Formalising the π-Calculus Using Nominal Logic

Jesper Bengtson and Joachim Parrow

Department of Information Technology, University of Uppsala, Sweden

Abstract. We formalise the pi-calculus using the nominal datatype package, a 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 unison 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 [8] 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, 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 Approach

The goal of our project is to provide a library in an automated theorem prover, Isabelle/HOL [11], 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 [1].

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. This kind of reasoning does not necessarily imply that proofs conducted in this manner are incorrect, but for a full formal formalisation of a system involving binders, a solid mathematical foundation where α-equivalence is clearly defined has to be created.

Our approach is to formulate the π-calculus using ideas from nominal logic developed by (Pitts et al. [13,5,17]). 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 [3] 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 [19] extends the work done by Pitts and Gabbay 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 which allows the user to create functions on nominal datatypes.

1.3 Results

Our contribution is to use this nominal package in Isabelle to describe the π-calculus. We have proved substantial portions of [9] 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.

1.4 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 Sections 3–7 we demonstrate how the π-calculus syntax, semantics, and bisimulation equivalences are represented in our framework. In the concluding section we compare our efforts to related work and comment on planned further work. An extended version of this paper together with the Isabelle source files can be found at http://www.it.uu.se/katalog/jesperb/pi.

2 The Nominal Datatype Package

For a more thorough presentation of the nominal datatype package the reader is referred to [19], 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.* 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 1. *Equivariance*
$$\text{eqvt}\ \mathcal{R} \overset{def}{=} \forall p\ T\ U.\ (T,\ U) \in \mathcal{R} \Longrightarrow (p \bullet T,\ p \bullet U) \in \mathcal{R}$$

Another key concept is the notion of *support.* The definition, in general, is that the support 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. One says that a name a is *fresh* for a term T, written $a \sharp T$, if a is not in the support of T.

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

Proposition 1. $[x].T = [y].U \Longrightarrow$
$(x = y \wedge T = U) \vee (x \neq y \wedge x \sharp U \wedge T = (x\ y) \bullet U)$

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. $c \sharp (x, y, T, U) \wedge [x].T = [y].U \Longrightarrow (x\ c) \bullet T = (y\ c) \bullet U$

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

3 Defining the π-Calculus

We present a version of the monadic π-calculus [9]. We assume that the reader is familiar with the basic ideas of its syntax and semantics.

Definition 2. *Defining the π-calculus in Isabelle*

```
nominal_datatype pi = PiNil
                    | Tau pi
                    | Input name "<<name>> pi"
                    | Output name name pi
                    | Match name name pi
                    | Sum pi pi
                    | Par pi pi
                    | Res "<<name>> pi"
                    | Bang pi
```

This definition is an example of Isabelle syntax. The notation $\ll$*name*$\gg$ *pi* indicates that *name* is bound in *pi*. For the rest of the paper we shall use the traditional syntax for π-calculus terms, e.g. writing inputs as $a(x)$ and restrictions as (νx).

The nominal datatype package automatically generates lemmas for reasoning about α-equivalence between processes – the ones generated from Prop. 1 can be found in the following proposition.

Proposition 3. *The most commonly used α-equivalence rules for the* Input- *and the* Restriction case*.*

Input: $a(x).P = b(y).Q \Longrightarrow a = b \wedge ((x = y \wedge P = Q) \vee$
$(x \neq y \wedge x \sharp Q \wedge P = (x\ y) \bullet Q))$

Restriction: $(\nu x)P = (\nu y)Q \Longrightarrow (x = y \wedge P = Q) \vee$
$(x \neq y \wedge x \sharp Q \wedge P = (x\ y) \bullet Q)$

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.

The most commonly used function is substitution where $P\{a/b\}$ (which can be read P with a for b) is the substitution of all occurrences of b in P with a.

4 Operational Semantics

4.1 Definitions

We use the standard operational semantics [9]. 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 [2].

However, in the π-calculus the action α may bind a name, and the scope of this binding extends into P'. In particular we shall sometimes need to α-convert the action together with the derivative P'. For this purpose, we create a *residual*-datatype which is a nominal datatype. It binds the bound names of an action also in the derivative.

