{\sim} P$$

Note that we do not need to define what a bisimulation is; our coinductive definition uses $P \rightsquigarrow_{\mathcal{R}} Q$ directly. This defines \sim to be the largest relation such that related agents can simulate each other preserving \sim .

Conducting proofs on bisimulation equivalence often boils down to proving the same thing twice – once for each direction. With our formulation it is often easy to just prove one direction and let the other be inferred automatically.

When proving that two processes are bisimilar, we pick a set \mathcal{X} which contains the processes and which respects the constraints of the corresponding preservation lemma. It then suffices to show that all members of \mathcal{X} are simulated preserving $\mathcal{X} \cup \sim$. The following coinduction rules are easily derivable from the ones generated by Isabelle.

Lemma 4.18.

$$\frac{(P, Q) \in \mathcal{X} \quad \forall P' Q'. (P', Q') \in \mathcal{X} \implies P' \rightsquigarrow_{\mathcal{X} \cup \sim} Q' \wedge Q' \rightsquigarrow_{\mathcal{X} \cup \sim} P'}{P \sim Q}$$

Lemma 4.19.

$$\frac{(P, Q) \in \mathcal{X} \quad \forall P' Q'. (P', Q') \in \mathcal{X} \implies P' \rightsquigarrow_{\mathcal{X}} Q' \wedge Q' \rightsquigarrow_{\mathcal{X}} P'}{P \sim Q}$$

The difference between the two rules is found in the goal where the weaker version requires the processes to be simulated preserving $\mathcal{X} \cup \sim$ whereas the stronger version only requires them to be simulated preserving \mathcal{X} . Unless otherwise specified, the first of the two is the one being used.

The coinductive definition of bisimulation is equal to the standard one where bisimulation is regarded as the union of all bisimulation relations.

Definition 4.20. A relation \mathcal{R} is a bisimulation relation if for all $(P, Q) \in \mathcal{R}$, $P \rightsquigarrow_{\mathcal{R}} Q$ and $Q \rightsquigarrow_{\mathcal{R}} P$. We define \sim' to be the union of all bisimulation relations.

We find the coinductive approach easier to work with and the proof that the two versions of bisimilarity are equal is straightforward.

Lemma 4.21. $\sim = \sim'$

Proof.

\Rightarrow By definition of \sim we get for all processes P and Q where $P \sim Q$ that $P \rightsquigarrow_{\sim} Q$ and $Q \rightsquigarrow_{\sim} P$. Hence \sim is a bisimulation relation.

\Leftarrow From the definition of \sim' we get an arbitrary bisimulation relation \mathcal{R} and processes P and Q where $(P, Q) \in \mathcal{R}$, $P \rightsquigarrow_{\mathcal{R}} Q$ and $Q \rightsquigarrow_{\mathcal{R}} P$. That $P \sim Q$ follows immediately by coinduction using lemma 4.19 where \mathcal{X} is set to \mathcal{R} . \square

An important property of the bisimulation relation is that it is equivariant. When doing proofs we rely heavily on Lemma 4.3 which requires the simulation relation to be equivariant.

Lemma 4.22. eqvt \sim

Proof. By coinduction using Prop. 4.18 on \sim . Set \mathcal{X} to be $\{(p \bullet P, p \bullet Q) \mid P \sim Q\}$. Using Lemma 4.2 the proof is quite straight forward since \sim is a subset of \mathcal{X} by instantiating \mathcal{X} with the identity permutation. \mathcal{X} is also trivially equivariant. \square

Another important property of strong bisimulation is that it is an equivalence relation.

Lemma 4.23. \sim is an equivalence relation.

Proof.

Reflexivity: Use coinduction and set \mathcal{X} to the identity relation. The proof then follows trivially from Lemma 4.4.

Symmetry: Follows trivially from the definition of \sim .

Transitivity: By coinduction where \mathcal{X} is set to $\sim \circ \sim$. The result then follows by using Lemma 4.5. \square

We can now prove one of our main theorems.

Theorem 4.24. *Strong bisimulation is preserved by all operators except the input-prefix, i.e.*

$$\begin{array}{lll}
 \text{if } P \sim Q \text{ then} & \tau.P \sim \tau.Q & (1) \\
 \text{and} & \bar{a}b.P \sim \bar{a}b.Q & (2) \\
 \text{and} & [a = b]P \sim [a = b]Q & (3) \\
 \text{and} & [a \neq b]P \sim [a \neq b]Q & (4) \\
 \text{and} & P + R \sim Q + R & (5) \\
 \text{and} & P \mid R \sim Q \mid R & (6) \\
 \text{and} & (\nu x)P \sim (\nu x)Q & (7) \\
 \text{and} & !P \sim !Q & (8)
 \end{array}$$

Proof. To prove (1) to (5), Lemmas 4.6-4.7 and 4.10-4.12 are used respectively.

When proving (7) we use coinduction and set \mathcal{X} to $\{((\nu x)P, (\nu x)Q) \mid P \sim Q\}$. Lemma 4.14 can then prove preservation of both simulations.

To prove (8) we strengthen our assumption that $P \sim Q$ to $(P, Q) \in \text{Rep} \sim$ and use the coinduction principle 4.19 with \mathcal{X} set to $\text{Rep} \sim$. The preservation properties of the simulations can then be inferred by induction over $\text{Rep} \sim$ resulting in three cases from the derivation rules of Rep . These can be proven by Lemmas 4.16, 4.13 and 4.14 respectively.

The proof for (6) is deferred to Chapter 5. \square

We now define strong equivalence as the largest bisimulation relation closed under substitution and prove our next theorem.

Theorem 4.25. \sim^s is a congruence.

This result uses Theorem 4.24. In the preservation proof for the ν -operator the bound name must be α -converted to not clash with the substitution chain. We also need the following lemma to prove closure under input-prefix.

Lemma 4.26. $P \sim^s Q \implies a(x).P \sim^s a(x).Q$

Proof. By the definition of \sim^s and Lemma 4.9. \square

5. AN EXAMPLE DERIVATION

As an example of our proof techniques, we here present the omitted part of the proof for Theorem 4.24(6) – that strong bisimulation is preserved by the parallel operator.

The proof strategy amounts to proving simulations $P \rightsquigarrow_{\mathcal{R}} Q$. We begin by stating the requirements on \mathcal{R} that are necessary for the proof to go through. We do this before instantiating \mathcal{R} , since this makes the proof more general and better structured.

Recall Lemma 4.13 which is our preservation result for the \mid -operator.

$$\begin{array}{ll}
 (1) & P \rightsquigarrow_{\mathcal{R}} Q \\
 (2) & (P, Q) \in \mathcal{R} \\
 (3) & \text{Id} \subseteq \mathcal{R} \\
 (4) & \forall P Q S. (P, Q) \in \mathcal{R} \implies (P \mid S, Q \mid S) \in \mathcal{R}' \\
 (5) & \forall P Q x. (P, Q) \in \mathcal{R}' \implies ((\nu x)P, (\nu x)Q) \in \mathcal{R}' \\
 \hline
 & P \mid S \rightsquigarrow_{\mathcal{R}'} Q \mid S
 \end{array}$$

Two of these conditions concern \mathcal{R}' . Condition (4) is straightforward – if P and Q are in \mathcal{R} , then $P \mid S$ and $Q \mid S$ must be in \mathcal{R}' . Condition (5) is a bit less obvious but since the parallel operator can introduce restrictions, \mathcal{R}' must also be preserved by the ν -operator. Assumptions (2) and (3) ensure that the processes are in the \mathcal{R} to begin with. This is not a prerequisite for simulation, but we need to know this in order to use (1) when a process stands still and we need to place it in parallel with the derivative of the other process in \mathcal{R}' .

We provide a more in depth look at the proof for Lemma 4.13.

