# A POLYNOMIAL TRANSLATION OF π-CALCULUS FCPS TO SAFE PETRI NETS [*]

VICTOR KHOMENKO [a], ROLAND MEYER [b], AND REINER HÜCHTING [c]

[a] Newcastle University, UK
  *e-mail address*: Victor.Khomenko@ncl.ac.uk

[b,c] University of Kaiserslautern, Germany
  *e-mail address*: {Meyer,Huechting}@cs.uni-kl.de

ABSTRACT. We develop a polynomial translation from finite control π-calculus processes to safe low-level Petri nets. To our knowledge, this is the first such translation. It is natural in that there is a close correspondence between the control flows, enjoys a bisimulation result, and is suitable for practical model checking.

## 1. INTRODUCTION

Many contemporary systems – be it software, hardware, or network applications – support functionality that significantly increases their power, usability, and flexibility:

**Interaction**   Mobile systems permeate our lives and are becoming ever more important. Indeed, the vision of pervasive computing, where devices like mobile phones and laptops are opportunistically engaged in interaction with a user and with each other, is quickly becoming a reality.

**Reconfiguration**   Systems implement a flexible interconnection structure, e.g. cores in Networks-on-Chip temporarily shut down to save power, resilient systems continue to deliver (reduced) functionality even if some of their modules develop faults, and ad-hoc networks build-up and destroy connections between devices at runtime.

**Resource allocation**   Many systems maintain multiple instances of the same resource (e.g. network servers, or processor cores in a microchip) that are dynamically allocated to tasks depending on the workload, power mode, and priorities of the clients.

The common feature of such systems is the possibility to form *dynamic* connections between individual modules. It is implemented using reference passing: a module can become aware of another module by receiving a reference (e.g. in form of a network address) to it. This reference enables subsequent communication between the modules, and can be understood as a dynamically created connection. We will refer to this class of systems as *Reference Passing Systems (RPS)*.

As people are increasingly dependent on the correct functionality of Reference Passing Systems, the costs incurred by design errors can be extremely high. However, even the conventional concurrent systems are notoriously difficult to design correctly because of the complexity of their synchronisation behaviour. Reference passing adds another layer of complexity by making the interconnect structure dynamic. Hence, formal verification has to be employed to ensure the correct behaviour of RPSs.

While the complexity of systems increases, the time-to-market is reducing. To address this, system design has changed from a holistic to a compositional process: the system is usually composed from pre-existing modules. This change in the design process has to be mirrored in the verification process. *Verification has to focus on the inter-modular level rather than on individual modules.* Indeed, it is reasonable nowadays to assume that individual modules are well-tested or formally verified by their vendors. Moreover, the inter-module communication fabric (e.g. a computer network) is usually built of standard components and uses standard protocols, and so can be assumed to be correct. This means bugs primarily arise in the interaction between modules, and here verification is required to ensure correctness of the *system as a whole.*

When we verify the interaction between modules, we take advantage of this *separation of verification concerns.* We only have to model the modules' interfaces but can abstract away from their internal behaviour as well as low-level communication concerns (e.g. network behaviour), which we assume to be checked by the vendor. Traditionally, the interface behaviour is verified using rely/guarantee techniques, potentially supported by theorem provers. Due to undecidability reasons, however, theorem proving generally requires substantial manual intervention to discharge some of the proof obligations. In the interest of a short time to market, our ambition is to lift fully automatic model checking to the inter-modular level.

There are several formalisms suitable for modelling the interface behaviour of RPSs. The main considerations in choosing such a formalism are its expressiveness and the tractability of the associated verification problems. Expressive formalisms like $\pi$-calculus [20] and Ambient Calculus [5] are Turing powerful and so undecidable in general. Fortunately, the ability to pass references *per se* does not lead to undecidability. One can impose restrictions on the communication capabilities [1,16], control flow [6,24], or interconnection shape [14,16] to recover decidability while retaining a reasonable modelling power.

*Finite Control Processes (FCP)* [6] are a fragment of $\pi$-calculus where the system is modelled as a parallel composition of a fixed number of sequential entities (threads). The control of each thread is represented by a finite automaton, and the number of threads is bounded in advance. The threads communicate synchronously via channels that they create, exchange and destroy at runtime. These capabilities are often sufficient to faithfully model the interface behaviour of RPSs. The appeal of FCPs is in combining this modelling power with decidability of verification [6,17].

In this paper, we contribute to FCP verification, following an established approach. We translate the FCP under study into a safe low-level Petri net (PN). This translation bridges

the gap between expressiveness and verifiability: While π-calculus is suitable for modelling RPSs but difficult to verify due to the complicated semantics, PNs are a low-level formalism equipped with efficient analysis algorithms. With the translation, all verification techniques and tools that are available for PNs can be applied to analyse the (translated) process.

1.1. **A polynomial translation.** There is a large body of literature on π-calculus to Petri net translations (cf. Sect. 1.2 for a detailed discussion). Complexity-theoretic considerations, however, suggest that they are all suboptimal for FCPs — either in terms of size [3, 11, 16, 17] or because of a too powerful target formalism [1, 7, 12]. The following shows that a polynomial translation of FCPs into low-level safe PNs must exist.

Since the state of an FCP can always be described by a string of length linear in the process size, an FCP can be simulated by a Turing machine (TM) with a tape of linear length. Moreover, a TM with bounded tape can be modelled by a safe PN of polynomial size (in the size of the control and the tape length), see e.g. [9]. Finally, a linear translation from safe PNs to FCPs is described in Sect. 10. That is, the three formalisms can simulate each other with only polynomial overhead. This argument is in fact constructive, and shows PSPACE-completeness of FCP model checking. Even more, it shows that we can use safe PNs to verify FCPs. The problem with this translation via TMs is that the safe PN resulting from an FCP would be large and contrived, and thus of limited use for practical verification.

These considerations motivated us to look for a direct polynomial translation of FCPs to safe PNs, which is the main contribution of this paper. We stress that our translation is not just a theoretical result, but is also quite practical:

- it is natural in the sense that there is a strong correspondence between the control flow of the FCP and the resulting PN;
- the transition system of the FCP and that of its PN representation are bisimilar, which makes the latter suitable for checking temporal properties of the former;
- the resulting PN is compact (polynomial even in the worst case);
- we propose several optimisations that significantly reduce the PN's size in practice;
- we propose several extensions of the translation, in particular to polyadic communication, polyadic synchronisation, and match/mismatch operators;
- experiments demonstrate that the translation is suitable for practical verification.

Technically, our translation relies on three insights: (i) the behaviour of an FCP $\nu r.(S_1 \mid S_2)$ coincides with the behaviour of $(S_1\{n/r\} \mid S_2\{n/r\})$ where the restricted name $r$ has been replaced by a fresh public name $n$ (a set of fresh names that is linear in the size of the FCP will be sufficient); (ii) we have to recycle fresh names, and so implement reference counters for them; and (iii) we hold substitutions explicit and give them a compact representation using decomposition, e.g., $\{a, b/x, y\}$ into $\{a/x\}$ and $\{b/y\}$.

1.2. **Related work.** There are two main approaches to FCP verification. The first is to directly generate the state space of the model, as is done (on-the-fly) by the Mobility Workbench (MWB) [26]. This approach is relatively straightforward but has a number of disadvantages. In particular, its scalability is poor due to the complexity of the π-calculus semantics, which restricts the use of heuristics for pruning the state space, and due to the need for expensive operations (like equivalence checks [13]) every time a new state is generated. Furthermore, some efficient model checking techniques like symbolic representations of state spaces are difficult to apply.

The alternative approach, and the one followed here, is to translate a $\pi$-calculus term into a simpler formalism, e.g. a Petri net, that is then analysed. This method does not depend on a concrete verification technique for $\pi$-calculus but can adapt any such technique for PNs. In particular, RPSs often are highly concurrent, and so translating them into a true concurrency formalism like PNs opens the door for partial-order reductions in verification. This alleviates the problem of combinatorial state space explosion, that is, a small specification often has a huge number of reachable states that is beyond the capabilities of existing computers.

Although several translations of $\pi$-calculus to Petri nets have been proposed, none of them provides a polynomial translation of FCPs to safe PNs. The verification kit HAL [10] translates a model into a *History Dependent Automaton* — a finite automaton where states are labelled by sets of names that represent restrictions [21, 24]. For model checking, these automata are further translated to finite automata [10]. Like in our approach, the idea is to replace restrictions with fresh names, but the translation stores full substitutions, which may yield an exponential blow up of the finite automaton. Our translation avoids this blow up by compactly representing substitutions by PN markings. This, however, needs careful substitution manipulation and reference counting.

Amadio and Meyssonnier [1] replace unused restricted names by generic free names. Their translation instantiates substitutions, e.g. $(\overline{x_1}\langle y_1\rangle.\overline{x_2}\langle y_2\rangle)\{a, b, a, b/x_1, y_1, x_2, y_2\}$ is represented by $\overline{a}\langle b\rangle.\overline{a}\langle b\rangle$. This creates an exponential blow up: since the substitutions change over time, $m$ public names and $n$ variables may yield $m^n$ instantiated terms. Moreover, since the number of processes to be modified by replacement is not bounded in [1], Amadio and Meyssonnier use PNs with transfer. (Their translation handles a subset of $\pi$-calculus incomparable with FCPs.) As this paper shows, transfer nets are an unnecessarily powerful target formalism for FCPs — e.g. reachability is undecidable in such nets [8].

Busi and Gorrieri study non-interleaving and causal semantics for the $\pi$-calculus and provide decidability results for model checking [3]. (That work has no bisimilarity proof, which is provided in [11].) The translations may be exponential for FCPs, again due to the instantiation of substitutions.

Devillers, Klaudel and Koutny [7] achieve a bisimilar translation of $\pi$-calculus into high-level Petri nets, thus using a Turing complete target formalism where automatic analyses are necessarily incomplete. In [12], this translation is used for unfolding-based model checking; to avoid undecidability, the processes are restricted to be recursion-free — a class of limited practical applicability.

Peschanski, Klaudel and Devillers [23] translate $\pi$-graphs (a graphical variant of $\pi$-calculus) into high-level PNs. The technique works on a fragment that is equivalent to FCPs. However, the target formalism is unnecessarily powerful, and the paper provides no experimental evaluation.

Our earlier translation [16] identifies groups of processes that share restricted names. In [17], we modify it to generate safe low-level PNs, and use an unfolding-based model checker. The experiments indicate that this technique is more scalable than the ones above, and it has the advantage of generating low-level rather than high-level PNs. However, the PN may still be exponentially large.

## 2. BASIC NOTIONS

2.1. **Petri nets.** A *Petri net* (PN) is a tuple $N \stackrel{\mathrm{df}}{=} (P, T, F, M_0)$ such that $P$ and $T$ are disjoint sets of *places* and *transitions*, $F \subseteq (P \times T) \cup (T \times P)$ is a *flow relation*, and $M_0$ is the *initial marking* of $N$, where a *marking* $M : P \to \mathbb{N} \stackrel{\mathrm{df}}{=} \{0, 1, 2, \ldots\}$ is a multiset of places in $N$. We draw PNs in the standard way: places are represented as circles, transitions as boxes, the flow relation by arcs, and a marking by tokens within circles. The *size* of $N$ is

$$\|N\| \stackrel{\mathrm{df}}{=} |P| + |T| + |F| + |M_0|.$$

We denote by $\,^\bullet z \stackrel{\mathrm{df}}{=} \{y \mid (y, z) \in F\}$ and $z^\bullet \stackrel{\mathrm{df}}{=} \{y \mid (z, y) \in F\}$ the *preset* and *postset* of $z \in P \cup T$, respectively. A transition $t$ is *enabled at marking* $M$, denoted by $M \stackrel{t}{\to}$, if $M(p) > 0$ for every $p \in \,^\bullet t$. Such a transition can *fire*, leading to the marking $M'$ with

$$M'(p) \stackrel{\mathrm{df}}{=} M(p) - F(p, t) + F(t, p) \quad \text{for every } p \in P.$$

We denote the firing relation by $M \stackrel{t}{\to} M'$ or by $M \to M'$ if the identity of the transition is irrelevant. The set of *reachable markings of* $N$ is denoted by $Reach(N)$. The *transition system* of $N$ is

$$\mathcal{T}(N) \stackrel{\mathrm{df}}{=} (Reach(N), \to, M_0).$$

A PN $N$ is *k-bounded* if $M(p) \le k$ for every $M \in Reach(N)$ and every place $p \in P$, and *safe* if it is 1-bounded. This paper focuses on safe PNs. A set of places in a PN is *mutually exclusive* if at any reachable marking at most one of them contains tokens. A place $p$ is a *complement* of a set $Q$ of mutually exclusive places if at any reachable marking $p$ contains a token iff none of the places in $Q$ contains a token. If $Q = \{q\}$, the places $p$ and $q$ are *complements* of each other.