Definition 3. *The residual datatype*

```
datatype subject = Input name
                 | BoundOutput name

datatype freeRes = Output name name
                 | Tau

nominal_datatype residual = BoundR subject "<<name>> pi"
                          | FreeR freeRes pi
```

We introduce a notation for an arbitrary action with a bound name, i. e., an *Input*- or a *Bound Output* action.

Definition 4

(i) $P \xrightarrow{a\ll x\gg} P'$ denotes a transition with the bound name x in the action. Note that a is of type `subject`.

(ii) $P \xrightarrow{\alpha} P'$ denotes a transition without bound names. Note that α is of type `freeRes`.

We can now define our operational semantics using inductively defined sets, and the set will contain tuples of two elements – one process and one residual.

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 like $x \notin \mathrm{bn}(\alpha)$, i.e. x is not in the bound names of α. 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. The operational semantics, including the split rules for **Par** and **Res** can be found in Fig. 1.

$$\frac{}{a(x).P \xrightarrow{a(x)} P}\ \textbf{Input} \qquad \frac{}{\bar{a}b.P \xrightarrow{\bar{a}b} .P}\ \textbf{Output} \qquad \frac{}{\tau.P \xrightarrow{\tau} P}\ \textbf{Tau}$$

$$\frac{P \xrightarrow{\alpha} P'}{[a = a]P \xrightarrow{\alpha} P'}\ \textbf{Match} \qquad \frac{P \xrightarrow{\bar{a}b} P' \quad a \neq b}{(\nu b)P \xrightarrow{\bar{a}(b)} P'}\ \textbf{Open} \qquad \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'}\ \textbf{Sum}$$

$$\frac{P \xrightarrow{a\ll x\gg} P' \quad x \sharp Q}{P \mid Q \xrightarrow{a\ll x\gg} P' \mid Q}\ \textbf{ParB} \qquad \frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q}\ \textbf{ParF}$$

$$\frac{P \xrightarrow{a(x)} P' \quad Q \xrightarrow{\bar{a}b} Q'}{P \mid Q \xrightarrow{\tau} P'\{b/x\} \mid Q'}\ \textbf{Comm} \qquad \frac{P \xrightarrow{a(x)} P' \quad Q \xrightarrow{\bar{a}(y)} Q' \quad y \sharp P}{P \mid Q \xrightarrow{\tau} (\nu y)(P'\{y/x\} \mid Q')}\ \textbf{Close}$$

$$\frac{P \xrightarrow{a\ll x\gg} P' \quad y \sharp (a,\ x)}{(\nu y)P \xrightarrow{a\ll x\gg} (\nu y)P'}\ \textbf{ResB} \qquad \frac{P \xrightarrow{\alpha} P' \quad y \sharp \alpha}{(\nu y)P \xrightarrow{\alpha} (\nu y)P'}\ \textbf{ResF}$$

$$\frac{P \mid !P \xrightarrow{\alpha} P'}{!P \xrightarrow{\alpha} P'}\ \textbf{Replication}$$

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

4.2 Induction and Case Analysis Rules

Isabelle will automatically create rules for both induction and case analysis of the semantics. These rules, however, assume that the equivalence relation used is syntactic equivalence and not α-equivalence. While the nominal datatype package automatically creates induction rules for nominal datatypes there is no such automation for the kind of inductively defined sets that we use in the semantics. The rules generated by Isabelle for our operational semantics suffer from two problems, which we now address in turn.

The first problem is that some semantic 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 automatically for nominal data types, and for the same reason: avoiding name clashes and α-conversions later in the proof). This idea stems from [19] which has been developed independently of our work in [18]. The logical framework has also been covered in [14].

As an example a derived rule for case analysis of the parallel operator is shown in the following proposition where the parameter $\mathcal{C}$ is a set of context names:

Lemma 1. *The derived case analysis rule for the parallel operator with no bound names in the transition.*

$$\frac{\begin{array}{l} P \mid Q \xrightarrow{\alpha} R' \\ \forall P'. P \xrightarrow{\alpha} P' \wedge R' = P' \mid Q \Longrightarrow Prop \\ \forall P'\ Q'\ a\ x\ b.\ P \xrightarrow{a(x)} P' \wedge Q \xrightarrow{\bar{a}b} Q' \wedge \alpha = \tau \wedge \\ \qquad\qquad\qquad R' = P'\{b/x\} \mid Q' \wedge x \sharp \mathcal{C} \Longrightarrow Prop \\ \forall P'\ Q'\ a\ x\ y.\ P \xrightarrow{a(x)} P' \wedge Q \xrightarrow{\bar{a}(y)} Q' \wedge y \sharp P \wedge \alpha = \tau\ \wedge \\ \qquad\qquad\qquad R' = (\nu y)(P'\{y/x\} \mid Q') \wedge x \sharp \mathcal{C} \wedge y \sharp \mathcal{C} \Longrightarrow Prop \end{array}}{Prop}$$