Proof. By Definition 4.1 we shall show:

$$\begin{aligned} (\forall a \ x \ T'. \ Q \mid S \xrightarrow{a \ll x} T' \wedge x \# P \mid S \implies \\ \exists P'. P \mid S \xrightarrow{a \ll x} P' \wedge \text{derivative}(a, x, P', T', \mathcal{R}')) \wedge \\ (\forall T' \ \alpha. Q \mid S \xrightarrow{\alpha} T' \implies \exists P'. P \mid S \xrightarrow{\alpha} P' \wedge (P', T') \in \mathcal{R}') \end{aligned}$$

We can now do case analysis on $Q \mid S \xrightarrow{\alpha} T'$ and $Q \mid S \xrightarrow{a \ll x} T'$. We get eight cases (the four rules for parallel composition as seen in Fig. 1 and their symmetric versions). We will focus on the *Close*-case, as it nicely demonstrates the advantages of the nominal package. Using our derived case analysis rule, Lemma 3.9, we can make sure that the bound names which appear in the *Close*-case do not clash with P by setting \mathcal{C} to P . After induction we get:

$$\begin{array}{ll} (6) & Q \xrightarrow{a(x)} Q' & (\text{assumption}) \\ (7) & S \xrightarrow{\bar{a}(y)} S' & (\text{assumption}) \\ (8) & x \# P & (\mathcal{C} = P \text{ in Lemma 3.9}) \\ (9) & y \# P & (\mathcal{C} = P \text{ in Lemma 3.9}) \\ \\ (10) & \exists P'. P \xrightarrow{a(x)} P' \wedge \text{derivative}((\text{Input } a), x, P', Q') & (1, 6, 8, \text{Def. 4.1}) \\ \\ (11) & P \xrightarrow{a(x)} P' & (10) \\ (12) & P \mid S \xrightarrow{\tau} (\nu y)(P'\{y/x\}, S') & (\text{Close}, 11, 7, 9) \\ \\ (13) & (P'\{y/x\}, Q'\{y/x\}) \in \mathcal{R} & (10, \text{Def. 4.1}) \\ (14) & (P'\{y/x\} \mid S', Q'\{y/x\} \mid S') \in \mathcal{R}' & (4, 13) \\ (15) & ((\nu y)(P'\{y/x\} \mid S'), (\nu y)(Q'\{y/x\} \mid S')) \in \mathcal{R}' & (5, 14) \\ \hline & \exists P'. P \mid S \xrightarrow{\tau} P' \wedge (P', (\nu y)(Q'\{y/x\} \mid S')) \in \mathcal{R}' & (12, 15) \end{array}$$

□

The above is a step-by-step version of the Isabelle proof and it mimics the way one could do a strict pen-and-paper version of the proof. Note how in steps 6 and 7, the bound names of both transitions generated by the induction rule are set to be fresh for P . We would otherwise have to α -convert both transitions. As it stands, all α -conversions are abstracted away completely. Steps 13-15 uses the preservation properties of \mathcal{R} and \mathcal{R}' to prove that the proper derivatives are in \mathcal{R}' .

Furthermore, we have to prove a lemma on chains of restrictions, since the *Close*-operator introduces new restrictions, as was also seen in lemma 3.9.

Definition 5.1. $(\nu \tilde{v})P$ denotes a chain of restrictions applied to P where \tilde{v} is a list, possibly empty, of restrictions.

Lemma 5.2. *Introduction rule for restriction chains:*

$$\frac{P \rightsquigarrow_{\mathcal{R}} Q \quad \text{eqvt } \mathcal{R} \quad \forall P Q x. (P, Q) \in \mathcal{R} \implies ((\nu x)P, (\nu x)Q) \in \mathcal{R}}{(\nu \tilde{v})P \rightsquigarrow_{\mathcal{R}} (\nu \tilde{v})Q}$$

Proof. By induction on \tilde{v} . □

The intuition behind the lemma is quite simple. If a simulation relation \mathcal{R} is preserved by the ν -operator and P simulates Q preserving \mathcal{R} , then since \mathcal{R} is preserved by restriction and thus $(\nu x)P$ simulates $(\nu x)Q$ preserving \mathcal{R} for an arbitrary name x , then by induction $(\nu \tilde{v})P$ must simulate $(\nu \tilde{v})Q$ preserving \mathcal{R} where \tilde{v} is an arbitrary chain of restricted names. This is a general lemma which is used repeatedly when proving bisimulations using the parallel operator.

We now proceed to the main proof of Theorem 4.24(6) using coinduction. We will need a set \mathcal{X} which captures the agents we are interested in and prove the simulations which compose the bisimulation. We define \mathcal{X} as $\{((\nu \tilde{v})(P \mid R), (\nu \tilde{v})(Q \mid R)) \mid P \sim Q\}$. The two simulation proofs we use reside in our main lemma since they share the same assumption, which is the way the proof is done inside Isabelle.

Proof. If $P \sim Q$ then $P \mid R \sim Q \mid R$.

- (1) $P \sim Q$ (assumption)
- (2) $(P \mid R, Q \mid R) \in \mathcal{X}$ (1, def. of \mathcal{X})

In order to use coinduction using Prop. 4.18 we must prove that every pair in \mathcal{X} simulates preserving $\mathcal{X} \cup \sim$. The members of \mathcal{X} have chains of restrictions so we first have to use Lemma 3.9 with a specific simulation relation in order to reason about them.

Lemma 5.3. *if $P \rightsquigarrow_{\sim} Q$ then $P \mid R \rightsquigarrow_{\mathcal{X} \cup \sim} Q \mid R$*

Proof.

$$\frac{\begin{array}{ll} \text{(i)} & P \rightsquigarrow_{\sim} Q \quad \text{(assumption)} \\ \text{(ii)} & \forall P Q R. (P, Q) \in \sim \implies (P \mid R, Q \mid R) \in \mathcal{X} \cup \sim \quad \text{(Def. of } \mathcal{X}) \\ \text{(iii)} & \forall P Q x. (P, Q) \in \mathcal{X} \cup \sim \implies \\ & ((\nu x)P, (\nu x)Q) \in \mathcal{X} \cup \sim \quad \text{(Def. of } \mathcal{X}, \text{ Lemma 4.24)} \end{array}}{P \mid R \rightsquigarrow_{\mathcal{X} \cup \sim} Q \mid R} \quad \text{(Lemma 3.9, i-iii, 1)}$$

□

From this lemma we see why Lemma 4.13 has to have different relations in the assumptions and the conclusion. The simulation we can assume is $P \rightsquigarrow_{\sim} Q$ but the one we need to prove is $P \mid R \rightsquigarrow_{\mathcal{X} \cup \sim} Q \mid R$.

We can now extend our simulation to include chains of restrictions.

Lemma 5.4. *If $P \rightsquigarrow_{\sim} Q$ then $(\nu \tilde{v})(P \mid R) \rightsquigarrow_{\mathcal{X} \cup \sim} (\nu \tilde{v})(Q \mid R)$*

Proof.

$$\frac{\begin{array}{ll} \text{(i)} & P \rightsquigarrow_{\sim} Q \quad \text{(assumption)} \\ \text{(ii)} & P \mid R \rightsquigarrow_{\mathcal{X} \cup \sim} Q \mid R \quad \text{(Lemma 5.3, i)} \\ \text{(iii)} & \text{eqvt}(\mathcal{X} \cup \sim) \quad \text{(def. of } \mathcal{X}, \text{ Lemma 4.22)} \\ \text{(iv)} & \forall P Q x. (P, Q) \in \mathcal{X} \cup \sim \implies \\ & ((\nu x)P, (\nu x)Q) \in \mathcal{X} \cup \sim \quad \text{(def. of } \mathcal{X}, \text{ Lemma 4.24)} \end{array}}{(\nu \tilde{v})(P \mid R) \rightsquigarrow_{\mathcal{X} \cup \sim} (\nu \tilde{v})(Q \mid R)} \quad \text{(Lemma 5.2, ii-iv)}$$

□

We can now prove our goal:

$$P \mid R \sim Q \mid R \quad (\text{coinduction, 2, Lemma 5.4, Def. 8.3}) \quad \square$$

It is interesting to note that we only have to prove simulations one way. When set up this way, Isabelle manages the symmetric versions of the proofs automatically. Of course, if the relation is not symmetric, such as in the proof of $(\nu x)P \sim P$ if $x \nmid P$, the two different directions require separate proofs, just as when doing the proofs on paper.

6. STRUCTURAL CONGRUENCE

Structural congruence rules are used to equate processes which are structurally different but intuitively behave in the same way. The way these rules are implemented differ in different formalisations. A common approach is to let the labeled transition system replace a term for a structurally congruent one in order to enable transitions. Another approach, and the one that we have chosen, is to prove that all structurally congruent terms are also bisimilar. The rules for structural congruence can be found in Fig. 2.

Theorem 6.1. *If $P \equiv Q$ then $P \sim Q$.*

As in the previous section we need to create auxiliary lemmas for all simulations we are interested in. Proving Theorem 6.1 requires that every structural congruence rule is proven individually. We will here demonstrate the most complicated example which is to prove associativity of the \mid -operator. We will need the following two lemmas for simulation.