2.2. **Finite control processes.** In $\pi$-calculus [19, 25], threads communicate via synchronous message exchange. The key idea in the model is that messages and the channels they are sent on have the same type: they are just *names* from some set $\Phi \stackrel{\mathrm{df}}{=} \{a, b, x, y, i, f, r, \ldots\}$. This means a name that has been received as a message in one communication may serve as a channel in a later interaction. To communicate, processes consume *prefixes*

$$\pi ::= \overline{a}\langle b \rangle \ \mid\ a(x) \ \mid\ \tau.$$

The *output prefix* $\overline{a}\langle b \rangle$ sends name $b$ along channel $a$. The *input prefix* $a(x)$ receives a name that replaces $x$ on channel $a$. Prefix $\tau$ stands for a *silent action*.

*Threads*, also called *sequential processes*, are constructed as follows. A *choice process* $\sum_{i \in I} \pi_i.S_i$ over a finite set of indices $I$ executes a prefix $\pi_i$ and then behaves like $S_i$. The special case of choices over an empty index set $I = \emptyset$ is denoted by $\mathbf{0}$. This process terminates a thread. A *restriction* $\nu r.S$ generates a name $r$ that is different from all other names in the system. To implement parameterised recursion, we use *calls to process identifiers* $K\lfloor \tilde{a} \rfloor$. We defer their explanation for the moment. To sum up, threads take the form

$$S ::= K\lfloor \tilde{a} \rfloor \ \mid\ \sum_{i \in I} \pi_i.S_i \ \mid\ \nu r.S.$$

We use $\mathcal{S}$ to refer to the set of all threads. A *finite control process (FCP)* $F$ is a *parallel composition* of a fixed number of threads:

$$F ::= \nu\tilde{r}.(S_{init,1} \mid \ldots \mid S_{init,n}).$$

Here, $\nu\tilde{r}$ with $\tilde{r} = r_1\ldots r_k$ denotes a (perhaps empty) sequence of restrictions $\nu r_1\ldots\nu r_k$. We use $P$ to refer to an arbitrary process, be it a thread, an FCP, or a term obtained by structural congruence defined below. We denote iterated parallel compositions by $\prod_{i\in I} P_i$.

Our presentation of parameterised recursion using calls $K\lfloor\tilde{a}\rfloor$ follows [25]. Process identifiers $K$ are taken from some set $\Psi \stackrel{\text{df}}{=} \{H, K, L, \ldots\}$ and have a *defining equation* $K(\tilde{f}) := S$. Thread $S$ can be understood as the implementation of $K$. The process has a list of distinct *formal parameters* $\tilde{f} = f_1, \ldots, f_k$ that are replaced by *factual parameters* $\tilde{a} = a_1, \ldots, a_k$ when a call $K\lfloor\tilde{a}\rfloor$ is executed. Note that $\tilde{a}$ and $\tilde{f}$ have the same length. When talking about an *FCP specification* $F$, we mean process $F$ with all its defining equations.

To implement the replacement of formal parameters $\tilde{f}$ by $\tilde{a}$ in calls to process identifiers, we use *substitutions*. A substitution is a function $\sigma : \Phi \to \Phi$ that maps names to names. If we make domain and codomain explicit, $\sigma : A \to B$ with $A, B \subseteq \Phi$, we require $\sigma(a) \in B$ for all $a \in A$ and $\sigma(x) = x$ for all $x \in \Phi \setminus A$. We use $\{\tilde{a}/\tilde{f}\}$ to denote the substitution $\sigma : \tilde{f} \to \tilde{a}$ with $\sigma(f_i) \stackrel{\text{df}}{=} a_i$ for $i \in \{1, \ldots, k\}$. The *application of substitution* $\sigma$ *to* $S$ is denoted by $S\sigma$ and defined in the standard way [25].

Input prefix $a(i)$ and restriction $\nu r$ *bind* the names $i$ and $r$, respectively. The *set of bound names* in a process $P$ is $bn(P)$. A name which is not bound is *free*, and the *set of free names* in $P$ is $fn(P)$. We permit $\alpha$-conversion of bound names. Therefore, wlog., we make the following assumptions common in $\pi$-calculus theory and collectively referred to as *no clash* (**NC**) henceforth. For every FCP specification $F$, we require that: (i) a name is bound at most once, a name $f$ is used at most once in a formal parameter list, bound and free names are disjoint, bound names and formal parameters are disjoint, formal parameters and free names in $F$ are disjoint; and (ii) if $\sigma = \{\tilde{a}/\tilde{x}\}$ is applied to $S$ then $bn(S) \cap (\tilde{a} \cup \tilde{x}) = \emptyset$.

Assuming (**NC**), the names in an FCP specification $F$ and the corresponding defining equations can be partitioned into the following sets: set $\mathcal{R}$ of names bound by restriction operators, set $\mathcal{I}$ of names bound by input prefixes, set $\mathcal{F}$ of names used as formal parameters in defining equations, and set $\mathcal{P}$ of all the other names — they are called *public*.

We are interested in the relation between the size of an FCP specification and the size of its PN representation. The *size* of an FCP specification is defined as the size of its initial term plus the sizes of the defining equations:

$$\|\mathbf{0}\| \stackrel{\text{df}}{=} 1 \qquad \|\sum_{i\in I}\pi_i.S_i\| \stackrel{\text{df}}{=} 3|I|-1+\sum_{i\in I}\|S_i\| \qquad \|\nu r.P\| \stackrel{\text{df}}{=} 1+\|P\| \qquad \|K\lfloor\tilde{a}\rfloor\| \stackrel{\text{df}}{=} 1+|\tilde{a}|$$
$$\|S_{Init,1} \mid \ldots \mid S_{Init,n}\| \stackrel{\text{df}}{=} n-1+\sum_{1\le i\le n}\|S_{Init,i}\| \qquad \|K(\tilde{f}) := S\| \stackrel{\text{df}}{=} 1+|\tilde{f}|+\|S\|.$$

Intuitively, we count the operations, process identifiers, and names in each element of the specification, e.g. the factor 3 in $3|I|$ refers to a send or receive prefix of size two, followed by a plus.

To define the behaviour of a process, we rely on the *structural congruence* relation $\equiv$ on process terms. It is the smallest congruence satisfying the axioms in Fig. 1, i.e. $\alpha$-conversion of bound names is allowed, $\mid$ is commutative and associative with $\mathbf{0}$ as the neutral element, and for choices associativity and commutativity are enforced by the notation $\sum_{i\in I}\pi_i.S_i$,

$$\nu r.P \equiv \nu r'.P\{r'/r\} \text{ if } r' \notin \mathit{fn}\,(P) \qquad a(x).S \equiv a(x').S\{x'/x\} \text{ if } x' \notin \mathit{fn}\,(S)$$

$$P_1 \mid P_2 \equiv P_2 \mid P_1 \qquad (P_1 \mid P_2) \mid P_3 \equiv P_1 \mid (P_2 \mid P_3) \qquad P \mid \mathbf{0} \equiv P$$

$$\nu r.\mathbf{0} \equiv \mathbf{0} \qquad \nu r_1.\nu r_2.P \equiv \nu r_2.\nu r_1.P \qquad \nu r.(P_1 \mid P_2) \equiv P_1 \mid \nu r.P_2 \text{ if } r \notin \mathit{fn}\,(P_1)$$

Figure 1: Axioms defining structural congruence $\equiv$.

$$\text{(Tau) } \tau.S + \ldots \to S \qquad \text{(Const) } K\lfloor \tilde{a} \rfloor \to S\{\tilde{a}/\tilde{x}\} \text{ if } K(\tilde{x}) := S$$

$$\text{(React) } x(y).S_1 + \ldots \mid \overline{x}\langle z \rangle.S_2 + \ldots \to S_1\{z/y\} \mid S_2$$

$$\text{(Par) } \frac{P_1 \to P_1'}{P_1 \mid P_2 \to P_1' \mid P_2} \qquad \text{(Res) } \frac{P \to P'}{\nu a.P \to \nu a.P'} \qquad \text{(Struct) } \frac{P_1 \equiv P_1' \to P_2' \equiv P_2}{P_1 \to P_2}$$

Figure 2: Axioms and rules defining reaction relation $\to$.

$\nu$ can be eliminated when applied to $\mathbf{0}$, is commutative, and its scope can be extended to include a concurrent process not containing free occurrences of the bound name.

The behaviour of $\pi$-calculus processes is determined by *reaction relation* $\to$ between terms [19, 25], see Fig 2. It has the axioms for consuming silent prefixes, identifier calls and communications, and is defined to be closed under $\mid$, $\nu$, and $\equiv$. By $\mathit{Reach}(F)$ we denote the set of *all processes reachable from $F$*. The *transition system* of $F$ factorises the reachable processes along structural congruence,

$$\mathcal{T}(F) \overset{\text{df}}{=} (\mathit{Reach}(F)/_{\equiv}, \hookrightarrow, \underline{F}),$$

where $\underline{F}$ denotes the congruence class of $F$ wrt. $\equiv$ and $\underline{F_1} \hookrightarrow \underline{F_2}$ if $F_1 \to F_2$. That is, structurally congruent processes are collapsed into a single state, and the transition relation is amended appropriately.

*Normal form assumptions.* To ease the definition of the translation and the corresponding correctness proofs, we make assumptions about the shape of the FCP specification. These assumptions are not restrictive, as any FCP can be translated into the required form. First, we require that the sets of identifiers called by different threads (both directly from $F$ and indirectly from defining equations) are disjoint. This ensures that the threads have disjoint descriptions of their control flows and corresponds to the notion of a *safe* FCP in [17]. The assumption can be achieved by replicating some defining equations. The resulting specification $F'$ is bisimilar with $F$ and has size $O(n\|F\|) = O(\|F\|^2)$, where $n$ is the number of threads. We illustrate the construction on the FCP specification $F = K\lfloor a, b \rfloor \mid L\lfloor a, c \rfloor$ (left) together with its replicated version $F' = K^1\lfloor a, b \rfloor \mid L^2\lfloor a, c \rfloor$ (right):

$$K(f_1, f_2) := \tau.L\lfloor f_1, f_2 \rfloor \qquad\qquad K^1(f_1^1, f_2^1) := \tau.L^1\lfloor f_1^1, f_2^1 \rfloor$$

$$L(f_3, f_4) := \tau.K\lfloor f_3, f_4 \rfloor \qquad\qquad L^1(f_3^1, f_4^1) := \tau.K^1\lfloor f_3^1, f_4^1 \rfloor$$

$$K^2(f_1^2, f_2^2) := \tau.L^2\lfloor f_1^2, f_2^2 \rfloor$$

$$L^2(f_3^2, f_4^2) := \tau.K^2\lfloor f_3^2, f_4^2 \rfloor.$$

We can also ensure that defining equations do not call themselves, i.e. that the body of $K(\tilde{f}) := S$ does not contain any calls of the form $K\lfloor \tilde{a} \rfloor$. Indeed, we can replace any such

call with $K'\lfloor\tilde{a}\rfloor$, using a new defining equation $K'(\tilde{f}') := K\lfloor\tilde{f}'\rfloor$. This increases the size of the FCP only linearly, and ensures we do not have to remap parts of $\tilde{f}$ to $\tilde{f}$ when passing the parameters of a call, which simplifies the translation.