The two semantic rules which introduce bound names are the *Comm*- and the *Close* rules. The rule can be instantiated with an arbitrary finite set of context names $\mathcal{C}$ and these bound names will be set fresh for that set.

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. 1 and 2, 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 4. *The automatically generated case analysis rule for the ν-operator, based on Prop. 1, where no bound name occurs in the action.*

$$\frac{\begin{array}{c} (\nu x)P \xrightarrow{\alpha} P' \\ \forall Q\ Q'\ \beta\ y.\ Q \xrightarrow{\beta} Q' \wedge y \sharp \beta \wedge (\nu x)P = (\nu y)Q \wedge \alpha = \beta \wedge P' = (\nu y)Q' \Longrightarrow Prop \end{array}}{Prop}$$

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. 1 or Prop. 2 but neither of these rules are convenient to work with. Prop. 1 causes a case explosion which forces us to prove the same thing several times for different permutations on terms and Prop. 2 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 2. *Case analysis rule derived from Prop. 4.*

$$\frac{\begin{array}{c} (\nu x)P \xrightarrow{\alpha} P' \\ \forall P''.\ P \xrightarrow{\alpha} P'' \wedge x \sharp \alpha \wedge P' = (\nu x)P'' \Longrightarrow Prop \end{array}}{Prop}$$

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.

In this way, all checks for α-equivalence between agents have been abstracted away and we are left with one very simple case to work with. Similar rules have been derived from all generated induction rules, on the depth of inference as well as case analysis of all operators. The only operator which requires an induction rule rather than a case analysis rule is the !-operator as it is the only operator which occurs in the premise of its inference rule, as can be seen in Fig. 1. There is also an induction rule over all possible transitions.

5 Strong Bisimulation

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:

$$P \stackrel{\text{def}}{=} a(u).(\nu b)\bar{b}x.0$$
$$Q \stackrel{\text{def}}{=} a(x).0$$

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 $a(x).(\nu b)\bar{b}x$. 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 5. *The agent P can simulate the agent Q preserving $\mathcal{R}$, written $P \leadsto_{\mathcal{R}} Q$, if*

$$\begin{aligned}&(\forall a\ x\ Q'.\ Q \xrightarrow{a\ll x\gg} Q' \wedge x \sharp P \Longrightarrow\\ &\qquad\qquad \exists P'.\ P \xrightarrow{a\ll x\gg} P' \wedge \mathit{derivative}(a,\ x,\ P',\ Q',\ \mathcal{R})) \wedge\\ &(\forall \alpha\ Q'.\ Q \xrightarrow{\alpha} Q' \Longrightarrow \exists P'.\ P \xrightarrow{\alpha} P' \wedge (P', Q') \in \mathcal{R})\end{aligned}$$

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

Note that the argument a in derivative is of type `subject` as described in Def. 3. Thus, the requirement is that if Q has an action then P has the same action, and the derivatives P' and Q' are in $\mathcal{R}$.

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 $(P, Q) \in \mathcal{R} \to P \leadsto_{\mathcal{R}} Q$; 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 [6], yields shorter proofs. Our main improvement, however, is in the treatment of the bound name x. 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 1 we bypass this tedium and derive the following introduction rule for an arbitrary finite set of context names $\mathcal{C}$. This set is provided by the user to ensure that the bound name is distinct from any name occurring so far in the proof.

Lemma 3. *An introduction rule for simulation avoiding name clashes.*

$$\frac{\begin{array}{l}\text{eqvt } \mathcal{R}\\ (\forall a\ x\ Q'.\ Q \overset{a\ll x\gg}{\longrightarrow} Q' \wedge x \sharp \mathcal{C} \Longrightarrow \\ \qquad\qquad \exists P'.\ P \overset{a\ll x\gg}{\longrightarrow} P' \wedge \mathit{derivative}(a,\ x,\ P',\ Q',\ \mathcal{R}))\\ (\forall \alpha\ Q'.\ Q \xrightarrow{\alpha} Q' \Longrightarrow \exists P'.\ P \xrightarrow{\alpha} P' \wedge (P', Q') \in \mathcal{R})\end{array}}{P \leadsto_{\mathcal{R}} Q}$$

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 proof trivial.