Lemma 6.2.

$$\frac{\begin{array}{l} \forall P Q R. ((P \mid Q) \mid R, P \mid (Q \mid R)) \in \mathcal{R} \\ \forall P Q x. (P, Q) \in \mathcal{R} \implies ((\nu x)P, (\nu x)Q) \in \mathcal{R} \\ \forall P Q R x. x \nmid P \implies ((\nu x)((P \mid Q) \mid R), P \mid (\nu x)(Q \mid R)) \in \mathcal{R} \\ \forall P Q R x. x \nmid R \implies (((\nu x)(P \mid Q)) \mid R, (\nu x)(P \mid (Q \mid R))) \in \mathcal{R} \end{array}}{(P \mid Q) \mid R \rightsquigarrow_{\mathcal{R}} P \mid (Q \mid R)}$$

Proof. By case analysis over the \mid -operator. This proof contains 18 cases. The proofs individually are not very hard, there are just a lot of cases to cover. The assumptions used about the relation \mathcal{R} are used extensively in the proof. □

Lemma 6.3.

$$\frac{\begin{array}{l} \forall P Q R. (P \mid (Q \mid R), (P \mid Q) \mid R) \in \mathcal{R} \\ \forall P Q x. (P, Q) \in \mathcal{R} \implies ((\nu x)P, (\nu x)Q) \in \mathcal{R} \\ \forall P Q R x. x \nmid P \implies (P \mid (\nu x)(Q \mid R), (\nu x)((P \mid Q) \mid R)) \in \mathcal{R} \\ \forall P Q R x. x \nmid R \implies ((\nu x)(P \mid (Q \mid R)), ((\nu x)(P \mid Q)) \mid R) \in \mathcal{R} \end{array}}{P \mid (Q \mid R) \rightsquigarrow_{\mathcal{R}} (P \mid Q) \mid R}$$

Proof. Similar to Lemma 6.2. □

The structural congruence \equiv is defined as the smallest congruence satisfying the following laws:

- (1) If P and Q are variants of α -conversion then $P \equiv Q$.
- (2) The abelian monoid laws for Parallel: commutativity $P \mid Q \equiv Q \mid P$, associativity $(P \mid Q) \mid R \equiv P \mid (Q \mid R)$, and $\mathbf{0}$ as unit $P \mid \mathbf{0} \equiv P$; and the same laws for Sum.
- (3) The unfolding law $!P \equiv P \mid !P$
- (4) The scope extension laws

$$\begin{aligned}
 (\nu x)\mathbf{0} &\equiv \mathbf{0} \\
 (\nu x)(P \mid Q) &\equiv P \mid (\nu x)Q \quad \text{if } x \sharp P \\
 (\nu x)(P + Q) &\equiv P + (\nu x)Q \quad \text{if } x \sharp P \\
 (\nu x)[u = v]P &\equiv [u = v](\nu x)P \quad \text{if } x \neq u \text{ and } x \neq v \\
 (\nu x)[u \neq v]P &\equiv [u \neq v](\nu x)P \quad \text{if } x \neq u \text{ and } x \neq v \\
 (\nu x)(\nu y)P &\equiv (\nu y)(\nu x)P
 \end{aligned}$$

Figure 2: The definition of structural congruence.

In order to do the rest of this proof efficiently it turns out that we need to use other rules for structural congruence since Lemma 6.2 and 6.3 make heavy use of scoping rules. The coinduction rule (Prop. 4.18) allows us to work with an arbitrary relation, but to include the laws of structural congruence in this relation would be cumbersome. Instead we create the following coinduction rule.

Lemma 6.4. *Compositional coinduction rule. Let \mathcal{Y} be $\sim \circ (\mathcal{X} \cup \sim) \circ \sim$.*

$$\frac{\forall P' Q'. (P, Q) \in \mathcal{X} \quad \text{eqvt } \mathcal{X} \quad (P', Q') \in \mathcal{X} \implies P' \rightsquigarrow_{\mathcal{Y}} Q' \wedge Q' \rightsquigarrow_{\mathcal{Y}} P'}{P \sim Q}$$

Proof. By coinduction and transitivity of simulation. □

We can now prove associativity of the $|$ -operator.

Lemma 6.5. $(P \mid Q) \mid R \sim P \mid (Q \mid R)$

Proof. By coinduction using Lemma 6.4 and setting \mathcal{X} to $\{((\nu \tilde{v})((P \mid Q) \mid R), (\nu \tilde{v})(P \mid (Q \mid R)))\}$. Lemma 6.3 and 6.2 can then be used together with the laws for scope extrusion to complete the proof. □

The next step is to prove that all structurally congruent terms are strongly equivalent.

Theorem 6.6. $P \equiv Q \implies P \sim^s Q$

Proof. Nearly all work has already been done in Theorem 6.1. These proofs do, however, require manual alpha conversions when dealing with scoping rules as the cases where the restricted name clashes with the substitution chain must be taken into consideration. This is an example of where pen-and-paper proofs often are less rigorous than strictly required. □

The proofs we have done in this section are not overly complicated but require a solid attention to detail. Many of the proofs have many cases and even though the results have never been in doubt, having them fully machine checked convinces us that no case has been overlooked. Moreover, without the framework to abstract away from bound names the amount of cases for all different α -variants would have been very much larger.

7. WEAK BISIMULATION

7.1. Basic definitions. Weak bisimulation equivalence is often called observation equivalence. The intuition is that τ -transitions are considered internal and hence invisible to the outside environment. For two processes to be observation equivalent, they only need to mimic the visible actions of each other. More formally, we reason about a τ -chain $P \xrightarrow{\hat{\tau}} P'$ as the reflexive transitive closure of τ -actions, i.e. $P \xrightarrow{\hat{\tau}} P' \stackrel{\text{def}}{=} P \xrightarrow{\tau}^* P'$. A weak transition is then said to be an action preceded and succeeded by a τ -chain.

Weak late bisimulation is complicated for input actions. It requires substitutions made as a result of the input to be applied immediately to the input derivative before the succeeding τ -chain is executed, and that one such derivative can continue to simulate for all possible received names, see e.g. [35]. Therefore the weak late semantics needs to carry additional information in the labels as follows.

Definition 7.1.

$$\begin{aligned} P \xrightarrow{\alpha} P' &\stackrel{\text{def}}{=} P \xrightarrow{\hat{\tau}} \alpha \xrightarrow{\hat{\tau}} P' \\ P \xrightarrow{\bar{a}(x)} P' &\stackrel{\text{def}}{=} P \xrightarrow{\hat{\tau}} \bar{a}(x) \xrightarrow{\hat{\tau}} P' \\ P \xrightarrow{u:a(x)@P''} P' &\stackrel{\text{def}}{=} P \xrightarrow{\hat{\tau}} a(x) \xrightarrow{\hat{\tau}} P'' \wedge P'' \{u/x\} \xrightarrow{\hat{\tau}} P' \end{aligned}$$

Residuals are written in the same way for weak as for strong transitions, except for the input case which is written $u : a(x)@P'' \prec P'$. A transition can also be written as $P \xRightarrow{} \text{Res}$ where Res is a residual.

The transition $P \xrightarrow{u:a(x)@P''} P'$ means that P can do a τ -chain and then $a(x)$ to an agent P'' where x is substituted for u and another τ -chain is done to P' . The agent P'' represents the exact state where the substitution is made. This will be important when we define weak simulation.

Note that the bound name x in the bound output case is bound in P' and normal α -conversions can be applied. Also, even though we are modeling a late semantics, the name x is *not* bound in P' in the input-transition as it is substituted for u before the τ -chain. We can still do α -conversions through the following lemma:

Lemma 7.2. *if $P \xrightarrow{u:a(x)@P''} P'$ and $y \# P$ then $P \xrightarrow{u:a(y)@(x y) \bullet P''} P'$*

We also need to weaken the transitions in the standard way:

Definition 7.3. Weak late transitions

$$\begin{aligned} P \xrightarrow{\hat{\alpha}} P' &\stackrel{\text{def}}{=} P \xrightarrow{\hat{\tau}} P' \text{ if } \alpha = \tau \\ &P \xrightarrow{\alpha} P' \text{ otherwise} \end{aligned}$$

We can now define weak late simulation.

Definition 7.4. The agent P can weakly late simulate the agent Q preserving \mathcal{R} , written $P \approx_{\mathcal{R}} Q$, if

$$\begin{aligned} & (\forall a \ x \ Q'. \ Q \xrightarrow{\bar{a}(x)} Q' \wedge x \# P \implies \\ & \quad \exists P'. \ P \xrightarrow{\bar{a}(x)} P' \wedge (P', Q') \in \mathcal{R}) \wedge \\ & (\forall a \ x \ Q'. \ Q \xrightarrow{a(x)} Q' \wedge x \# P \implies \\ & \quad \exists P''. \ \forall u. \ \exists P'. \ P \xrightarrow{u:a(x)@P''} P' \wedge (P', Q'\{u/x\}) \in \mathcal{R}) \wedge \\ & (\forall \alpha \ Q'. \ Q \xrightarrow{\alpha} Q' \implies \exists P'. \ P \xrightarrow{\hat{\alpha}} P' \wedge (P', Q') \in \mathcal{R}) \end{aligned}$$

The important aspect of weak late simulation is the fact mentioned above – that an input-action $a(x)$ must be matched by a weak transition with the same input derivative P'' for *all* possible instantiations u of the bound name. From our definition, we can derive an introduction rule for weak simulation similar to the one done for strong simulation in Lemma 4.3.

In the standard way we define another version of simulation \approx where we require the simulating process to do at least one action to mimic the simulated agent. The definition of \approx is the same as for $\approx_{\mathcal{R}}$ except that the simulating process in the last conjunct uses $\xrightarrow{\alpha}$ instead of $\xrightarrow{\hat{\alpha}}$.