## 3. Principles of the translation

This section informally explains the polynomial translation of FCPs to safe PNs. The main idea is to replace restricted names by fresh public ones. Indeed, $F = \nu\tilde{r}.(S_1 \mid \ldots \mid S_n)$ behaves like $S_1\{\tilde{n}/\tilde{r}\} \mid \ldots \mid S_n\{\tilde{n}/\tilde{r}\}$, provided the names $\tilde{n}$ are fresh. These new names are picked from a set $\mathcal{N}$, and since for FCP specifications there is a bound on the number of restricted names in all processes reachable from $F$, a finite $\mathcal{N}$ suffices. But how to support name creation and deletion with a constant number of free names? The trick is to reuse the names: $n \in \mathcal{N}$ may first represent a restricted name $r_1$ and later a different restricted name $r_2$. To implement this recycling of names, we keep track of whether or not $n \in \mathcal{N}$ is currently used in the process. This can be understood as reference counting.

The translation takes the set of names $\mathcal{N}$ as a parameter. Already a fairly large set $\mathcal{N}_{\mathcal{RIF}}$ of cardinality $|\mathcal{R}| + |\mathcal{I}| + |\mathcal{F}|$ is sufficient to prove polynomiality of the translation. The rationale is that there should be enough values to map each bound name to a unique value. Indeed, one can provide a *domain function* $dom : \mathcal{P} \cup \mathcal{R} \cup \mathcal{I} \cup \mathcal{F} \to 2^{\mathcal{P} \cup \mathcal{N}}$ that gives, for each name $x$ of $F$, an overapproximation of the set of possible values of $x$. The rough overapproximation above employs

$$dom_{\mathcal{RIF}}(x) \overset{\mathrm{df}}{=} \left\{ \begin{array}{ll} \{x\} & \text{if } x \in \mathcal{P} \\ \mathcal{N}_{\mathcal{RIF}} & \text{if } x \in \mathcal{R} \\ \mathcal{P} \cup \mathcal{N}_{\mathcal{RIF}} & \text{if } x \in \mathcal{I} \cup \mathcal{F}. \end{array} \right.$$

We explain how to compute better domains by static analysis in Sect. 7.

The translation is then defined to be the composition

$$N(F) \overset{\mathrm{df}}{=} N_{Subst} \lhd H(N(S_1) \parallel \ldots \parallel N(S_n)).$$

The first net $N_{Subst}$ compactly represents substitutions $\sigma : \mathcal{R} \cup \mathcal{I} \cup \mathcal{F} \to \mathcal{P} \cup \mathcal{N}$ and implements reference counting. It only consists of places and is detailed in Sect. 3.1. The second net $H(N(S_1) \parallel \ldots \parallel N(S_n))$ represents the control flow of the FCP. Each net $N(S_i)$ is a finite automaton that reflects the control flow of thread $S_i$. Importantly, the transitions of $N(S_i)$ are annotated with synchronisation actions and sets of commands that explicitly handle the introduction and removal of name bindings. Parallel composition $\parallel$ synchronises the subnets of all threads. The operator places the nets side by side and merges pairs of transitions with complementary synchronisation actions $send(a, b)$ and $rec(a, b)$. Hiding $H$ then removes the original transitions. After hiding, transitions in the control flow net only carry name binding commands. The implementation operator $\lhd$ implements them by adding arcs between the control flow net and $N_{Subst}$. We elaborate on the construction of the control flow in Sect. 3.2, and Sect. 3.3 illustrates the translation on an example.

3.1. **Petri net representation of substitutions.** A substitution $\sigma : \mathcal{R} \cup \mathcal{I} \cup \mathcal{F} \to \mathcal{P} \cup \mathcal{N}$ maps the bound names and formal parameters occurring in the FCP to their values. The compact PN representation of substitutions is a key element of the proposed translation. It should efficiently support the following operations:

| | $p_1$ | $p_2$ | $\ldots$ | | $n_1$ | $n_2$ | $\ldots$ |
|---|---|---|---|---|---|---|---|
| $i_1$ | ○$[i_1{=}p_1]$ | ○$[i_1{=}p_2]$ | $\ldots$ | | ○$[i_1{=}n_1]$ | ⊙$[i_1{=}n_2]$ | $\ldots$ |
| | | | | | ⊙$[i_1{\neq}n_1]$ | ○$[i_1{\neq}n_2]$ | $\ldots$ |
| $i_2$ | ○$[i_2{=}p_1]$ | ○$[i_2{=}p_2]$ | $\ldots$ | | ○$[i_2{=}n_1]$ | ○$[i_2{=}n_2]$ | $\ldots$ |
| | | | | | ⊙$[i_2{\neq}n_1]$ | ⊙$[i_2{\neq}n_2]$ | $\ldots$ |
| $\vdots$ | $\vdots$ | $\vdots$ | $\ddots$ | | $\vdots$ | $\vdots$ | $\ddots$ |
| $f_1$ | ○$[f_1{=}p_1]$ | ○$[f_1{=}p_2]$ | $\ldots$ | | ⊙$[f_1{=}n_1]$ | ○$[f_1{=}n_2]$ | $\ldots$ |
| | | | | | ○$[f_1{\neq}n_1]$ | ⊙$[f_1{\neq}n_2]$ | $\ldots$ |
| $f_2$ | ○$[f_2{=}p_1]$ | ○$[f_2{=}p_2]$ | $\ldots$ | | ○$[f_2{=}n_1]$ | ○$[f_2{=}n_2]$ | $\ldots$ |
| | | | | | ⊙$[f_2{\neq}n_1]$ | ⊙$[f_2{\neq}n_2]$ | $\ldots$ |
| $\vdots$ | $\vdots$ | $\vdots$ | $\ddots$ | | $\vdots$ | $\vdots$ | $\ddots$ |
| $r_1$ | restricted names are never mapped | | | | ○$[r_1{=}n_1]$ | ○$[r_1{=}n_2]$ | $\ldots$ |
| $r_2$ | to public ones, so no places here | | | | ○$[r_2{=}n_1]$ | ○$[r_2{=}n_2]$ | $\ldots$ |
| $\vdots$ | | | | | $\vdots$ | $\vdots$ | $\ddots$ |
| | | | | | ⊙$[r_*{\neq}n_1]$ | ⊙$[r_*{\neq}n_2]$ | $\ldots$ |

Figure 3: Illustration of $N_{Subst}$ with a substitution marking that corresponds to $\sigma$ : $\{i_1, f_1\} \to \tilde{r} \cup \mathcal{P}$ where $\sigma(i_1) = r_1$ and $\sigma(f_1) = r_2$ with $r_1 \neq r_2$. The marking represents $r_1$ by $n_2$ and $r_2$ by $n_1$.

**Initialisation of a restricted name**   It should be possible to find a value $val \in \mathcal{N}$ to which no bound name or formal parameter is currently mapped, and map a given restricted name $r$ to $val$.

**Remapping**   When name $v$ is communicated to a thread, some input name $i$ has to be mapped to $\sigma(v)$. Similarly, a formal parameter $f$ has to be mapped to $\sigma(v)$ during a process call that uses $v$ as a factual parameter. Since $v$ can occur several times in the list of factual parameters, one should be able to map several formal parameters to $\sigma(v)$ in one step, i.e. by one PN transition.

**Unmapping**   When a bound name or formal parameter $v$ is forgotten, its mapping has to be removed. During a process call, it often happens that $\sigma(v)$ is assigned to one or more formal parameters and simultaneously $v$ is forgotten. Therefore, it is convenient to be able to remap and unmap $v$ in one step.

Ideally, the three kinds of operations described above should not interfere when applied to names in distinct threads, so that they can be performed concurrently; note that due to **(NC)** the bound names and formal parameters are always different. This prevents the introduction of arbitration, and so has a beneficial effect on the performance of some model checking methods (e.g. those using partial order techniques).

In what follows, we describe a representation of substitutions that satisfies all the formulated requirements. This safe PN, which is depicted in Fig. 3, only consists of places. A place $[var = val]$, when marked, represents the fact that $var \in \mathcal{R} \cup \mathcal{I} \cup \mathcal{F}$ is mapped to $val \in \mathcal{P} \cup \mathcal{N}$. Our translation will maintain the following invariants. (i) For each $var \in \mathcal{I} \cup \mathcal{F}$ and $val \in \mathcal{N}$, place $[var \neq val]$ is complementary to $[var = val]$. (ii) For each $val \in \mathcal{N}$, the places $[r_1 = val], [r_2 = val], \ldots$ are mutually exclusive, so that no two restricted names can be mapped to the same value. Moreover, $[r_* \neq val]$ is complementary to all these places.

(iii) For each $var \in \mathcal{R} \cup \mathcal{I} \cup \mathcal{F}$, the places $[var = val]$ are mutually exclusive (i.e. a name can be mapped to at most one value), where $val$ runs through $dom(var)$.

The choice of the cardinality of $\mathcal{N}$ is of crucial importance. As explained above, it should be sufficiently large to guarantee that there will always be a name that can be used to initialise a restriction. But taking an unnecessary big value for this parameter increases the size of the PN as well as the number of reachable states.

The operations on the substitution are implemented as follows:

**Initialisation of a restricted name**   To find a value $val \in \mathcal{N}$ that is not referenced, and to map a given restricted name $r$ to $val$, the transition has to:
- test by read arcs that the places $[i_1 \neq val], [i_2 \neq val], \ldots$ and $[f_1 \neq val], [f_2 \neq val], \ldots$ have tokens (i.e. no input name or formal parameter is currently mapped to $val$);
- consume the token from $[r_* \neq val]$ (checking thus that no restricted name is currently mapped to $val$);
- produce a token at $[r = val]$ (mapping thus $r$ to $val$).

**Remapping**   When a communication binds the value of a name $v$ to an input name $i$, the corresponding transition consumes the token from $[i \neq \sigma(v)]$ (provided $\sigma(v) \in \mathcal{N}$) and produces a token in $[i = \sigma(v)]$. In the case of identifier calls, $v$ may occur several times in the list of factual parameters, and so several formal parameters $f_{k_1}, \ldots, f_{k_l}$ have to be bound to $\sigma(v)$. This can be handled by a single transition consuming the tokens from $[f_{k_j} \neq \sigma(v)]$ and producing tokens in $[f_{k_j} = \sigma(v)]$ for $j \in \{1, \ldots, l\}$. The same transition can unmap $v$ if necessary.

**Unmapping**   When a bound name or formal parameter $var$ that is mapped to $val$ is forgotten, the mapping should be removed. This is modelled by a transition consuming the token from $[var = val]$ and, if $val \in \mathcal{N}$, producing a token in $[var \neq val]$ (if $var \in \mathcal{I} \cup \mathcal{F}$) or $[var_* \neq val]$ (if $var \in \mathcal{R}$).

3.2. **Petri net representation of the control flow.** We elaborate on the translation of a thread $S_i$ into the net $N(S_i)$ that reflects the control flow. Each subterm $st$ of $S_i$ corresponds to a subnet of $N(S_i)$ with a unique entry place $p_{st}$. This place is initially marked if $st$ corresponds to the thread's initial expression. The communication prefixes are modelled by transitions. These transitions are labelled with synchronisation actions as well as commands to explicitly introduce and remove name bindings in the substitution net. At this point, however, the transitions are just stubs. The synchronisation between threads is introduced by parallel composition $\|$, the synchronisation between threads and substitution net is performed by the implementation operator $\triangleleft$.

The subnets corresponding to terms $st$ are defined as follows (note that each thread is a sequential process, so $|$ does not occur in the term):

**Initialisation of restricted names**   The subnet corresponding to $\nu r.S$ maps $r$ to some currently unused $n \in \mathcal{N}$, cf. Fig. 5. More precisely, for each $n \in \mathcal{N}$ we create a transition $t_n^r$ consuming a token from the entry place of $\nu r.S$, producing a token in the entry place of the subnet implementing $S$, and performing the initialisation of the restricted name $r$ with the value $n$ as explained in Sect. 3.1. Note that the transitions $t_n^r$ arbitrate between the names in $\mathcal{N}$, allowing any of the currently unused names to be selected for the initialisation of $r$.