Bisimulation equivalence can now be described using coinduction, or as the greatest fixed point derived from a monotonic function.

Definition 6. *Bisimulation equivalence, $\sim$, is a* coinductive *definition.*

$$P \sim Q \overset{def}{=} P \leadsto_{\sim} Q \ \wedge\ Q \leadsto_{\sim} P$$

Note that we do not need to define what a bisimulation is; our coinductive definition uses $P \leadsto_{\mathcal{R}} Q$ 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 checking whether or not two processes are bisimilar, one picks a set $\mathcal{X}$ which contains the processes and which represents what it is we are trying to prove. It then suffices to show that all members of $\mathcal{X}$ are simulated preserving $\mathcal{X} \cup \sim$. The coinduction rule is automatically generated by Isabelle.

Strong bisimulation is not a congruence as it is not preserved by the input-prefix. We write $\mathcal{R}^s$ for the closure of the relation $\mathcal{R}$ under all substitutions.

Definition 7. *$P\ \mathcal{R}^s\ Q \overset{def}{=} \forall \sigma.\ P\sigma\ \mathcal{R}\ Q\sigma$ where σ is a chain of substitutions.*

From this we can define strong equivalence as the largest bisimulation relation closed under substitution. One of our main results is proving in Isabelle that strong equivalence is a congruence.

Theorem 1. *$\sim^s$ is a congruence.*

6 Weak Bisimulation

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 $P \overset{\hat{\tau}}{\Longrightarrow} P'$ as the reflexive transitive closure of τ-actions, i.e. $P \overset{\hat{\tau}}{\Longrightarrow} P' \overset{\text{def}}{=} P \xrightarrow{\tau}^{*} P'$. A weak transition is then said to be an action preceded and succeeded by a τ-chain.

In the simulation of an input, weak late simulation is complicated. 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. [12]. Therefore the weak late semantics needs to carry additional information in the labels as follows.

Definition 8. *Weak late transitions*

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

As with our previous transitions, we let α range over free actions with no bound names. 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 4. *if $P \overset{u:a(x)@P''}{\Longrightarrow} P'$ and $y \sharp P$ then $P \overset{u:a(y)@(x\ y)\bullet P''}{\Longrightarrow} P'$*

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

Definition 9. $P \overset{\hat{\alpha}}{\Longrightarrow} P' \overset{def}{=} P \overset{\hat{\tau}}{\Longrightarrow} P'$ *if* $\alpha = \tau$
$P \overset{\alpha}{\Longrightarrow} P'$ *otherwise*

We can now define weak late simulation.

Definition 10. *The agent P can weakly late simulate the agent Q preserving $\mathcal{R}$, written $P \gtrsim_{\mathcal{R}} Q$, if*

$$(\forall a\ x\ Q'.\ Q \xrightarrow{\bar{a}(x)} Q' \wedge x \sharp P \Longrightarrow$$
$$\exists P'.\ P \overset{\bar{a}(x)}{\Longrightarrow} P' \wedge (P',\ Q') \in \mathcal{R}) \wedge$$
$$(\forall a\ x\ Q'.\ Q \xrightarrow{a(x)} Q' \wedge x \sharp P \Longrightarrow$$
$$\exists P''.\ \forall u.\ \exists P'.\ P \overset{u:a(x)@P''}{\Longrightarrow} P' \wedge (P',\ Q'\{u/x\}) \in \mathcal{R}) \wedge$$
$$(\forall \alpha\ Q'.\ Q \xrightarrow{\alpha} Q' \Longrightarrow \exists P'.\ P \overset{\hat{\alpha}}{\Longrightarrow} P' \wedge (P',\ Q') \in \mathcal{R})$$

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 3.

Weak bisimulation equivalence is defined using coinduction in exactly the same way as strong bisimulation.