Definition 7.5. $P \approx_{\mathcal{R}} Q$ if

$$\begin{aligned} & (\forall a \ x \ Q'. \ Q \xrightarrow{\bar{a}(x)} Q' \wedge x \# P \implies \\ & \quad \exists P'. \ P \xrightarrow{\bar{a}(x)} P' \wedge (P', Q') \in \mathcal{R}) \wedge \\ & (\forall a \ x \ Q'. \ Q \xrightarrow{a(x)} Q' \wedge x \# P \implies \\ & \quad \exists P''. \ \forall u. \ \exists P'. \ P \xrightarrow{u:a(x)@P''} P' \wedge (P', Q'\{u/x\}) \in \mathcal{R}) \wedge \\ & (\forall \alpha \ Q'. \ Q \xrightarrow{\alpha} Q' \implies \exists P'. \ P \xrightarrow{\alpha} P' \wedge (P', Q') \in \mathcal{R}) \end{aligned}$$

7.2. Lifted semantics. Our preservation proofs for weak transitions are very similar to the corresponding proofs for strong transitions. We achieve this by *lifting* the operational semantics, i.e. mapping each rule from Fig. 1 to a corresponding rule using weak transitions. The following transition system can be derived for the transitions defined in Def. 7.1.

Lemma 7.6. *The lifted semantics for the transitions defined in Def. 7.1.*

$$\begin{array}{c} a(x).P \xrightarrow{u:a(x)@P} P\{u/x\} \quad \mathbf{Input} \qquad \bar{a}b.P \xrightarrow{\bar{a}b} P \quad \mathbf{Output} \qquad \tau.P \xrightarrow{\tau} P \quad \mathbf{Tau} \\ \\ \frac{P \Vdash Res}{[a = a]P \Vdash Res} \quad \mathbf{Match} \qquad \frac{P \Vdash Res \quad a \neq b}{[a \neq b] \Vdash Res} \quad \mathbf{Mismatch} \\ \\ \frac{P \xrightarrow{\bar{a}b} P' \quad a \neq b}{(\nu b)P \xrightarrow{\bar{a}(b)} P'} \quad \mathbf{Open} \qquad \frac{P \Vdash Res}{P + Q \Vdash Res} \quad \mathbf{Sum} \qquad \frac{P \xrightarrow{u:a(x)@P''} P' \quad x \# Q}{P \mid Q \xrightarrow{u:a(x)@P'' \mid Q} P' \mid Q} \quad \mathbf{ParIn} \\ \\ \frac{P \xrightarrow{\bar{a}(x)} P' \quad x \# Q}{P \mid Q \xrightarrow{\bar{a}(x)} P' \mid Q} \quad \mathbf{ParBO} \end{array}$$

$$\begin{array}{c}
\frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \quad \mathbf{ParF} \qquad \frac{P \xrightarrow{b:a(x)@P''} P' \quad Q \xrightarrow{\bar{a}b} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'} \quad \mathbf{Comm} \\
\\
\frac{P \xrightarrow{u:a(x)@P''} P' \quad Q \xrightarrow{\bar{a}(y)} Q' \quad y \# P}{P \mid Q \xrightarrow{\tau} (\nu y)(P' \mid Q')} \quad \mathbf{Close} \qquad \frac{P \xrightarrow{\bar{a}(x)} P' \quad y \# (a, x)}{(\nu y)P \xrightarrow{\bar{a}(x)} (\nu y)P'} \quad \mathbf{ResBO} \\
\\
\frac{P \xrightarrow{u:a(x)@P''} P' \quad y \# (a, u, x)}{(\nu y)P \xrightarrow{u:a(x)@(\nu y)P''} (\nu y)P'} \quad \mathbf{ResIn} \qquad \frac{P \xrightarrow{\alpha} P' \quad x \# \alpha}{(\nu x)P \xrightarrow{\alpha} (\nu x)P'} \quad \mathbf{ResF} \\
\\
\frac{P \mid !P \Longrightarrow Res}{!P \Longrightarrow Res} \quad \mathbf{Replication}
\end{array}$$

When trying to lift the semantics to the transitions defined in Def. 7.3 we encounter difficulties. The rules which do not have $\xrightarrow{\hat{\alpha}}$ in the assumptions trivially follow from Lemma 7.6, but of the remaining, only **ParF** and **ResF** can be lifted.

Corollary 7.7. *The lifted rules for **ParF** and **ResF**.*

$$\frac{P \xrightarrow{\hat{\alpha}} P'}{P \mid Q \xrightarrow{\hat{\alpha}} P' \mid Q} \quad \mathbf{ParF} \qquad \frac{P \xrightarrow{\hat{\alpha}} P' \quad x \# \alpha}{(\nu x)P \xrightarrow{\hat{\alpha}} (\nu x)P'} \quad \mathbf{ResF}$$

The operational rules from Fig. 1 that we cannot be lifted in this manner, as opposed to the ones in Lemma 7.6, are **Match**, **Mismatch**, **Sum** and **Replication** in the case where $\alpha = \tau$ and $P = P'$.

7.3. Preservation properties. To prove preservation properties for weak simulations we need to lift the preservation proofs from strong simulations to weak ones. For \approx this turns out to be unproblematic. The lemmas require the same assumptions to be proven with the addition that we sometimes need to know that if $(P, Q) \in \mathcal{R}$ then $P \approx_{\mathcal{R}} Q$. The reason for this is that after following a τ -chain, we need to know that we are still inside the simulation. For \approx , however, the lemmas that we could not lift in Cor. 7.7 need their assumption strengthened. These lemmas are:

Lemma 7.8.

$$\frac{P \approx_{\mathcal{R}} Q \quad \forall P Q a. (P, Q) \in \mathcal{R} \implies ([a = a]P, Q) \in \mathcal{R}}{[a = b]P \approx_{\mathcal{R}} [a = b]Q}$$

Proof. By the definition of \approx and Prop. 3.6. In the case where the τ -transition stands still, the second assumption is used to prove that the derivatives are still in \mathcal{R} . \square

Lemma 7.9.

$$\frac{P \approx_{\mathcal{R}} Q \quad \forall P Q a b. (P, Q) \in \mathcal{R} \wedge a \neq b \implies ([a \neq b]P, Q) \in \mathcal{R}}{[a \neq b]P \approx_{\mathcal{R}} [a \neq b]Q}$$

Proof. By the definition of \approx and Prop. 3.7. In the case where the τ -transition stands still, the second assumption is used to prove that the derivatives are still in \mathcal{R} . \square

Lemma 7.10.

$$\frac{\begin{array}{l} (P, Q) \in \mathcal{R} \quad \text{eqvt } \mathcal{R} \\ \forall P Q. (P, Q) \in \mathcal{R} \implies P \rightsquigarrow_{\mathcal{R}} Q \\ \forall P Q. (P \mid !P, Q) \in \mathcal{R} \implies (!P, Q) \in \mathcal{R} \end{array}}{!P \rightsquigarrow_{\text{Rep } \mathcal{R}} !Q}$$

Proof. Similar to Lemma 4.16 but when a τ -action stands still the fourth assumption is used. \square

The other preservation lemmas look the same as their strong counterparts. Their proofs need to treat input-actions differently as there is a noticeable difference in how input-actions are treated in strong and weak simulations. Other than this, the proofs follow the same pattern.

Weak bisimulation equivalence is defined using coinduction in exactly the same way as strong bisimulation. As a result, all coinduction rules which were generated for strong bisimulation are also generated for weak.

Definition 7.11. Weak bisimulation equivalence, \approx , is the largest relation satisfying:

$$P \approx Q \implies P \rightsquigarrow_{\approx} Q \wedge Q \rightsquigarrow_{\approx} P$$

Weak bisimulation is not a congruence since it is neither preserved by the $+$ -operator nor by the input-prefix, but it is preserved by all other operators.

Theorem 7.12. \approx is preserved by all operators except $+$ and input prefix.

Proof. The first step in this proof is to use the lifted preservation rules for weak simulation. In order to prove preservation of **Match**, **Mismatch** and **Replication**, we need the results $P \approx [a = a]P$, $P \approx [a \neq b]P$ when $a \neq b$ as well as the structural congruence result $P \mid !P \approx !P$. \square

To obtain a congruence we follow the standard procedure. The proofs of the preservation lemmas for \rightleftharpoons are similar to their strong counterparts since all rules from the operational semantics can be lifted using Lemma 7.6.

We can now define weak congruence.

Definition 7.13. $P \cong Q \stackrel{\text{def}}{=} P \rightleftharpoons_{\approx} Q \wedge Q \rightleftharpoons_{\approx} P$

Note that this is not a coinductive definition since it refers to \approx . The proof that \cong is preserved by all operators except input-prefix corresponds closely to our corresponding proof for \sim . The proof that \cong^s is a congruence follows in the same manner.