If such an arbitration is undesirable,[1] separate pools of values can be used for each thread, as described in Sect. 7.

**Handling 0**   The subnet for **0** is comprised of the entry place only, which means an execution of **0** terminates a thread. An alternative for implementing termination is to also unmap all the bound names and formal parameters in whose scope this **0** resides. We note that this unmapping is not necessary, as the used resources (in particular, the values from $\mathcal{N}$ to which these variables are mapped) will not be needed.

**Handling calls**   Consider $K\lfloor\tilde{a}\rfloor$ with $K(\tilde{f}) := S$. The entry place of $K\lfloor\tilde{a}\rfloor$ is followed by a subnet that maps $\tilde{f}$ to the values of $\tilde{a}$. All the other bound names and formal parameters in whose scope this call resides become unmapped. Finally, the control is transferred to the entry place of the translation of $S$.

   To make this idea precise, let $B$ denote the set of bound names and formal parameters that are forgotten in the call. Let $A$ be the set of names occurring in $\tilde{a}$ (perhaps multiple times). The required change in the substitution can be modelled by the assignments

$$X_i \leftarrow a \text{ for each } a \in A \qquad \text{and} \qquad \emptyset \leftarrow a \text{ for each } a \in B \setminus A.$$

Here, $X_i$ is the set of formal parameters to which the value of the factual parameter $a$ is assigned. So the $X_i$ are disjoint non-empty sets whose union is $\tilde{f}$. An assignment $X \leftarrow a$ (where $X$ can be empty) simultaneously maps all the variables in $X$ to $\sigma(a)$ and, if $a \in B$, unmaps $a$. Since no two assignments reference the same name, they cannot interfere and thus can be executed in any order or concurrently.

   The subnet implementing an assignment $X \leftarrow a$ has one entry and one exit place and is constructed as follows. For each $val \in dom(a)$ we create a transition $t_{val}$ which:
- consumes a token from the entry place and produces a token on the exit place;
- for each $f \in X$, consumes a token from $[f \neq val]$ (provided this place exists, i.e. $val \in \mathcal{N}$) and produces a token on $[f = val]$;
- if $a \in B$, consumes a token from $[a = val]$, and, in case $val \in \mathcal{N}$, produces a token on $[a \neq val]$ (or on $[r_* \neq val]$ if $a$ is a restricted name).

Such subnets can be combined in either sequential or parallel manner (in the latter case additional fork and join transitions are needed).

**Handling sums**   We assume that sums are *guarded,* i.e. have the form $\sum_{i \in I} \pi_i.S_i$. The entry place of the subnet is connected to the entry places of the translations of each $S_i$ by transitions. In case of communication actions $\pi_i \neq \tau$, these transitions are stubs that carry appropriate synchronisation actions and name binding commands.

**Parallel composition and hiding**   Given two stub transitions $t'$ and $t''$ in different threads representing prefixes $\overline{a}\langle b\rangle$ and $x(y)$, parallel composition $\parallel$ adds a set of transitions implementing the communication, cf. Fig. 6. If static analysis (see Sect. 7) shows that the prefixes are potentially synchronisable, we create for each $i \in dom(a) \cap dom(x)$ and $j \in dom(b) \cap dom(y)$ a transition $t_{ij}$ which:
- consumes tokens from the input places of the stubs $t'$ and $t''$ and produces tokens on their output places;

---

[1]E.g. due to its negative impact on some model checking techniques.  Note however that symmetry reduction mitigates this negative effect, as all the states that are reached by the arbitration are equivalent.

- checks by read arcs that $[a = i]$ and $[x = i]$ are marked, i.e. the substitution maps $a$ and $x$ to the same value $i$ and thus the synchronisation is possible (if $a$ and/or $x$ are in $\mathcal{P}$ then the corresponding arc is not needed);
- checks by a read arc that $[b = j]$ is marked, consumes a token from $[y \neq j]$ (if this place exists) and produces a token on $[y = j]$ (mapping thus $y$ to the value of $b$).

If the synchronisation is possible, exactly one of these transitions is enabled (depending on the values of $a$, $b$, and $x$); else none of these transitions is enabled. After all such synchronisations are performed, the stub transitions are removed from the net by hiding $H$.

3.3. **Example.** Fig. 4 shows the complete translation of the FCP $\nu r.\overline{p}\langle r\rangle.\mathbf{0} \,|\, p(x).\mathbf{0}$. The meanings of the places in $N_{Subst}$ are as in Fig. 3, and the places in the control flow are labelled by the corresponding subterms.

As the FCP has two bound names, $r$ and $x$, we take $\mathcal{N} = \{n_1, n_2\}$. The initialisation of $r$ is represented by two transitions, corresponding to the values $n_1$ and $n_2$. The only possible communication in this example is between the prefixes $\overline{p}\langle r\rangle$ and $p(x)$. Note that the communication is over the public channel $p$, and the communicated values are from $dom(r) \cap dom(x) = \{n_1, n_2\} \cap \{p, n_1, n_2\} = \{n_1, n_2\}$. Hence there are two transitions modelling this communication.



Figure 4: Translation of $\nu r.\overline{p}\langle r\rangle.\mathbf{0} \,|\, p(x).\mathbf{0}$.

## 4. Size of the resulting FCP

We now evaluate the contributions of various parts of the translation to the size of the final safe PN. (Note that the asymptotic size of a safe PN is fully determined by its total number of places, transitions and arcs, as the size of the initial marking is bounded by the number of places.) Recall that we use $\mathcal{N}_{\mathcal{RIF}}$ as the set of names from which the values for restricted names are picked.

**The substitution** $N_{Subst}$    This net consists of

$$(|\mathcal{I}| + |\mathcal{F}|)\,|\mathcal{P}| + (2|\mathcal{I}| + 2|\mathcal{F}| + |\mathcal{R}| + 1)\,|\mathcal{N}_{\mathcal{RIF}}|$$

places, with no transitions or arcs, see Fig. 3, which is $O(\|F\|^2)$ in the worst case.

**Mapping a name in** $\mathcal{I} \cup \mathcal{F}$    A separate transition with $O(1)$ incident arcs is created for each value in $\mathcal{P} \cup \mathcal{N}_{\mathcal{RIF}}$. Hence, the cost of mapping a single input name or formal parameter is $O(\|F\|)$ in the worst case. (The cost of initialising a restricted name is discussed later.)

**Unmapping a name in** $\mathcal{I} \cup \mathcal{F} \cup \mathcal{R}$    A separate transition with $O(1)$ incident arcs is created for each name in $\mathcal{P} \cup \mathcal{N}_{\mathcal{RIF}}$ (in case of an input name or formal parameter) or $\mathcal{N}_{\mathcal{RIF}}$ (in case of a restricted name). Hence the cost of unmapping a single name is $O(|\mathcal{P}| + |\mathcal{N}_{\mathcal{RIF}}|) = O(\|F\|)$ in the worst case.

**Stop processes** A single place is created for each occurrence of **0**, i.e. the total contribution is $O(\|F\|)$ in the worst case.

**Calls** In the worst case all the calls have $O(\|F\|)$ parameters in total, which have to be mapped and unmapped. Hence the contribution is $O(\|F\|^2)$.

**Restrictions** In the worst case the number of restrictions is $O(\|F\|)$, and the corresponding names have to be initialised and then unmapped. For initialisation of a restricted name, a separate transition is created for each value in $\mathcal{N}_{\mathcal{RIF}}$, and this transition has $O(|\mathcal{I}| + |\mathcal{F}|)$ incident arcs, see Fig. 5. Hence the contribution is $O(\|F\| \, |\mathcal{N}_{\mathcal{RIF}}| \, (|\mathcal{I}| + |\mathcal{F}|)) = O(\|F\|^3)$.

**Choices and communication** Implementing the branching resulting from sums contributes $O(\|F\|)$ to the final total. Furthermore, $\tau$-prefixes also contribute at most $O(\|F\|)$. In the worst case the numbers of sending and receiving prefixes are $O(\|F\|)$, and almost all pairs of send/receive actions can synchronise; thus the total number of such synchronisations is $O(\|F\|^2)$. Recall that for a pair of actions $\overline{x_1}\langle y_1 \rangle$ and $x_2(y_2)$, a separate transition with $O(1)$ incident arcs is generated for each pair of values in $\mathcal{P} \cup \mathcal{N}_{\mathcal{RIF}}$, see Fig. 6. Hence, the contribution is $O(\|F\|^2 \, |\mathcal{P} \cup \mathcal{N}_{\mathcal{RIF}}|^2) = O(\|F\|^4)$, dominating thus the other parts of the translation. However, the communication splitting optimisation described in Sect. 7 reduces this contribution down to $O(\|F\|^3)$.

Totaling the above contributions shows that the size of the resulting PN is $O(\|F\|^3)$. Furthermore, converting a general FCP into a safe one can increase the size by a factor bounded by the number of threads, i.e. quadratically in the worst case, see Sect. 2. Therefore, the translation is polynomial for general FCPs too.

It should also be noted that the worst case size computed above is rather pessimistic: the translation admits several practical optimisations, see Sect. 7. The experimental results in Sect. 9 demonstrate that for realistic FCPs the sizes of the resulting PNs are moderate.

## 5. Definition of the translation

We now formalise the proposed translation. To do that, we add further assumptions on the form of the FCP. Again, these assumptions are not restrictive: any FCP can be transformed into the required form. However, the assumptions significantly simplify the correctness proofs by reducing the number of cases that have to be considered.

5.1. **Additional normal form assumptions.** We augment the **(NC)** assumptions as follows: in a defining equation $K(\tilde{f}) := S$, $fn\,(S) = \tilde{f}$, i.e. public names are not allowed in $S$. This assumption can be enforced by passing the required public names as parameters.

To avoid case distinctions for the initial process, we assume there are artificial defining equations $K_{Init,i}(\tilde{f}_{Init,i}) := S_{Init,i}$ with $fn\,(S_{Init,i}) = \tilde{f}_{Init,i} \subseteq \mathcal{F}$, that are called by a virtual initialisation step. Their purpose is to guarantee that the $S_{Init,i}$ have the free names $\tilde{f}_{Init,i}$. We then apply substitutions to assign the expected values to these parameters. This means we can write the given FCP as

$$F = \nu \tilde{r}.(S_{Init,1}\sigma_1 \mid \ldots \mid S_{Init,n}\sigma_n),$$

where $\sigma_i : \tilde{f}_{Init,i} \to \tilde{r} \cup \mathcal{P}$. We additionally assume that the $S_{Init,i}$ are choices or calls and that the FCP does not contain the **0** process.

Moreover, if we have an input $x(y).S$ then we assume $y \in fn\,(S)$, which can be achieved by adding an artificial parameter to the call at the end of the process. Similarly, for a restriction $\nu r.S$ we assume $r \in fn\,(S)$. Restrictions not satisfying this requirement can be dropped due to structural congruence.