Definition 11. *Weak bisimulation equivalence, $\approx$, is a* coinductive *definition.*
$P \approx Q \overset{def}{=} P \mathrel{\leadsto}_{\approx} Q \ \wedge\ Q \mathrel{\leadsto}_{\approx} P$

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. The first step in in this proof is to lift the operational semantics to a weak context, that is, for every operational semantics rule we derive a weak counterpart. This means that the proof strategies for strong equivalence carry over to weak equivalence. As an example Lemma 5 derives the weak operational semantics for the |-operator.

Lemma 5

$$\frac{P \overset{u:a(x)@P''}{\Longrightarrow} P' \quad x \sharp Q}{P \mid Q \overset{u:a(x)@P''|Q}{\Longrightarrow} P' \mid Q}\ \textbf{ParIn}$$

$$\frac{P \overset{\bar{a}(x)}{\Longrightarrow} P' \quad x \sharp Q}{P \mid Q \overset{\bar{a}(x)}{\Longrightarrow} P' \mid Q}\ \textbf{ParBO} \qquad \frac{P \overset{\hat{\alpha}}{\Longrightarrow} P'}{P \mid Q \overset{\hat{\alpha}}{\Longrightarrow} P' \mid Q}\ \textbf{ParF}$$

$$\frac{P \overset{b:a(x)@P''}{\Longrightarrow} P' \quad Q \overset{\bar{a}b}{\Longrightarrow} Q'}{P \mid Q \overset{\tau}{\Longrightarrow} P' \mid Q'}\ \textbf{Comm} \frac{P \overset{y:a(x)@P''}{\Longrightarrow} P' \quad Q \overset{\bar{a}(y)}{\Longrightarrow} Q' \quad y \sharp P}{P \mid Q \overset{\tau}{\Longrightarrow} (\nu y)(P' \mid Q')}\ \textbf{Close}$$

All operational rules cannot be lifted in this manner. The rules from Fig. 1 where we cannot do this are *Match*, *Sum* and *Replication* in the case where $\alpha = \tau$ and $P = P'$. In order to prove preservation of *Match* and *Replication*, we have to prove that $P \approx [a = a]P$ and $P \mid !P \approx\ !P$.

To obtain a congruence we follow the standard procedure. We define weak congruence simulations, $\mathrel{\leadsto}$, in the same way as weak simulations, Def. 10, but for the initial free transitions, we replace $\overset{\hat{\alpha}}{\Longrightarrow}$ with $\overset{\alpha}{\Longrightarrow}$. In other words, the simulating process must match at least the first action from the other process, even invisible ones. We have proven that it is possible to lift all the operational rules from Fig 1 to $\overset{\alpha}{\Longrightarrow}$.

Definition 12. $P \cong Q \overset{def}{=} P \mathrel{\Rightarrow\!\!>_{\approx}} Q \ \wedge \ Q \mathrel{\Rightarrow\!\!>_{\approx}} P.$

Note that this is not a recursive 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 2. *$\cong^s$ is a congruence.*

7 Early Semantics

In the early semantics the input action carries the name received rather than a bound name, so we have $a(x).P \xrightarrow{au} P\{u/x\}$ for all u. We have created transition systems for both early and late operational semantics. The proof strategies involved for dealing with the two different approaches are nearly identical, even though the actual theories are disjoint. The connection we have proved between them is that every early τ-transition has a corresponding late τ-transition and vice versa.

The derived early and late weak semantics are much more similar to each other then their strong counterparts. The reason for this is that in the weak late operational semantics, the instantiations of input bound names occurs inside the transition before the succeeding τ-chain. This becomes apparent when we look at our lifted rule for input-actions:

Lemma 6. $a(x).P \overset{u:a(x)@P}{\Longrightarrow} P\{u/x\}.$

Even in our late semantics this looks like an early transition since it contains the name u received in the input. The difference between weak early and late semantics is not so much in the transition system, but in the definition of simulation.

All proofs that we have done for the late semantics, simulation- and bisimulation relations have been done also for the early semantics. We have also proven that all late bisimulation relations that we have considered are included in their early counterparts.

8 Results and Conclusions

8.1 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 [9], in particular preservation properties of strong and weak bisimulation, and both late and early. Other results include that all all late τ-transitions have a corresponding early one and vice versa and that all late bisimulation relations have an early counterport. Moreover, we have proven that all structurally congruent terms are bisimilar using both early and late semantics. We have created a substantial library concerning the fundamental mechanism 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* [21]. We believe this to be the most extensive formalisation of a process calculus ever to be done inside a theorem prover.