Theorem 7.14. \cong is preserved by all operators except input-prefix.

Proof. This proof is nearly identical to the one for Theorem 4.24, but we use our preservation proofs for \rightleftharpoons instead of the ones for \rightsquigarrow . \square

Lemma 7.15. \cong^s is a congruence

Proof. Similar to Lemma 4.25. \square

7.4. Relationships between equivalences. We prove that $\sim \subseteq \cong \subseteq \approx$. Among other things, this implies that the weaker bisimulation equivalences contain structural congruence. The first part of this proof is to establish correspondance properties between the different types of transitions.

Corollary 7.16.

$$\begin{aligned} \text{If } P \xrightarrow{\bar{a}(x)} P' \text{ then } P \xrightarrow{\bar{a}(x)} P' \\ \text{If } P \xrightarrow{\alpha} P' \text{ then } P \xrightarrow{\alpha} P' \\ \text{If } P \xrightarrow{a(x)} P' \text{ then } P \xrightarrow{u:a(x)@P'} P'\{u/x\} \end{aligned}$$

Proof. Follows from the definition of $P \xRightarrow{} Res$ by adding empty τ -chains before and after the transitions. \square

The next step is to do the same for simulations.

Corollary 7.17. *If $P \rightsquigarrow_{\mathcal{R}} Q$ then $P \simeq_{\mathcal{R}} Q$*

Proof. By the definition of \rightsquigarrow , \simeq and Cor. 7.16. \square

And finally for weak congruence.

Corollary 7.18. *If $P \sim Q$ then $P \cong Q$*

Proof. By the definition of \sim , \cong and Cor. 7.17. \square

The corresponding proof for our congruence relations follow trivially.

Corollary 7.19. *If $P \sim^s Q$ then $P \cong^s Q$*

Proof. Follows from the definitions of \sim^s , \cong^s and Cor. 7.18. \square

We can use the same technique when reasoning about weak bisimulation.

Corollary 7.20. *If $P \xRightarrow{\alpha} P'$ then $P \xRightarrow{\hat{\alpha}} P'$*

Proof. Follows from the definitions of $\xRightarrow{\alpha}$ and $\xRightarrow{\hat{\alpha}}$ as $\xRightarrow{\alpha}$ can do everything $\xRightarrow{\hat{\alpha}}$ can do except doing an empty sequence of τ s. \square

Followed by simulation

Corollary 7.21. *If $P \simeq Q$ then $P \approx Q$*

Proof. Follows from the definitions of \simeq , \approx and Cor. 7.20. \square

And finally for weak bisimulation.

Corollary 7.22. *If $P \cong Q$ then $P \approx Q$*

Proof. Follows from the definitions of \approx , \cong and Cor. 7.21. \square

Using the techniques above our results follow as a simple corollary.

Corollary 7.23.

$$\begin{aligned} \text{If } P \equiv Q \text{ then } P \approx Q \\ \text{and } P \cong Q \\ \text{and } P \cong^s Q \end{aligned}$$

Proof. Follows from Theorem 6.6 and Corollaries 7.18, 7.22 and 7.19. \square

7.5. The Hennessy Lemma. As an example we prove the Hennessy Lemma.

Theorem 7.24. $P \approx Q$ iff $\tau.P \cong Q \vee P \cong Q \vee P \cong \tau.Q$

Proof. We first prove the lemma in the direction left-to-right. We will need the following auxiliary lemmas.

Lemma 7.25. If $P \approx_{\mathcal{R}} Q$ then $\tau.P \simeq_{\mathcal{R}} Q$

Proof. By the definition of \simeq . The interesting case is the Q does a τ -action, and $\tau.P$ can always mimic that with at least one step since $P \approx_{\mathcal{R}} Q$. \square

Lemma 7.26.

$$\frac{P \approx_{\mathcal{R}} Q \quad \forall Q'. Q \xrightarrow{\tau} Q' \implies (P, Q') \notin \mathcal{R}}{P \simeq_{\mathcal{R}} Q}$$

Proof. This follows from the definition of \approx and \simeq . The only difference being that for a τ -transition, the simulating process by \approx can do an empty sequence of τ s. whereas in \simeq it cannot. In our assumptions we remove this option. \square

Lemma 7.27. If $P \xrightarrow{\tau} P'$ and $(P', Q) \in \mathcal{R}$ then $P \simeq_{\mathcal{R}} \tau.Q$

Proof. Follows from the definition of \simeq . \square

We can now complete our proof for the left-to-right direction of the Hennessy lemma by doing proofs on the following cases:

- (1) $\exists P'. P \xrightarrow{\tau} P' \wedge P' \approx Q$
- (2) $\exists Q'. Q \xrightarrow{\tau} Q' \wedge P \approx Q'$

In the case that 1 or 2 holds we use Lemmas 7.25 and 7.27 to prove the first and third disjunct. In the case that neither hold, Lemma 7.26 can be used for both directions of the bisimulation. This concludes the proof in the left-to-right direction.

We will need the following lemmas for the direction right-to-left.

Lemma 7.28. If $\tau.P \simeq_{\mathcal{R}} Q$ then $P \approx_{\mathcal{R}} Q$

Proof. By the definition of \approx . If Q does a τ -action and $\tau.P$ simulates by doing a single τ -step, P can stand still and end up in the same state. Otherwise, P can always move to the same state as $\tau.P$ by doing one less τ -step. \square

Lemma 7.29.

$$\frac{P \simeq_{\mathcal{R}} \tau.Q \quad \forall P' Q'. (P', Q') \in \mathcal{R} \implies P' \approx_{\mathcal{R}} Q'}{P \approx_{\mathcal{R}} Q}$$

Proof. From the definition of \simeq we get a τ -chain $P \implies_{\tau} P'$ for some P' where $(P', Q) \in \mathcal{R}$. We also know that $P' \approx_{\mathcal{R}} Q$. By the definition of \approx we can add the chain $P \implies_{\tau} P'$ to any simulation of Q . \square

To finish the the proof we use Lemmas 7.28 and 7.29 for the first and third disjunct and Cor. 7.22 for the second one. \square

$a(x).P \xrightarrow{au}_e P\{u/x\}$ Input	$\bar{a}b.P \xrightarrow{\bar{a}b}_e P$ Output	$\tau.P \xrightarrow{\tau}_e P$ Tau
$\frac{P \mapsto_e \text{Res}}{[a = a]P \mapsto_e \text{Res}}$ Match	$\frac{P \mapsto_e \text{Res} \quad a \neq b}{[a \neq b]P \mapsto_e \text{Res}}$ Mismatch	
$\frac{P \xrightarrow{\bar{a}b}_e P' \quad a \neq b}{(\nu b)P \xrightarrow{\bar{a}(b)}_e P'}$ Open	$\frac{P \mapsto_e \text{Res}}{P + Q \mapsto_e \text{Res}}$ Sum	
$\frac{P \xrightarrow{\bar{a}(x)}_e P' \quad x \# Q}{P \mid Q \xrightarrow{\bar{a}(x)}_e P' \mid Q}$ ParB	$\frac{P \xrightarrow{\alpha}_e P'}{P \mid Q \xrightarrow{\alpha}_e P' \mid Q}$ ParF	
$\frac{P \xrightarrow{ab}_e P' \quad Q \xrightarrow{\bar{a}b}_e Q'}{P \mid Q \xrightarrow{\tau}_e P' \mid Q'}$ Comm	$\frac{P \xrightarrow{ay}_e P' \quad Q \xrightarrow{\bar{a}(y)}_e Q' \quad y \# P}{P \mid Q \xrightarrow{\tau}_e (\nu y)(P' \mid Q')}$ Close	
$\frac{P \xrightarrow{\bar{a}(x)}_e P' \quad y \# (a, x)}{(\nu y)P \xrightarrow{\bar{a}(x)}_e (\nu y)P'}$ ResB	$\frac{P \xrightarrow{\alpha}_e P' \quad y \# \alpha}{(\nu y)P \xrightarrow{\alpha}_e (\nu y)P'}$ ResF	
$\frac{P \mid !P \mapsto_e \text{Res}}{!P \mapsto_e \text{Res}}$ Replication		

Figure 3: The *Par*- and the *Res*-rule in the early operational semantics are still split, but the input action contains no bound names. Symmetric versions have been elided.

8. EARLY SEMANTICS AND BISIMULATION

8.1. Early semantics. In the early semantics the input action carries the name received rather than a bound name, so we have that the process $a(x).P$ can receive all names u doing an action au and ending up in the derivative $P\{u/x\}$. The main difference to late semantics is that substitution is done at the input prefix rule, i.e. as early as possible, and not during communication.

The way we write actions differ somewhat from the late semantics. We write the early transitions in a similar way, but with a subscript e to differentiate them from the late ones. Moreover, In the early semantics, a transition $\xrightarrow{\alpha}_e$ can include an input-transition as it does not contain a bound name. The intuition is that an action is denoted α if it contains no binders. As a result, our Isabelle definition for early residuals need to be changed.