5.2. **Construction of $N_{Subst}$.** To represent a substitution like $\{a, b/x, y\}$, we decompose it into elementary substitutions $\{a/x\} \cup \{b/y\}$ of single names. The substitution net has corresponding places $[x=a]$ and $[y=b]$ for each elementary substitution that may occur in such a decomposition. Moreover, there is a second set of places, $[x{\neq}n]$ and $[r_*{\neq}n]$, keeping track of whether an input, a formal parameter, or a restriction is bound to $n \in \mathcal{N}$. These places complement the corresponding substitution places, in particular $[r_*{\neq}n]$ indicates that no restricted name is bound to $n$. (Since at most one restriction can be bound to $n$, this one complement place is sufficient.) $N_{Subst}$ has no transitions. We defer the explanation of its initial marking for the moment. Formally, $N_{Subst} \overset{\mathrm{df}}{=} (P_{Subst} \cup P_{Ref}, \emptyset, \emptyset, M_0)$ with

$$P_{Subst} \overset{\mathrm{df}}{=} ((\mathcal{I} \cup \mathcal{F}) \times \{=\} \times \mathcal{P}) \cup ((\mathcal{I} \cup \mathcal{F} \cup \mathcal{R}) \times \{=\} \times \mathcal{N}) \quad P_{Ref} \overset{\mathrm{df}}{=} (\mathcal{I} \cup \mathcal{F} \cup \{r_*\}) \times \{\neq\} \times \mathcal{N}.$$

**Substitution markings and correspondence**    A marking $M$ of $N_{Subst}$ is called a *substitution marking* if it satisfies the following constraints:

$$\textbf{(SM1)} \quad M([r_*{\neq}n]) + \sum_{r \in \mathcal{R}} M([r{=}n]) = 1 \qquad \sum_{a \in \mathcal{P} \cup \mathcal{N}} M([x{=}a]) \leq 1 \quad \textbf{(SM2)}$$

$$M([x{=}n]) + M([x{\neq}n]) = 1. \quad \textbf{(SM3)}$$

**(SM1)** holds for every $n \in \mathcal{N}$ and states that at most one restricted name is bound to $n$. Moreover, there is a token on $[r_*{\neq}n]$ iff there is no such binding. **(SM2)** states that every name $x \in \mathcal{I} \cup \mathcal{F} \cup \mathcal{R}$ is bound to at most one $a \in \mathcal{P} \cup \mathcal{N}$. The reference counter has to keep track of whether a name $x \in \mathcal{I} \cup \mathcal{F}$ maps to a fresh name $n \in \mathcal{N}$, which motivates **(SM3)**.

Consider now a substitution $\sigma : (\mathcal{I}' \cup \mathcal{F}' \to \mathcal{P} \cup \tilde{r}) \cup (\mathcal{R}' \to \tilde{r})$ where $\mathcal{I}' \subseteq \mathcal{I}$, $\mathcal{F}' \subseteq \mathcal{F}$, $\mathcal{R}' \subseteq \mathcal{R}$, and the second component $\mathcal{R}' \to \tilde{r}$ is injective. A substitution marking $M$ of $N_{Subst}$ is said to *correspond to* $\sigma$ if the following hold:

**(COR1)**: For all $x \in (\mathcal{I} \cup \mathcal{F} \cup \mathcal{R}) \setminus dom(\sigma)$ and $a \in \mathcal{N} \cup \mathcal{P}$, $M([x{=}a]) = 0$.

**(COR2)**: For all $x \in dom(\sigma)$ with $\sigma(x) \in \mathcal{P}$, $M([x{=}\sigma(x)]) = 1$.

**(COR3)**: For all $x \in dom(\sigma)$ with $\sigma(x) \in \tilde{r}$, there is $n \in \mathcal{N}$ s.t. $M([x{=}n]) = 1$.

**(COR4)**: The choice of $n$ preserves the equality of names as required by $\sigma$: for all $x, y \in dom(\sigma)$ with $\sigma(x), \sigma(y) \in \tilde{r}$, we have

$$\sigma(x) = \sigma(y) \quad \text{iff} \quad \text{for all } n \in \mathcal{N},\ M([x{=}n]) = M([y{=}n]).$$

Recall that we translate the specification $F = \nu\tilde{r}.(S_{Init,1}\sigma_1 \mid \ldots \mid S_{Init,n}\sigma_n)$. As the *initial marking* of $N_{Subst}$, we fix some substitution marking that corresponds to $\sigma_1 \cup \ldots \cup \sigma_n$. As we shall see, every choice of fresh names $\tilde{n}$ for $\tilde{r}$ indeed yields bisimilar behaviour. Note that **(NC)** ensures that the union of substitutions is again a function. Fig. 3 illustrates $N_{Subst}$ and the concepts of substitution markings and correspondence.

5.3. **Construction of** $N(S_{Init})$. Petri net $N(S_{Init})$ reflects the control flow of thread $S_{Init}$. To synchronise send and receive prefixes in different threads, we annotate its transitions with labels from

$$\mathcal{L} \stackrel{\text{df}}{=} \{\tau, send(a,b), rec(a,b) \mid a, b \in \mathcal{P} \cup \mathcal{N}\}.$$

To capture the effect that reactions have on substitutions, transitions also carry a set of commands from

$$\mathcal{C} \stackrel{\text{df}}{=} \{map(x,b), unmap(x,b), test([x=b]) \mid x \in \mathcal{I} \cup \mathcal{F} \cup \mathcal{R} \text{ and } b \in \mathcal{P} \cup \mathcal{N}\}.$$

Using these sets, a *control flow net* is defined to be a tuple $(P, T, F, M_0, l, c)$, where $(P, T, F, M_0)$ is a PN and $l : T \to \mathcal{L}$ and $c : T \to \mathbb{P}(\mathcal{C})$ are the transition labelling functions.

    As $S_{Init}$ is a sequential process, transitions in $N(S_{Init})$ will always have a single input and a single output place. This allows us to understand $N(S_{Init})$ as a finite automaton, and hence define it implicitly via a new labelled transition system for $S_{Init}$. Recall that $\mathcal{S}$ is the set of sequential processes. We augment them by lists of names, $\mathcal{S} \times (\mathcal{I} \cup \mathcal{F} \cup \mathcal{R})^*$, carrying the names that have been forgotten and should be eventually unmapped in $N_{Subst}$. Among such augmented processes, we define the labelled transition relation



Figure 5: Translation of a restriction with $map(r,n)$ implemented.

$$\twoheadrightarrow \subseteq (\mathcal{S} \times (\mathcal{I} \cup \mathcal{F} \cup \mathcal{R})^*) \times \mathcal{L} \times \mathbb{P}(\mathcal{C}) \times (\mathcal{S} \times (\mathcal{I} \cup \mathcal{F} \cup \mathcal{R})^*).$$

Each transition carries a label and a set of commands, and will yield a PN transition.

    For restrictions $\nu r.S$, we allocate a fresh name. Since we can select any name that is not in use, such a transition exists for every $n \in \mathcal{N}$:

$$(\nu r.S, \lambda) \xrightarrow[\{map(r,n)\}]{\tau} (S, \lambda). \qquad\qquad (TRANS_\nu)$$

Fig. 5 depicts the transition, together with the implementation of mapping defined below.

    Silent actions yield $\tau$-labelled transitions with empty sets of commands as expected:

$$(\tau.S + \dots, \lambda) \xrightarrow[\emptyset]{\tau} (S, \lambda \cdot \lambda'), \qquad\qquad (TRANS_\tau)$$

where $\lambda' = fn\,(\tau.S + \dots) \setminus fn\,(S)$ contains the names that were free in the choice process but have been forgotten in $S$. With an ordering on $\mathcal{P} \cup \mathcal{N}$, we can understand this set as a sequence.

    Communications are more subtle. Consider $\overline{x}\langle y\rangle.S + \dots$ that sends $y$ on channel $x$. With appropriate tests, we find the names $a$ and $b$ to which $x$ and $y$ are mapped. These names then determine the transition label. So for all $a, b \in \mathcal{P} \cup \mathcal{N}$, we have

$$(\overline{x}\langle y\rangle.S + \dots, \lambda) \xrightarrow[\{test([x=a]), test([y=b])\}]{send(a,b)} (S, \lambda \cdot \lambda'). \qquad\qquad (TRANS_{snd})$$

$$H(N_1 \| N_2) \qquad\qquad N_{Subst} \vartriangleleft H(N_1 \| N_2)$$



Figure 6: Translation of communication (left), parallel composition and hiding (center), and implementation of commands (right).

Sequence $\lambda'$ again contains the names that have been forgotten during this step. A receive action in $x(y).S + \dots$ is handled like a send, but introduces a new binding. For all $a, b \in \mathcal{P} \cup \mathcal{N}$, we have

$$(x(y).S + \dots, \lambda) \xrightarrow[\{test([x=a]),map(y,b)\}]{rec(a,b)} (S, \lambda \cdot \lambda'). \qquad (TRANS_{rec})$$

There are similar transitions for the remaining prefixes $\pi_i$ with $i \in I$. Fig. 6(left) illustrates the transitions for send and receive actions.

For a call $K \lfloor x_1, \dots, x_n \rfloor$ with $K(f_1, \dots, f_n) := S$, the idea is to iteratively update the substitution by binding the formal parameters to the factual ones. (Note that we assumed $fn(S) = \{f_1, \dots, f_n\}$.) Afterwards, we unmap the names in $\lambda$, which will then include the factual parameters. Since no equation calls itself, we do not accidentally unmap the just mapped formal parameters. The following transitions are created for each $a \in \mathcal{P} \cup \mathcal{N}$:

$$(K \lfloor x_1, \dots, x_m \rfloor, \lambda) \xrightarrow[\{test([x_m=a]),map(f_m,a)\}]{\tau} (K \lfloor x_1, \dots, x_{m-1} \rfloor, \lambda'), \qquad (TRANS_{call_1})$$

where $\lambda' \stackrel{\mathrm{df}}{=} \lambda$ if $x_m \in \lambda$ and $\lambda' \stackrel{\mathrm{df}}{=} \lambda \cdot x_m$ otherwise. (This case distinction ensures that we will unmap a factual parameter precisely once, even if it occurs multiple times in the list of factual parameters.) When all parameters have been passed, we unmap the names in $\lambda \neq \varepsilon$, by creating the following transitions for each $a \in \mathcal{P} \cup \mathcal{N}$:

$$(K \lfloor - \rfloor, x \cdot \lambda) \xrightarrow[\{unmap(x,a)\}]{\tau} (K \lfloor - \rfloor, \lambda). \qquad (TRANS_{call_2})$$

When $\lambda = \varepsilon$ has been reached, we transfer the control to the body $S$ of the defining equation:

$$(K \lfloor - \rfloor, \varepsilon) \xrightarrow[\emptyset]{\tau} (S, \varepsilon). \qquad (TRANS_{call_3})$$

Petri net $N(S_{Init})$ is the restriction of $(\mathcal{S} \times (\mathcal{I} \cup \mathcal{F} \cup \mathcal{R})^*, \twoheadrightarrow)$ to the augmented processes that are reachable from $(S_{Init}, \varepsilon)$ via $\twoheadrightarrow$. The initial marking puts one token on place $(S_{Init}, \varepsilon)$ and leaves the remaining places unmarked.

5.4. **Operations on nets.** We now describe the operations composing the nets.

**Parallel composition** $\|$   Parallel composition of labelled nets is classical in Petri net theory. The variant we use is inspired by [2]: $N_1 \parallel N_2$ forms the disjoint union of $N_1$ and $N_2$, and then synchronises the transitions $t_1$ in $N_1$ that are labelled by $l_1(t_1) = send(a, b)$ (resp. $rec(a, b)$) with the transitions $t_2$ in $N_2$ that are labelled by $l_2(t_2) = rec(a, b)$ (resp.

$send(a,b)$). The result is a new transition $(t_1, t_2)$ labelled by $\tau$, which carries the (disjoint) union of the commands for $t_1$ and $t_2$. Note that the transitions of $N_1$ and $N_2$ are still available for further synchronisations with some $N_3$. This in particular implies that $\parallel$ is associative and commutative.

**Hiding $H$** The *hiding operator* removes from a PN $N$ all transitions $t$ with $l(t) \neq \tau$. Since $H(N)$ contains only $\tau$-labelled transitions, we can omit the labelling function from the result. The combination of parallel composition and hiding is illustrated in Fig. 6(center).

**Implementation operation $\lhd$** Consider the two Petri nets $N_1 = N_{Subst} = (P_1, \emptyset, \emptyset, M_{0,1})$ and $N_2 = H(N(S_{Init,1}) \parallel \ldots \parallel N(S_{Init,n})) = (P_2, T, F_2, M_{0,2}, c)$ defined so far. The implementation operation

$$N_1 \lhd N_2 \stackrel{\mathrm{df}}{=} (P_1 \cup P_2, T, F_2 \cup F, M_{0,1} \cup M_{0,2})$$

yields a standard Petri net without labelling. Its purpose is to implement the commands labelling the transitions of $N_2$ by adding arcs between the two nets. We fix a transition $t \in T$ and a command $c \in c(t)$, and define the arcs that have to be added between $t$ and some places of $N_1$ to implement $c$. We do the case analysis for the possible types of $c$:

$test([x{=}b])$ We add a loop to place $[x{=}b]$: $([x{=}b], t), (t, [x{=}b]) \in F$.