The nominal package is still work in progress and it is constantly being updated. One very recent addition allows for users to define functions on their nominal datatypes using an automatically generated recursion combinator [16]. At the moment we only use substitution as a function (both single and sequential).

8.2 Related Work

The π-calculus has been subject for many attempts at formalisations. Early sketches in HOL include [10,7]. Later attempts have also been made using de-Bruijn indices where names are encoded using natural numbers. More recent work by Gabbay utilised FM set theory [4], the precursor of nominal logic, although this attempt was later abandoned. The most extensively used approach is higher order abstract syntax (HOAS) where weak HOAS is the technique most similar to ours.

de Bruijn indices are heavily used in software which reasons about terms with binders; an example for the π-calculus is the Mobility Workbench [20]. They work well in these environments as they have very nice algorithmic properties. However, these properties do not provide an intuitive mathematical framework.

Fraenkel Mostowski set theory was one of the first serious attempts to fomalise nominal logic. It is standard ZF set theory but with an extra freshness axiom added. In [4], Gabbay formalises a portion of the π-calculus in FM. 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 without much support for anything. This choice of framework was necessary since Isabelle/HOL contains the axiom of choice which is inconsistent with FM.

Higher order abstract syntax (HOAS) is the approach most similar to ours. It has been used to model the π-calculus in both Coq [6], by Honsell et. al., and in Isabelle by Röckl and Hirschkoff [15]. 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 [15], 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 [6] 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 to worry about 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.

8.3 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 transfered to many other calculi.

Acknowledgements. We would like to thank Stefan Berghofer for his generous help with the understanding of 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.

References

1. 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.
2. Jesper Bengtson. Generic implementations of process calculi in Isabelle. In *The 16th Nordic Workshop on Programming Theory (NWPT'04)*, pages 74–78, 2004.
3. M. J. Gabbay. A theory of inductive definitions with α-equivalence, PhD thesis, University of Cambridge, 2000.
4. M. J. Gabbay. The π-calculus in FM. In Fairouz Kamareddine, editor, *Thirty-five years of Automath*. Kluwer, 2003.
5. M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13:341–363, 2001.
6. Furio Honsell, Marino Miculan, and Ivan Scagnetto. π-calculus in (co)inductive type theory. *Theoretical Computer Science*, 253(2):239–285, 2001.
7. Thomas F. Melham. A mechanized theory of the pi-calculus in HOL. *Nordic Journal of Computing*, 1(1):50–76, 1994.

8. R. Milner. *A Calculus of Communicating Systems.* Number 92 in LNCS. Springer-Verlag, 1980.
9. Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I/II. *Inf. Comput.*, 100(1):1–77, 1992.
10. 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.
11. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic.* Springer-Verlag, 2002.
12. Joachim Parrow. An introduction to the pi-calculus. In *Handbook of Process Algebra*, pages 479–543. Elsevier, 2001.
13. A. M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 186:165–193, 2003.
14. A. M. Pitts. Alpha-structural recursion and induction. *Journal of the ACM*, 53:459–506, 2006.
15. Christine Röckl and Daniel Hirschkoff. A fully adequate shallow embedding of the π-calculus in Isabelle/HOL with mechanized syntax analysis. *J. Funct. Program.*, 13(2):415–451, 2003.
16. C. Urban and S. Berghoffer. A recursion combinator for nominal datatypes implemented in Isabelle/HOL, Accepted to IJCAR 2006.
17. C. Urban, A. M. Pitts, and M. J. Gabbay. Nominal unification. *Theoretical Computer Science*, 323:473–497, 2004.
18. Christian Urban and Michael Norrish. A formal treatment of the barendregt variable convention in rule inductions. In *MERLIN '05: Proceedings of the 3rd ACM SIGPLAN workshop on Mechanized reasoning about languages with variable binding*, pages 25–32, New York, NY, USA, 2005. ACM Press.
19. Christian Urban and Christine Tasson. Nominal techniques in Isabelle/HOL. In *CADE*, pages 38–53, 2005.
20. 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.
21. Markus Wenzel. Isar - a generic interpretative approach to readable formal proof documents. In *Theorem Proving in Higher Order Logics*, pages 167–184, 1999.