Definition 8.1. The early residual datatype.

```

datatype freeRes = InputR name name
                  | OutputR name name
                  | TauR

nominal_datatype residual = BoundOutputR name "<<name>> pi"
                          | FreeR freeRes pi

```

8.2. Early bisimulation. The definition of early simulation is similar to its late counterpart. The difference between the two is that no distinction has to be made for the input-action as the substitution takes place before any communication is made.

Definition 8.2. The agent P can early simulate the agent Q preserving \mathcal{R} , written $P \rightsquigarrow_{e\mathcal{R}} Q$, if

$$\begin{aligned} & (\forall a \ x \ Q'. \ Q \xrightarrow{\bar{a}(x)}_e Q' \wedge x \# P \implies \exists P'. \ P \xrightarrow{\bar{a}(x)}_e P' \wedge (P', Q') \in \mathcal{R}) \wedge \\ & (\forall \alpha \ Q'. \ Q \xrightarrow{\alpha}_e Q' \implies \exists P'. \ P \xrightarrow{\alpha}_e P' \wedge (P', Q') \in \mathcal{R}) \end{aligned}$$

Bisimulation is again defined using our standard coinduction technique.

Definition 8.3. Early bisimulation equivalence, \sim_e , is the largest relation satisfying:

$$P \sim_e Q \implies P \rightsquigarrow_{e\sim_e} Q \wedge Q \rightsquigarrow_{e\sim_e} P$$

All the preservation proofs and congruence results for late bisimulation have also been done for early. This did require creating rules for case analysis on the early operational semantics in a similar way as was done for late. We have created the library of preservation lemmas similar to the one for late semantics. This work was pretty straightforward and the two libraries work in the same way except for how they treat input actions. Once this was done, the proofs for early bisimulation were nearly identical to their late counterparts and required very little extra work.

Theorem 8.4. \sim_e is preserved by all operators except input-prefix.

Theorem 8.5. \sim_e^s is a congruence.

8.3. Weak early bisimulation. We have also proven our results for weak early bisimulation. We use the same technique as we did for weak late bisimulation by lifting the early operational semantics to a weak counterpart. The weak early operational semantics can be written on a simpler form, however, as weak early simulation does not require any knowledge of the point that a substitution was made. A weak early transition is hence written $\xrightarrow{\alpha}_e$ or $\xrightarrow{\hat{\alpha}}_e$, where α is an arbitrary transition.

Lemma 8.6. *The lifted semantics for the weak early operational semantics.*

$$\begin{array}{l} a(x).P \xrightarrow{au}_e P\{u/x\} \quad \mathbf{Input} \qquad \bar{a}b.P \xrightarrow{\bar{a}b}_e P \quad \mathbf{Output} \qquad \tau.P \xrightarrow{\tau}_e P \quad \mathbf{Tau} \\ \\ \frac{P \Vdash_e Res}{[a = a]P \Vdash_e Res} \quad \mathbf{Match} \qquad \frac{P \Vdash_e Res \quad a \neq b}{[a \neq b] \Vdash_e Res} \quad \mathbf{Mismatch} \qquad \frac{P \xrightarrow{\bar{a}b}_e P' \quad a \neq b}{(\nu b)P \xrightarrow{\bar{a}(b)}_e P'} \quad \mathbf{Open} \\ \\ \frac{P \Vdash_e Res}{P + Q \Vdash_e Res} \quad \mathbf{Sum} \qquad \frac{P \xrightarrow{\bar{a}(x)}_e P' \quad x \# Q}{P \mid Q \xrightarrow{\bar{a}(x)}_e P' \mid Q} \quad \mathbf{ParB} \qquad \frac{P \xrightarrow{\alpha}_e P'}{P \mid Q \xrightarrow{\alpha}_e P' \mid Q} \quad \mathbf{ParF} \\ \\ \frac{P \xrightarrow{a(b)}_e P' \quad Q \xrightarrow{\bar{a}b}_e Q'}{P \mid Q \xrightarrow{\tau}_e P' \mid Q'} \quad \mathbf{Comm} \qquad \frac{P \xrightarrow{a(y)}_e P' \quad Q \xrightarrow{\bar{a}(y)}_e Q' \quad y \# P}{P \mid Q \xrightarrow{\tau}_e (\nu y)(P' \mid Q')} \quad \mathbf{Close} \\ \\ \frac{P \xrightarrow{\bar{a}(x)}_e P' \quad y \# (a, x)}{(\nu y)P \xrightarrow{\bar{a}(x)}_e (\nu y)P'} \quad \mathbf{ResB} \qquad \frac{P \xrightarrow{\alpha}_e P' \quad x \# \alpha}{(\nu x)P \xrightarrow{\alpha}_e (\nu x)P'} \quad \mathbf{ResF} \qquad \frac{P \mid !P \Vdash_e Res}{!P \Vdash_e Res} \quad \mathbf{Replication} \end{array}$$

This semantics is very similar to its late counterpart. The reason for this is that in the weak late operational semantics, the instantiations of input bound names occur inside the transition before the succeeding τ -chain. This becomes apparent when we compare the lifted rules for **Input**. In the late semantics, it looks like an early transition since it contains the name u received in the input. The rules **Close** and **Comm** also behave in the same way. We have proven Lemmas 8.13, 8.14, 8.15, 8.16 and Theorem 8.17 for the correspondence of the weak late and early transition systems.

We do encounter the same problem when trying to lift the $\hat{\alpha} \rightarrow_e$ transitions in that **Match**, **Mismatch**, **Sum** and **Replication** cannot be lifted, for the same reason as in the late semantics. The lifted early rules correspond more closely to their strong counterparts than the lifted late rules correspond to theirs. The weak early and late rules are very similar to each other since the Input-rules behave in the same intuitive manner. The difference between the two semantics is not so much in the operational rules as in the definition of simulation.

Definition 8.7. The agent P can weakly early simulate the agent Q preserving \mathcal{R} , written $P \approx_{e\mathcal{R}} Q$, if

$$\begin{aligned} (\forall a x Q'. Q \xrightarrow{\bar{a}(x)}_e Q' \wedge x \# P \implies \exists P'. P \xrightarrow{\bar{a}(x)}_e P' \wedge (P', Q') \in \mathcal{R}) \wedge \\ (\forall \alpha Q'. Q \xrightarrow{\alpha}_e Q' \implies \exists P'. P \xrightarrow{\hat{\alpha}}_e P' \wedge (P', Q') \in \mathcal{R}) \end{aligned}$$

Definition 8.8. Weak early bisimulation equivalence, \approx_e , is the largest relation satisfying:

$$P \approx_e Q \stackrel{\text{def}}{=} P \approx_{e\approx_e} Q \wedge Q \approx_{e\approx_e} P$$

Theorem 8.9. \approx is preserved by all operators except $+$ and input-prefix.

Proof. Similar to the proof for Theorem 7.12. □

Weak early bisimulation is not a congruence for the same reason as weak late bisimulation, and in order to create a congruence we need to define a weak early congruence simulation, \simeq_e , by replacing the $\hat{\alpha} \rightarrow_e$ in Def. 8.7 by $\alpha \rightarrow_e$.

We can now define our weak early congruence.

Definition 8.10. $P \cong_e Q \stackrel{\text{def}}{=} P \simeq_{e\cong_e} Q \wedge Q \simeq_{e\cong_e} P$

Theorem 8.11. \cong_e is preserved by all operators except input-prefix.

Proof. Similar to the proof for Theorem 7.14. □

Lemma 8.12. \cong_e^s is a congruence.

Proof. Proved in a similar way as Lemma 7.15. □

8.4. Relationships between equivalences. Not surprisingly, strong early and weak early relations enjoy the same inclusion properties as their late counterparts, i.e. $\sim_e \subseteq \cong_e \subseteq \approx_e$. Furthermore, $\sim \subseteq \sim_e$.

The proof for the latter is more involved and requires correspondance proofs between strong early and late actions. The connection we have proved between them is that every early τ -transition has a corresponding late τ -transition and vice versa. More precisely, the following lemmas are proven:

Lemma 8.13. $P \xrightarrow{ab}_e P' \text{ iff } P \xrightarrow{ab} P'$

Proof. By induction over the possible output transitions. \square

Lemma 8.14. $P \xrightarrow{\bar{a}(x)}_e P' \text{ iff } P \xrightarrow{\bar{a}(x)} P'$

Proof. By induction over the possible bound output transitions.