$map(x, p), map(x, n), map(r, n)$ A map command differentiates according to whether the first component is an input name or a formal parameter $x \in \mathcal{I} \cup \mathcal{F}$, or whether it is a restricted name $r \in \mathcal{R}$. If $x$ is assigned a public name, $map(x, p) \in c(t)$ with $p \in \mathcal{P}$, we add an arc producing a token in the corresponding place of the substitution net: $(t, [x{=}p]) \in F$. If $x$ is assigned some $n \in \mathcal{N}$, $map(x, n) \in c(t)$, we additionally remove the token from the reference counter: $(t, [x{=}n]), ([x{\neq}n], t) \in F$. To represent the restricted name $r \in \mathcal{R}$ by a name $n \in \mathcal{N}$, we first check that no other name is currently mapped to $n$ using the reference counter for $n$. In case $n$ is currently not in use, we introduce the binding $[r{=}n]$ to the substitution net: $([r_*{\neq}n], t), (t, [r{=}n]) \in F$ and $\{([x{\neq}n], t), (t, [x{\neq}n]) \mid x \in \mathcal{I} \cup \mathcal{F}\} \subseteq F$.

$unmap(x, p), unmap(x, n), unmap(r, n)$ An unmap removes the binding of $x \in \mathcal{I} \cup \mathcal{F}$: $([x{=}p/n], t) \in F$. Moreover, if $n \in \mathcal{N}$ it updates the reference counter: $(t, [x{\neq}n]) \in F$. When we remove the binding of $r \in \mathcal{R}$ to $n \in \mathcal{N}$, we update $[r_*{\neq}n]$ in the reference counter: $([r{=}n], t), (t, [r_*{\neq}n]) \in F$.

Fig. 5 illustrates the implementation of mapping for a restriction, $map(r, n)$. Tests and mapping of an input name are shown in Fig. 6(right).

## 6. Correctness of the translation

To prove the translation correct, we relate $F$ and $N(F)$ by a suitable form of bisimulation. The problem is that $N(F)$ may perform several steps to mimic one transition of $F$. The reason is that changes to substitutions (as induced e.g. by $\nu r.S$) are handled by transitions in $N(F)$ whereas $F$ uses structural congruence, i.e. a substitution change does not necessarily lead to a step in the reaction relation of $F$. To obtain a clean relationship between the models, we restrict the transition system of $N(F)$ to so-called *stable markings* and *race free transition sequences* between them. Intuitively, stable markings correspond to the choices and process calls in $F$, and race free transition sequences mimic the reaction steps between them. We show below that this restriction is insignificant, as any transition sequence is equivalent to some race free one.

Place $(S, \lambda)$ of $N(F) = N_{Subst} \lhd H(N(S_{Init,1}) \parallel \ldots \parallel N(S_{Init,n}))$ is called *stable* if $S$ is a choice or a call to a process identifier with full parameter list. Marking $M$ of $N(F)$ is called *stable* if, in every control flow net $N(S_{Init,i})$, it marks a stable place. We denote by $Reach_{Stbl}(N(F))$ the set of stable markings that are reachable in $N(F)$.

A transition sequence $t_1, \ldots, t_n$ between stable markings $M, M' \in Reach_{Stbl}(N(F))$ is *race free* if exactly one $t_i$ is either of the form $(TRANS_\tau)$ for a silent action, of the form $(t, t')$ for communication actions $(TRANS_{snd})$, $(TRANS_{rec})$, or of the form $(TRANS_{call_3})$ for an identifier call, cf. Sect. 5.3. Thus, a race free transition sequence corresponds to precisely one step in the reaction relation of $F$, characterised by $t_i$, while the other transitions implement the substitution changes between $M$ and $M'$. In particular, no intermediary marking is stable. We denote the fact that there is such a race free transition sequence by $M \Rightarrow M'$.

We now show that every transition sequence reaching a stable marking $M$ can be replaced by a series of race free transition sequences. This means the restriction to race free sequences is inconsequential.

**Lemma 6.1.** *For every transition sequence $M_1 \to^+ M_2$ between $M_1, M_2 \in Reach_{Stbl}(N(F))$, there is a sequence $M_1 \Rightarrow^+ M_2'$ with $M_2' \in Reach_{Stbl}(N(F))$ and*
- *the control flow parts of $M_2$ and $M_2'$ coincide, and*
- *the substitution parts of $M_2$ and $M_2'$ correspond to a same substitution.*

*Moreover, the latter sequence is a rearrangement of the former one, modulo transitions implementing map and unmap operations for restricted names using different values.*

*Proof.* To obtain $M_1 \Rightarrow^+ M_2'$, we proceed as follows. We project the sequence $M_1 \to^+ M_2$ to the transitions that reflect $\pi$-calculus reactions. After each such transition, we insert the required initialisations of restricted names by unused values (not necessarily the same as in the original sequence). Before each call to a process identifier, we insert the necessary unmapping operations (the ones for restricted names are amended to use the values given during the corresponding initialisations). The result is race free. To see that the sequence is enabled, note that initialisations of restricted names cannot be blocked because the pool of fresh names is large enough. Moreover, unmap operations can never be blocked. $\square$

6.1. **Bisimulation.** Since the initial marking $M_0$ of $N(F)$ is stable by the assumption on $S_{Init,i}$ from Sect. 5.1, we can define the *stable transition system of $N(F)$* as

$$\mathcal{T}_{Stbl}(N(F)) \stackrel{\text{df}}{=} (Reach_{Stbl}(N(F)), \Rightarrow, M_0).$$

**Theorem 6.2.** *The transition system of $F$ and the stable transition system of $N(F)$ are bisimilar, $\mathcal{T}(F) \sim \mathcal{T}_{Stbl}(N(F))$, via the bisimulation $\mathcal{B}$ defined below.*

We defer the proof for the moment. To define the bisimulation relation, we use the fact that every process reachable from $F$ is structurally congruent to some $\nu\tilde{r}.(S_1\sigma_1 \mid \ldots \mid S_n\sigma_n)$. Here, $S_i$ is a choice or an identifier call that has been derived from some process $S$ with $K(\tilde{f}) := S$. Derived means $(S, \varepsilon) \twoheadrightarrow^+ (S_i, \lambda_i)$ so that no intermediary process is a call to a process identifier. As second requirement, we have

$$\sigma_i : fn(S_i) \cup \lambda_i \to \tilde{r} \cup \mathcal{P}. \quad \textbf{(DOM)}$$

This means the domain of $\sigma_i$ are the free names in $S_i$ together with the names $\lambda_i$ that have already been forgotten. The two sets are disjoint, $fn(S_i) \cap \lambda_i = \emptyset$. The above process

actually is in Milner's standard form [19], but makes additional assumptions about the shape of threads and the domain of substitutions.

We define $\mathcal{B} \subseteq Reach(F)/_{\equiv} \times Reach_{Stbl}(N(F))$ to contain $(\underline{G}, M_1 \cup M_2) \in \mathcal{B}$ if there is a process $\nu\tilde{r}.(S_1\sigma_1 \mid \ldots \mid S_n\sigma_n) \equiv G$ as above so that the following hold:

- marking $M_1$ of $N_{Subst}$ corresponds to $\sigma_1 \cup \ldots \cup \sigma_n$ and
- for the control flow marking, we have $M_2(S_i, \lambda_i) = 1$ for all $i \in \{1, \ldots, n\}$.

To relate $\mathcal{T}(F)$ and the full transition system $\mathcal{T}(N(F))$, consider a transition sequence $M_1 \to^+ M_2$ between stable markings $M_1, M_2 \in Reach_{Stbl}(N(F))$ that need not be race free. Due to Lemma 6.1 it can be rearranged to a race free sequence $M_1 \Rightarrow^+ M_2'$. With the above bisimilarity, this race free transition sequence is mimicked by a sequence of $\pi$-calculus transitions $\underline{F} \to^+ \underline{G}$ with $(\underline{G}, M_2') \in \mathcal{B}$. With Lemma 6.1 and the definition of $\mathcal{B}$, we also have $(\underline{G}, M_2) \in \mathcal{B}$. In the reverse direction, a single process transition is still mimicked by a sequence of PN transitions (that happens to be race free). Hence, the following holds:

**Theorem 6.3.** *The transition systems of $F$ and $N(F)$ are weakly bisimilar, $\mathcal{T}(F) \approx \mathcal{T}(N(F))$, taking $\mathcal{B}$ defined above as a weak bisimulation.*

This result allows one to check temporal properties of FCPs using their PN representations.

6.2. **Bisimulation Proof.** We now turn to the proof of Theorem 6.2. We have to show that for each pair $(\underline{G}, M) \in \mathcal{B}$, every transition $\underline{G} \hookrightarrow \underline{G'}$ can be mimicked by a race free transition sequence in $N(F)$, i.e. there is a stable marking $M'$ with $M \Rightarrow M'$ such that $(\underline{G'}, M') \in \mathcal{B}$. Moreover and in turn, the race free transition sequences in $N(F)$ should be imitated in process $F$. The proof is split into two parts, formulated as Lemmas 6.4 and 6.5, for both directions respectively.

**Lemma 6.4.** *Consider $(\underline{G}, M) \in \mathcal{B}$. For all $\underline{G'}$ with $\underline{G} \hookrightarrow \underline{G'}$ there is a stable marking $M' \in Reach_{Stbl}(N(F))$ such that $M \Rightarrow M'$ and $(\underline{G'}, M') \in \mathcal{B}$.*

*Proof.* Process $G$ is structurally congruent to $\nu\tilde{r}.(S_1\sigma_1 \mid \ldots \mid S_n\sigma_n)$. By the base cases of the reaction rules, transition $\underline{G} \hookrightarrow \underline{G'}$ exists iff (1) either two processes $S_i\sigma_i$ and $S_j\sigma_j$ with $i \neq j \in \{1, \ldots, n\}$ communicate, (2) we resolve a call to a process identifier in some $S_i\sigma_i$, $i \in \{1, \ldots, n\}$, or (3) we have a $\tau$ action. Silent steps are easier than the former two and hence omitted in the proof.

**Case 1: Communication** For simplicity, we assume that: the first two threads communicate using the first prefixes; after the communication, the first thread yields choice or call $S_1'$; the second process creates precisely one restricted name before becoming a choice or a call $S_2'$; the communication is over restricted names and a restricted name is sent. The remaining cases are along similar lines. We thus have $G \equiv \nu\tilde{r}.(S_1\sigma_1 \mid \ldots \mid S_n\sigma_n)$ with

$$S_1 = \overline{x_1}\langle y_1 \rangle.S_1' + \ldots \quad S_2 = x_2(y_2).\nu r.S_2' + \ldots \quad \sigma_1(x_1) = \sigma_2(x_2) \in \tilde{r} \quad \sigma_1(y_1) \in \tilde{r}.$$

The process resulting from the communication is

$$G' \stackrel{\mathrm{df}}{=} \nu\tilde{r}.a_r.(S_1'\sigma_1 \mid S_2'\sigma_2' \mid S_3\sigma_3 \mid \ldots \mid S_n\sigma_n) \quad \text{with} \quad \sigma_2' \stackrel{\mathrm{df}}{=} \sigma_2\{\sigma_1(y_1)/y_2\}\{a_r/r\}.$$

We argue that $G'$ has the desired normal form. The processes $S_1'$ and $S_2'$ are choices or calls. Moreover, $(S, \varepsilon) \twoheadrightarrow^* (S_2, \lambda_2)$ implies $(S, \varepsilon) \twoheadrightarrow^* (S_2', \lambda_2 \cdot \lambda_2')$. This means $S_1'$ and $S_2'$

have been derived, as required. It remains to show **(DOM)**. We do the proof for $\sigma'_2$, the reasoning for $\sigma_1$ is simpler:

$$dom(\sigma'_2) \tag{6.1}$$

$$= dom(\sigma_2) \cup \{y_2, r\} \tag{6.2}$$

$$= \lambda_2 \cup fn\,(S_2) \cup \{y_2, r\} \tag{6.3}$$

$$= \lambda_2 \cup (fn\,(S_2) \setminus fn\,(\nu r.S'_2)) \cup (fn\,(\nu r.S'_2) \setminus \{y_2\}) \cup \{y_2, r\} \tag{6.4}$$

$$= \lambda_2 \cup (fn\,(S_2) \setminus fn\,(\nu r.S'_2)) \cup fn\,(S'_2) \cup \{y_2, r\} \tag{6.5}$$

$$= \lambda_2 \cup (fn\,(S_2) \setminus fn\,(\nu r.S'_2)) \cup fn\,(S'_2) \tag{6.6}$$

$$= \lambda_2 \cdot \lambda'_2 \cup fn\,(S'_2)\,. \tag{6.7}$$

Equation (6.3) is **(DOM)** for $\sigma_2$. Equation (6.4) uses the fact that

$$fn\,(S_2) = (fn\,(S_2) \setminus fn\,(\nu r.S'_2)) \cup (fn\,(\nu r.S'_2) \setminus \{y_2\}).$$

This is due to $fn\,(\nu r.S'_2) \setminus \{y_2\} \subseteq fn(S_2)$. Equation (6.5) is due to

$$(fn\,(\nu r.S'_2) \setminus \{y_2\}) \cup \{y_2, r\} = fn\,(S'_2) \cup \{y_2, r\}.$$

Equation (6.6) holds by $\{y_2, r\} \subseteq fn(S'_2)$. Finally, Equation (6.7) holds by definition of the augmented transition relation $\twoheadrightarrow$.