Before induction, the transitions are α -converted such that x is fresh for a and P . In the Open cases, Lemma 8.13 is used. \square

Lemma 8.15. *If* $P \xrightarrow{a(x)} P'$ *then* $P \xrightarrow{au}_e P'\{u/x\}$

Proof. By induction over the possible input transitions. Before induction, the late transition is α -converted such that x is fresh for a , u and P . \square

Lemma 8.16. *If* $P \xrightarrow{au}_e P'$ *then for all name contexts* \mathcal{C} , *there exists an* x *and a* P'' *s.t.* $P \xrightarrow{a(x)} P''$, $P' = P''\{u/x\}$ *and* $x \# \mathcal{C}$

Proof. By induction over the possible input-transitions. When doing the induction, the last conjunct of the goal is not used but only the first two ones. We can then take the results from the induction and eliminate the existential quantifiers, pick a new fresh name x' which is fresh for P'' and \mathcal{C} and instantiate the goal with x' and $(x \ x') \bullet P''$. \square

We can now prove our theorem.

Theorem 8.17. $P \xrightarrow{\tau}_e P' \text{ iff } P \xrightarrow{\tau} P'$

Proof. By induction over the possible τ -transitions. In the **Open**, **Comm** and **Close** cases, Lemma 8.13, 8.14, 8.15 and 8.16 are used. \square

We can now continue with our correspondence proofs between late and early semantics.

Lemma 8.18. *If* $P \rightsquigarrow_{\mathcal{R}} Q$ *then* $P \rightsquigarrow_{e\mathcal{R}} Q$

Proof. By case analysis of Def. 8.2. Lemmas 8.13, 8.14, 8.15, 8.16 and Theorem 8.17 are used to transform the early simulations to late ones and back again after applying Def. 4.1. Lemma 8.16 has the context \mathcal{C} instantiated as P to ensure that the generated bound name is fresh for P as required by Def. 4.1. \square

We now prove that all late bisimilar processes are also early bisimilar.

Theorem 8.19. *If* $P \sim Q$ *then* $P \sim_e Q$

Proof. By coinduction using Prop. 4.19 and setting \mathcal{X} to \sim . Lemma 8.18 then proves our goal. \square

Corollary 8.20. *If* $P \sim^s Q$ *then* $P \sim_e^s Q$

Proof. Follows trivially from Theorem 8.19. \square

With these we can very easily prove our theorems about structural congruence for early.

Corollary 8.21.

$$\begin{aligned} \text{If } P \equiv Q \text{ then } P \sim_e Q \\ \text{and } P \sim_e^s Q \end{aligned}$$

Proof. Follows trivially from Theorems 6.1, 8.19, 6.6 and Cor. 8.20. \square

Finally, for the weak early semantics:

Corollary 8.22.

$$\begin{aligned} \text{If } P \sim_e Q \text{ then } P \cong_e Q \\ \text{If } P \sim_e^s Q \text{ then } P \cong_e^s Q \\ \text{If } P \cong_e Q \text{ then } P \approx_e Q \end{aligned}$$

Proof. Similar to their corresponding proofs in section 7.4. □

From this our structural congruence results follow trivially.

Corollary 8.23.

$$\begin{aligned} \text{If } P \equiv Q \text{ then } P \cong_e Q \\ \text{and } P \approx_e Q \\ \text{and } P \cong_e^s Q \\ \text{and } P \approx_e^s Q \end{aligned}$$

Proof. From Theorems 6.1, 6.6 and Cor. 8.22. □

9. RESULTS AND CONCLUSIONS

9.1. Current Status. We have used the new nominal datatype package in Isabelle to model the π -calculus and our results are very encouraging. We have proved a substantial part of [30], in particular preservation properties of strong and weak bisimulation for both late and early operational semantics. Other results include that all late τ -transitions have a corresponding early one and vice versa and that all late bisimulation relations have an early counterpart. Moreover, we have proven that all the bisimulation relations we have investigated contain structural congruence. We have created a substantial library concerning the fundamental mechanisms in the π -calculus, such as substitution and transitions. One of our main contributions is that the proofs resemble the ones on paper very closely, since we make precise the traditional “hand waving” with respect to bound names. Since we are using Isabelle, we can write our proofs in a very readable form using *Isar* [47]. We believe this to be the most extensive formalisation of a process calculus ever done inside a theorem prover.

In recent work we put our formalisation to the test by proving that the axiomatisation of strong late bisimilarity is sound and complete [10]. The proofs were complex, but again mapped their pen-and-paper equivalents very closely and we made extensive use of the foundation provided in this paper.

The nominal package is still work in progress and it is constantly being updated. One recent addition allows for users to define functions on their nominal datatypes using an automatically generated recursion combinator [44]. At the moment the only function we use is substitution.

9.2. Related Work. The π -calculus has been subjected to many attempts at formalisations. Gabbay made a formalisation in [20] utilising FM set theory, the precursor of nominal logic. His work is mathematically close to ours. The rest of this section will focus mainly on formalisations which have been subject to mechanisation inside a theorem prover. Early sketches in HOL include [31, 26]. Later attempts have also been made using de-Brujin indices where names are encoded using natural numbers. The most extensively used approach is higher order abstract syntax (HOAS) where weak HOAS is the technique most similar to ours. We here comment on the more important approaches.

de Bruijn indices are heavily used in software which reasons about terms with binders; an example for the π -calculus is the Mobility Workbench [46]. They work well in these environments as they have very nice algorithmic properties. However, these properties do not provide an intuitive mathematical framework. In [24], Daniel Hirschhoff formalised a subset of the π -calculus excluding sum, match and mismatch in Coq using de Bruijn indices. The theories formalised was that early bisimulation is a congruence as well as the structural congruence results. Preliminary work was also made to help formalise Milner’s encoding of the λ -calculus [29]. Hirschhoff writes the following:

Technical work, however, still represents the biggest part of our implementation, mainly due to the managing of de Bruijn indexes. . . Of our 800 proved lemmas, about 600 are concerned with operators on free names.

Fraenkel Mostowski set theory was one of the first serious attempts to formalise nominal logic. It is standard ZF set theory but with an extra freshness axiom added. In [20], Gabbay formalises a portion of the π -calculus in FM. In this approach a N-quantifier (new quantifier) is used to generate names which are fresh for the current context. The nominal package does not provide support for this quantifier, but the same effect is achieved by instantiating our rules with a set of context names. Gabbay also started work on incorporating a framework for FM inside Isabelle [19] with which formalisations such as ours could be made. Unfortunately, this early version of nominal logic was incompatible with the axiom of choice and had to be used in Isabelle/PURE – a bare boned set of theories. This choice of framework was necessary since Isabelle/HOL contains the axiom of choice, but the attempt was later abandoned.

HOAS has been used to model the π -calculus in both Coq [25], by Honsell et. al., and in Isabelle by Röckl and Hirschhoff [39]. In [25] the late operational semantics is encoded together with late strong bisimulation. The proved results include that the algebraic laws presented in [30] are sound where the non-trivial proofs include preservation results for bisimulation and the results for structural congruence. When using HOAS terms, binders are represented as functions of type `name->term`. However, if these functions range over the entire function space they may produce exotic terms, so the formalisations need to ensure that those are avoided. In [39], a special well-formedness predicate is used to filter out the exotic terms. Another problem is that since abstraction is handled by the meta-logic of the theorem prover, reasoning about binders at the object level can become problematic. In [25] we can read:

The main drawback in HOAS is the difficulty of dealing with metatheoretic issues concerning names in process contexts, *i.e.* terms of type `name->proc`. As a consequence, some metatheoretic properties involving substitution and freshness of names inside proofs and processes, cannot be proved inside the framework and instead have to be postulated.

Our approach is completely free from any extra axioms, and since nominal logic is a first order approach we do not have exotic terms. Moreover, freshness conditions are part of the nominal infrastructure and all such conditions are explicitly known at the object level and do not have to be postulated, thus no extra infrastructure for choosing particular names is needed.

9.3. Impact and Further Work. Theorem provers suffer from a somewhat well-deserved reputation of being hard to use for the uninitiated. However, having theories formalised by a computer has significant advantages and making theorem provers easy to use for the general engineer is a high priority. We believe that our work helps in this venture. The challenging part has been to create inductive rules and easy-to-use definitions for simulation and bisimulation. With this done the actual proofs done in the theorem prover are not much harder than the ones done on paper.

Our next goal will be to provide support for model- and bisimulation checking on actual protocols such as ad-hoc routing. Particularly processes with infinite state space are of interest as these cannot be handled by automatic tools like the Mobility Workbench.

There are several variants of the π -calculus, polyadic π -calculus and higher order π -calculus just to name two. We believe that our definitions for simulation and bisimulation can easily be transferred to many other calculi.

Acknowledgments. We would like to thank Stefan Berghofer for his generous help with the inner workings of Isabelle, Christian Urban for developing the nominal datatype package and providing extensive support and insights, and Lars-Henrik Eriksson for discussions on theorem provers. We would also like to thank the anonymous referees for their many helpful and constructive comments.