We now argue that (1.a) there is $M' \in Reach_{Stbl}(N(F))$ so that $M \Rightarrow M'$ and (1.b) $(\underline{G'}, M') \in \mathcal{B}$.

**Claim 1.a: There is** $M' \in Reach_{Stbl}(N(F))$ **with** $M \Rightarrow M'$   Let $M = M_1 \cup M_2$ so that $M_1$ is the substitution marking and $M_2$ is the control flow marking. Since $(\underline{G}, M_1 \cup M_2) \in \mathcal{B}$, we have $M_2((S_1, \lambda_1)) = 1 = M_2((S_2, \lambda_2))$. Moreover, $M_1$ corresponds to $\sigma_1 \cup \ldots \cup \sigma_n$. In the following, we also use $\sigma$ to refer to this union. Since $\sigma_1(x_1), \sigma_2(x_2), \sigma_1(y_1) \in \tilde{r}$, by **(COR3)** we have fresh names $n_1, n_2, n_3 \in \mathcal{N}$ with $M_1([x_1{=}n_1]) = 1 = M_1([x_2{=}n_2]) = M_1([y_1{=}n_3])$. Since $\sigma_1(x_1) = \sigma_2(x_2)$, we conclude $n_1 = n_2$ by **(COR4)**.

It remains to show that there is a fresh name available in $\mathcal{N}$ which we can use to represent $r$. As $r \notin dom(\sigma)$, we have

$$|dom(\sigma)| < |\mathcal{I}| + |\mathcal{F}| + |\mathcal{R}| = |\mathcal{N}|.$$

With **(COR1)**, for $x \notin dom(\sigma)$ we have $M_1([x{=}n]) = 0$ for all $n \in \mathcal{N}$. For $x \in dom(\sigma)$, we have at most one place $[x{=}a]$ marked by **(SM2)**. Together, these mean there is a name $n \in \mathcal{N}$ with $M_1([x{=}n]) = 0$ for all $x \in \mathcal{I} \cup \mathcal{F} \cup \mathcal{R}$. Let this name be $n_r$. As $M_1([x{=}n_r]) = 0$ for $x \in \mathcal{I} \cup \mathcal{F} \cup \mathcal{R}$, **(SM1)** and **(SM3)** ensure $M_1([x{\neq}n_r]) = 1$ for $x \in \mathcal{I} \cup \mathcal{F} \cup \{r_*\}$.

Before parallel composition, the original net $N(S_{Init,1})$ had the following transition sequence leaving place $(S_1, \lambda_1)$:

$$(S_1, \lambda_1) \xrightarrow[\{test([x_1=n_1]),\, test([y_1=n_3])\}]{send(n_1,n_3)} (S'_1, \lambda'_1).$$

Similarly, from $(S_2, \lambda_2)$ in $N(S_{Init,2})$ we get

$$(S_2, \lambda_2) \xrightarrow[\{test([x_2=n_1]),\, map([y_2=n_3],\})]{rec(n_1,n_3)} (\nu r.S'_2, \lambda_2 \cdot \lambda'_2) \xrightarrow[\{map(r,n_r)\}]{\tau} (S'_2, \lambda_2 \cdot \lambda'_2).$$

Parallel composition joins the communicating transitions of the two nets, and we denote the result by $(t_1, t_2)$. Then hiding removes the original transitions $t_1$ labelled by $send(n_1, n_3)$

and $t_2$ labelled by $rec(n_1, n_3)$. Then, for $(t_1, t_2)$ and for the transition $t_r$ mapping $r$ to a fresh name, the implementation operation adds arcs to and from $N_{Subst}$.

We now show that the transition sequence $(t_1, t_2)\, t_r$ is enabled. We argued that $(S_i, \lambda_i)$ carries a token. This means the control flow is at the right place. We have $M_1([x_1{=}n_1]) = 1 = M_1([x_2{=}n_1]) = M_1([y_1{=}n_3])$. Hence, the test arcs to the substitution net are enabled. We have $y_2 \in bn(S_2)$. Hence, the name is not in the domain of $\sigma_2$ by **(DOM)** and **(NC)**. With **(COR1)**, $M([y_2{=}a]) = 0$ holds for all names $a \in \mathcal{P} \cup \mathcal{N}$. In particular, $M([y_2{=}n_3]) = 0$. With **(SM3)**, we conclude $M([y_2{\neq}n_3]) = 1$. This ensures $map(y_2, n_3)$ is enabled. For $t_r$, we have $M([x{\neq}n_r]) = 1$ for all $x \in \mathcal{I} \cup \mathcal{F} \cup \{r_*\}$. Hence, the transition is enabled.

The resulting marking $M'$ puts tokens on $(S'_1, \lambda'_1)$ and $(S'_2, \lambda_2 \cdot \lambda'_2)$ which are stable places. This means $M'$ is stable. The marking is reachable as $M$ was reachable. Moreover, transition sequence $(t_1, t_2) \cdot t_r$ above is race free.

**Claim 1.b:** $(\underline{G'}, M') \in \mathcal{B}$   Again $M' = M'_1 \cup M'_2$ where $M'_1$ is the marking of $N_{Subst}$ and $M'_2$ is the control flow. For the control flow, we moved the single token from $(S_1, \lambda_1)$ to $(S'_1, \lambda'_1)$ and from $(S_2, \lambda_2)$ to $(S'_2, \lambda_2 \cdot \lambda'_2)$ as required.

For $N_{Subst}$, we show that we obtain a substitution marking. We already argued that $M_1([y_2{=}a]) = 0$ for all $a \in \mathcal{P} \cup \mathcal{N}$ and hence $M_1([y_2{\neq}n_3]) = 1$. We consume the latter token and move it to $M'_1([y_2{=}n_3]) = 1$. This means we still map $y_2$ to at most one name as required by **(SM2)**. Moreover, the invariant on reference counting **(SM3)** is satisfied.

Name $r$ is not in the domain of $\sigma_2$. Hence, the places $[r{=}a]$ are empty for all $a \in \mathcal{N} \cup \mathcal{P}$. We move the token from $M_1([r_*{\neq}n_r]) = 1$ to $M'_1([r{=}n_r]) = 1$. As a result, the places $[r{=}a]$ for all $a \in \mathcal{N} \cup \mathcal{P}$ together carry at most one token as required by **(SM2)**. Moreover, the places $[r{=}n_r]$ for all $r \in \mathcal{R}$ plus $[r_*{\neq}n_r]$ carry precisely one token. This proves **(SM1)**. We have a substitution marking.

We have to show that $M'_1$ corresponds to $\sigma' \overset{\text{df}}{=} \sigma_1 \cup \sigma'_2 \cup \sigma_3 \cup \ldots \cup \sigma_n$. We only introduce bindings for $y_2$ and $r$. For $y_2$ we have $\sigma'_2(y_2) = \sigma_1(y_1) \in \tilde{r}$. Hence, it is correct that we map $M'_1([y_2{=}n_3]) = 1$ with $n_3 \in \mathcal{N}$. The reasoning is similar for $r$ with $\sigma'_2(r) = a_r$. **(COR3)** holds. Marking $M'_1$ only introduces tokens to the places $[y_2{=}n_3]$ and $[r{=}n_r]$ with $\{y_2, r\} \subseteq dom(\sigma')$. For the remaining names $x \in \mathcal{I} \cup \mathcal{F} \cup \mathcal{R} \setminus \{y_2, r\}$, it coincides with $M_1$. Note that for $x \notin dom(\sigma')$ we have $x \notin dom(\sigma)$. Hence, by **(COR1)** for $M_1$, we get $M'_1([x{=}a]) = M_1([x{=}a]) = 0$ for all $a \in \mathcal{P} \cup \mathcal{N}$. This proves **(COR1)** for $M'_1$.

It remains to show **(COR4)**: the equality required by $\sigma'$ coincides with the choice of fresh names. For $r$ we have $M'_1([r{=}n_r]) = 1$ and $M'_1([x{=}n_r]) = 0$ for all other names $r \neq x \in \mathcal{I} \cup \mathcal{F} \cup \mathcal{R}$. This coincides with the requirement that $\sigma'(r) \neq \sigma'(x)$. For $y_2$, we only consider $x \notin \{y_2, r\}$ and get

$$\sigma'(y_2) = \sigma'(x) \quad \text{iff} \quad \sigma(y_1) = \sigma(x)$$
$$\text{iff } M_1([y_1{=}n]) = M_1([x{=}n]) \text{ for all } n \in \mathcal{N}$$
$$\text{iff } M'_1([y_1{=}n]) = M'_1([x{=}n]) \text{ for all } n \in \mathcal{N}$$
$$\text{iff } M'_1([y_2{=}n]) = M'_1([x{=}n]) \text{ for all } n \in \mathcal{N}.$$

The first equivalence holds by $\sigma'(y_2) = \sigma(y_1)$. The second equivalence is **(COR4)** for $\sigma$, the third is the observation that $M_1$ and $M'_1$ coincide on all names except $y_2$ and $r$. The last equivalence is the fact that the rows for $y_1$ and $y_2$ coincide. This is by **(SM2)**, in combination with $M'_1([y_1{=}n_3]) = 1 = M'_1([y_2{=}n_3])$.

**Case 2: Identifier calls**   We have $G \equiv \nu \tilde{r}.(K\lfloor \tilde{x} \rfloor \sigma_1 \mid \ldots \mid S_n \sigma_n)$ with $K(\tilde{f}) := S$. We assume $S$ already is a choice or a call. The process resulting from the call $K\lfloor \tilde{x} \rfloor \sigma_1$ is

$$G' \stackrel{\mathrm{df}}{=} \nu \tilde{r}.(S \sigma_1' \mid \ldots \mid S_n \sigma_n) \quad \text{with} \quad \sigma_1' \stackrel{\mathrm{df}}{=} \{\sigma_1(\tilde{x})/\tilde{f}\}.$$

We argue that $G'$ has the desired normal form. The process $S$ is a choice or a call. It has been derived trivially as it is the defining process. For **(DOM)**, we have as desired

$$dom(\sigma_1') = \tilde{f} = \mathit{fn}\,(S) = \mathit{fn}\,(S) \cup \emptyset.$$

We now argue that (2.a) there is $M' \in \mathit{Reach}_{Stbl}(N(F))$ so that $M \Rightarrow M'$ and (2.b) $(\underline{G'}, M') \in \mathcal{B}$.