REFERENCES

- [1] Agda: An interactive theorem prover. <http://unit.aist.go.jp/cvs/Agda>.
- [2] Spec#. <http://research.microsoft.com/specsharp/>.
- [3] The Verisoft project. <http://www.verisoft.de>.
- [4] Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. In *Proceedings of POPL '01*, pages 104–115. ACM, January 2001.
- [5] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. In *Fourth ACM Conference on Computer and Communications Security*, pages 36–47. ACM Press, 1997.
- [6] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, Nathan J. Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The POPLmark challenge. In *International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, August 2005.
- [7] Gertrud Bauer and Tobias Nipkow. The 5 colour theorem in Isabelle/Isar. In *Theorem Proving in Higher Order Logics*, volume 2410 of *Lecture Notes in Computer Science*, pages 67–82. Springer-Verlag, 2002.
- [8] Jesper Bengtson. Generic implementations of process calculi in Isabelle. In *The 16th Nordic Workshop on Programming Theory (NWPT'04)*, pages 74–78, 2004.
- [9] Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffei. Refinement Types for Secure Implementations. In *IEEE Computer Security Foundation series*, 2008. To appear.
- [10] Jesper Bengtson and Joachim Parrow. A completeness proof for bisimulation in the pi-calculus using Isabelle. *Electron. Notes Theor. Comput. Sci.*, 192(1):61–75, 2007.
- [11] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.

- [12] Karthikeyan Bhargavan, Cedric Fournet, Andrew D. Gordon, and Stephen Tse. Verified interoperable implementations of security protocols. In *CSFW '06: Proceedings of the 19th IEEE Workshop on Computer Security Foundations*, pages 139–152, Washington, DC, USA, 2006. IEEE Computer Society.
- [13] Bruno Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *CSFW '01: Proceedings of the 14th IEEE Workshop on Computer Security Foundations*, page 82, Washington, DC, USA, 2001. IEEE Computer Society.
- [14] Gérard Boudol. The pi-calculus in direct style. In *Conference Record of POPL '97*, pages 228–241, 1997.
- [15] Luca Cardelli and Andrew D. Gordon. Mobile ambients. In *Foundations of Software Science and Computation Structures: First International Conference, FOSSACS '98*. Springer-Verlag, Berlin Germany, 1998.
- [16] Robert L. Constable, Stuart F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, NJ, 1986.
- [17] Cédric Fournet and Georges Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 372–385, St. Petersburg Beach, Florida, January 21–24 1996. ACM.
- [18] M. J. Gabbay. A theory of inductive definitions with α -equivalence, PhD thesis, University of Cambridge, 2000.
- [19] M. J. Gabbay. Automating Fraenkel-Mostowski Syntax. In *TPHOLs, 15th International Conference on Theorem Proving in Higher Order Logics*, number CP-2002-211736, pages 60–70. NASA, August 2002.
- [20] M. J. Gabbay. The pi-calculus in FM. In Fairouz Kamareddine, editor, *Thirty-five years of Automath*, volume 28 of *Applied Logic Series*, pages 247–269. Kluwer, 2003.
- [21] M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13:341–363, 2001.
- [22] G. Gonthier. A computer-checked proof of the four colour theorem. Technical report, Microsoft Research Cambridge, 2004.
- [23] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, New York, NY, USA, 1993.
- [24] Daniel Hirschhoff. A full formalisation of pi-calculus theory in the calculus of constructions. In *TPHOLs '97: Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics*, pages 153–169, London, UK, 1997. Springer-Verlag.
- [25] Furio Honsell, Marino Miculan, and Ivan Scagnetto. π -calculus in (co)inductive type theory. *Theoretical Computer Science*, 253(2):239–285, 2001.
- [26] Thomas F. Melham. A mechanized theory of the pi-calculus in HOL. *Nordic Journal of Computing*, 1(1):50–76, 1994.
- [27] R. Milner. *A Calculus of Communicating Systems*. Number 92 in LNCS. Springer-Verlag, 1980.
- [28] R. Milner. The polyadic pi-calculus: a tutorial. In F. L. Bauer, W. Brauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specification*, pages 203–246. Springer-Verlag, 1993.
- [29] Robin Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2(2):119–141, 1992.
- [30] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I/II. *Inf. Comput.*, 100(1):1–77, 1992.
- [31] Otmane Aït Mohamed. Mechanizing a pi-calculus equivalence in HOL. In *Proceedings of the 8th International Workshop on Higher Order Logic Theorem Proving and Its Applications*, pages 1–16, London, UK, 1995. Springer Verlag.
- [32] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer-Verlag, 2002.
- [33] Tobias Nipkow, Gertrud Bauer, and Paula Schultz. Flyspeck I: Tame graphs. In U. Furbach and N. Shankar, editors, *Automated Reasoning (IJCAR 2006)*, volume 4130, pages 21–35, 2006.
- [34] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, jun 1992. Springer-Verlag.
- [35] Joachim Parrow. An introduction to the pi-calculus. In *Handbook of Process Algebra*, pages 479–543. Elsevier, 2001.

- [36] Joachim Parrow and Björn Victor. The fusion calculus: Expressiveness and symmetry in mobile processes. In *Logic in Computer Science*, pages 176–185, 1998.
- [37] A. M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 186:165–193, 2003.
- [38] A. M. Pitts. Alpha-structural recursion and induction. *Journal of the ACM*, 53:459–506, 2006.
- [39] Christine Röckl and Daniel Hirschhoff. A fully adequate shallow embedding of the π -calculus in Isabelle/HOL with mechanized syntax analysis. *J. Funct. Program.*, 13(2):415–451, 2003.
- [40] Davide Sangiorgi and David Walker. *The π -calculus: A Theory of Mobile Processes*. Cambridge University Press, New York, NY, USA, 2001.
- [41] Natarajan Shankar. *Metamathematics, Machines and Gödel’s Proof*. Cambridge University Press, 1994.
- [42] C. Urban, A. M. Pitts, and M. J. Gabbay. Nominal unification. *Theoretical Computer Science*, 323:473–497, 2004.
- [43] Christian Urban. Nominal techniques in Isabelle/HOL. *Journal of Automated Reasoning*, 40(4):327–356, 2008.
- [44] Christian Urban and Stefan Berghofer. A recursion combinator for nominal datatypes implemented in Isabelle/HOL. In Ulrich Furbach and Natarajan Shankar, editors, *IJCAR*, volume 4130 of *Lecture Notes in Computer Science*, pages 498–512. Springer, 2006.
- [45] Christian Urban, Stefan Berghofer, and Michael Norrish. Barendregt’s variable convention in rule inductions. In Frank Pfenning, editor, *CADE*, volume 4603 of *Lecture Notes in Computer Science*, pages 35–50. Springer, 2007.
- [46] Björn Victor and Faron Moller. The Mobility Workbench — a tool for the π -calculus. In David Dill, editor, *CAV’94: Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 428–440. Springer-Verlag, 1994.
- [47] Markus Wenzel. Isar - a generic interpretative approach to readable formal proof documents. In Yves Bertot, Gilles Dowek, André Hirschowitz, C. Paulin, and Laurent Théry, editors, *TPHOLs*, volume 1690 of *Lecture Notes in Computer Science*, pages 167–184. Springer, 1999.

Examples of proof.sty

\infer draws beautiful proof figures easily:

$$(1) \quad \frac{\frac{B11 \& B12 \& B13 \quad B21 \& B22 \& B23}{B} \quad C}{A}$$

$$(2) \quad \frac{\frac{B11 \& B12 \& B13 \quad B21 \& B22 \& B23}{B} \quad C}{A1 \& A2 \& A3 \& A4 \& A5 \& A6}$$

$$(3) \quad \frac{C \quad \frac{B11 \& B12 \& B13 \quad B21 \& B22 \& B23}{B}}{A1 \& A2 \& A3 \& A4 \& A5 \& A6}$$

You can use also some variations:

$$(4) \quad \frac{B11 \& B12 \& B13 \quad \begin{array}{c} B21 \& B22 \& B23 \\ \vdots \end{array}}{\frac{B}{A} \quad C} \quad (1)$$

$$(5) \quad \frac{B11 \& B12 \& B13 \quad B21 \& B22 \& B23}{\sum_B \quad C} \quad (2)$$

$$\begin{array}{c} \vdots (1) \\ A1 \& A2 \& A3 \& A4 \& A5 \& A6 \end{array}$$

$$(6) \quad \frac{A \& B \& C}{A}$$

Here are more practical examples:

$$(7) \quad \frac{A \quad B}{A \& B} \quad (\&I) \quad \frac{A \& B}{A} \quad (\&E_l) \quad \frac{A \& B}{B} \quad (\&E_r)$$

$$\frac{\begin{array}{c} [A] \\ \vdots \\ B \end{array}}{A \rightarrow B} \quad (\rightarrow I) \quad \frac{A \rightarrow B \quad A}{B} \quad (\rightarrow E)$$

Some techniques: Use \vcenter for an equation of proofs.

(8)

$$\pi = A \frac{B \ C}{D \ E}$$

Use `\kern` to adjust the form of a proof.

(9)

$$\frac{A \ B \ C}{D \ E}$$