**Claim 2.a: There is $M' \in \mathit{Reach}_{Stbl}(N(F))$ with $M \Rightarrow M'$**   Let $M = M_1 \cup M_2$ so that $M_1$ is the substitution marking and $M_2$ is the control flow marking. Since $(\underline{G}, M_1 \cup M_2) \in \mathcal{B}$, we know that $M_1$ corresponds to $\sigma \stackrel{\mathrm{df}}{=} \sigma_1 \cup \ldots \cup \sigma_n$. For $M_2$, we have $M_2(K\lfloor \tilde{x} \rfloor, \lambda) = 1$. Moreover, by **(DOM)**, we have $\tilde{x} \cup \lambda = dom(\sigma_1)$. Hence, for every name $x_i \in \tilde{x} \cup \lambda$ we have a name $a_i \in \mathcal{P} \cup \mathcal{N}$ so that $M_1([x_i{=}a_i]) = 1$ by **(COR2)** and **(COR3)**. Since an equation does not call itself and since all formal parameters are unique by **(NC)**, we have $\tilde{f} \cap dom(\sigma) = \emptyset$ by **(DOM)**. This means $M_1([f{=}a]) = 0$ for all $f \in \tilde{f}$ and all $a \in \mathcal{N} \cup \mathcal{P}$. With **(SM3)**, we get $M_1([f{\neq}n]) = 1$ for all $f \in \tilde{f}$ and all $n \in \mathcal{N}$.

By definition, Petri net $N(S_{Init,1})$ has the following transition sequence:

$$(K\lfloor \tilde{x} \rfloor, \lambda) \xrightarrow[\{test([x_i{=}a_i]),map(f_i,a_i)\}]{\tau}{}^{+} (K\lfloor - \rfloor, \lambda') \xrightarrow[\{unmap(x_i,a_i)\}]{\tau}{}^{+} (K\lfloor - \rfloor, \varepsilon) \xrightarrow[\emptyset]{\tau} (S, \varepsilon).$$

The first transition sequence introduces the bindings for $\tilde{f}$ and moves the names in $\tilde{x}$ to $\lambda$. The result is $(K\lfloor - \rfloor, \lambda')$ with $\lambda' = \lambda \cdot \tilde{x}'$, where $\tilde{x}'$ is obtained from $\tilde{x}$ by removing the duplicates. The next transition sequence unmaps all names in $\lambda'$. Finally, the token is moved to $(S, \varepsilon)$.

We now show that the composed sequence is enabled. For the first sequence, the tests are enabled with $M_1([x_i{=}a_i]) = 1$. For formal parameters, mapping $map(f, p)$ with $p \in \mathcal{P}$ is always enabled, and $map(f, n)$ with $n \in \mathcal{N}$ requires $M_1([f{\neq}n]) = 1$. This holds by the above argumentation. The second transition sequence removes the tokens from $[x_i{=}a_i]$. Since we do not repeat names in $\tilde{x}'$ and since $\tilde{x} \cap \lambda = \emptyset$, all transitions are enabled. For $a_i = n \in \mathcal{N}$, unmapping introduces a token to $[x_i{\neq}n]$ or to $[r_*{\neq}n]$.

The resulting marking $M'$ puts a token on the stable place $(S, \varepsilon)$, i.e. $M'$ is stable. $M'$ is reachable as $M$ was reachable. Moreover, the transition sequence above is race free.

**Claim 2.b: $(\underline{G'}, M') \in \mathcal{B}$**   Again we have $M' = M_1' \cup M_2'$ where $M_1'$ is the marking of $N_{Subst}$ and $M_2'$ is the control flow marking. For the control flow, we moved the single token from $(K\lfloor \tilde{x} \rfloor, \lambda)$ to $(S, \varepsilon)$ as required.

For $N_{Subst}$, we show that we obtain a substitution marking. We already argued that $M_1([f{=}a]) = 0$ for all $a \in \mathcal{P} \cup \mathcal{N}$ and hence $M_1([f{\neq}n]) = 1$ for all $n \in \mathcal{N}$. We introduce a token $M_1'([f_i{=}a_i]) = 1$, potentially consuming the complement marking if $a_i = n \in \mathcal{N}$. **(SM2)** holds: names are bound at most once. The second transition sequence manipulates the places for $\tilde{x}' \cup \lambda$. These names are disjoint from $\tilde{f}$ due to $\tilde{f} \cap dom(\sigma_1) = \emptyset$ explained above. We remove all tokens $M_1([x_i{=}a_i]) = 1$ with $x_i \in \tilde{x}' \cup \lambda$. The implementation of unmap ensures we reinstall complement markings. More precisely, if $x_i = r \in \mathcal{R}$ and $a_i = n$, we mark $M_1'([r_*{\neq}n]) = 1$. Since by **(SM1)**, name $r$ was the only restriction bound to $n$,

the constraint continues to hold with $[r_* \neq n]$ marked. If $M_1([x_i{=}n]) = 1$ with $x_i \in \mathcal{I} \cup \mathcal{F}$, we get $M_1'([x_i{\neq}n]) = 1$. Hence, **(SM3)** continues to hold. We have a substitution marking.

We have to show that $M_1'$ corresponds to $\sigma' \stackrel{\mathrm{df}}{=} \sigma_1' \cup \sigma_2 \cup \ldots \cup \sigma_n$. We focus on $\sigma_1'$ and assume $\sigma_1'(f_i) \in \tilde{r}$. This means $\sigma_1(x_i) \in \tilde{r}$ for the corresponding name $x_i \in \tilde{x}$. Since $M_1$ corresponds to $\sigma$, by **(COR3)** for $M_1$ we have $M_1([x_i{=}n]) = 1$ for a name $n \in \mathcal{N}$. By **(SM2)**, $x_i$ is bound to only one name. This means $n$ has to be the name $a_i$, $n = a_i$, that we chose for the transition. As a result, we have $M_1'([f{=}n]) = 1$ with $n \in \mathcal{N}$ as required. For $\sigma_1'(f) \in \mathcal{P}$, the reasoning is similar. For the names in $\mathcal{I} \cup \mathcal{F} \cup \mathcal{R} \setminus (dom(\sigma_1) \cup \tilde{f})$, markings $M_1$ and $M_1'$ coincide. Hence, if $x \notin dom(\sigma')$ we either have $x \notin dom(\sigma)$ or we have $x \in dom(\sigma_1)$. In the former case, we get $M_1'([x{=}a]) = M_1([x{=}a]) = 0$ for all $a \in \mathcal{P} \cup \mathcal{N}$ by **(COR1)** for $\sigma$. In the latter case, the name has been explicitly unmapped by the second transition sequence. Hence, **(COR1)** holds for $\sigma'$.

It remains to show **(COR4)**: the equality required by $\sigma'$ coincides with the choice of fresh names. Consider $f_i, f_j \in \tilde{f}$:

$$\sigma'(f_i) = \sigma'(f_j) \quad \text{iff} \quad \sigma(x_i) = \sigma(x_j)$$
$$\text{iff } M_1([x_i{=}n]) = M_1([x_j{=}n]) \text{ for all } n \in \mathcal{N}$$
$$\text{iff } M_1'([f_i{=}n]) = M_1'([f_j{=}n]) \text{ for all } n \in \mathcal{N}.$$

The first equivalence holds by $\sigma_1'(f_i) = \sigma_1(x_i)$ and $\sigma_1'(f_j) = \sigma_1(x_j)$. The second is **(COR4)** for $\sigma$. The third equivalence is the fact that the rows for $x_i$ in $M_1$ and for $f_i$ in $M_1'$ coincide. This is by the fact that $x_i$ and $f_i$ mark at most one place $[x_i{=}a_i]$ and $[f_i{=}a_i]$ by **(SM2)**, and by the fact that this name $a_i$ coincides. The reasoning for $\sigma'(f) = \sigma'(x)$ with $x \in dom(\sigma_2 \cup \ldots \cup \sigma_n)$ is similar. $\qquad\square$

We now turn to the reverse direction and argue that $\underline{G}$ can imitate race free transition sequences enabled by $M$.

**Lemma 6.5.** *Let $(\underline{G}, M) \in \mathcal{B}$. For all $M' \in Reach_{Stbl}(N(F))$ so that $M \Rightarrow M'$ there is a process $\underline{G'}$ with $\underline{G} \hookrightarrow \underline{G'}$ and $(\underline{G'}, M') \in \mathcal{B}$.*

*Proof.* A race free transition sequence $M \Rightarrow M'$ corresponds to a communication among two processes (1), to an identifier call (2), or to a silent action (3). We only consider the first case, the remaining two are along similar lines.

**Case 1: Communication**    We reconstruct the race free transition sequence $M \Rightarrow M'$ to derive information about the shape of $M$ and $M'$. Since we model a communication, we have $M(S_1, \lambda_1) = 1$ in the net $N(S_{Init,1})$ with $S_1 = \overline{x_1}\langle y_1 \rangle.S_1' + \ldots$. Similarly, $M(S_2, \lambda_2) = 1$ in $N(S_{Init,2})$ with $S_2 = x_2(y_2).\nu r.S_2' + \ldots$. Here, $S_1'$ and $S_2'$ are meant to be choices or identifier calls. Thus, again the first two processes communicate and the second generates a fresh name. The race free transition sequence $M \rightarrow^+ M'$ in $N(F)$ is now $(t_1, t_2)\, t_r$ where

$$t_1 = (S_1, \lambda_1) \xrightarrow[\{test([x_1{=}n_1]), test([y_1{=}n_2])\}]{send(n_1, n_2)} (S_1', \lambda_1')$$

$$t_2 = (S_2, \lambda_2) \xrightarrow[\{test([x_2{=}n_1]), map([y_2{=}n_2],)\}]{rec(n_1, n_2)} (\nu r.S_2', \lambda_2 \cdot \lambda_2')$$

$$t_r = (\nu r.S_2', \lambda_2 \cdot \lambda_2') \xrightarrow[\{map(r, n_r)\}]{\tau} (S_2', \lambda_2 \cdot \lambda_2').$$

So we assume the communication is on a fresh channel $n_1$ and sends a fresh name $n_2$. The cases where $x_1$ or $y_1$ are mapped to public names are similar.

For marking $M$, the test commands that label transition $(t_1, t_2)$ allow us to conclude the marking in Line (6.8).

$$M([x_1{=}n_1]) = 1 \quad M([x_2{=}n_1]) = 1 \quad M([y_1{=}n_2]) = 1 \tag{6.8}$$

$$M([y_2{\neq}n_2]) = 1 \quad M([y_2{=}a]) = 0 \quad \forall a \in \mathcal{P} \cup \mathcal{N}. \tag{6.9}$$

In the following Line (6.9), the implementation of mapping requires a token on $[y_2{\neq}n_2]$. By **(SM3)**, this only gives $M([y_2{=}n_2]) = 0$. We derive that actually all $[y_2{=}a]$ are unmarked as follows. We have $(\underline{G}, M) \in \mathcal{B}$, which means $M$ is known to correspond to a process. This process has a substitution that does not contain $y_2$ in its domain. This is due to **(DOM)** in combination with the fact that $y_2$ is bound. Constraint **(COR1)** yields $M([y_2{=}a]) = 0$ for all $a \in \mathcal{P} \cup \mathcal{N}$.

$$M([x{\neq}n_r]) = 1 \; \forall x \in \mathcal{I} \cup \mathcal{F} \cup \{r_*\} \quad M([x{=}n_r]) = 0 \; \forall x \in \mathcal{I} \cup \mathcal{F} \cup \mathcal{R} \tag{6.10}$$

$$M([r{=}n]) = 0 \; \forall n \in \mathcal{N}. \tag{6.11}$$

That $t_r$ is enabled gives the first marking in Line (6.10). With **(SM1)** and **(SM3)**, we conclude that no name maps to $n_r$. Like for $y_2$, we get that $r$ does not map to any fresh name, Line (6.11).

In the control flow, marking $M'$ differs from $M$ in that $(S_1', \lambda_1')$ and $(S_2', \lambda_2 \cdot \lambda_2')$ instead of $(S_1, \lambda_1)$ and $(S_2, \lambda_2)$ are marked in $N(S_{Init,1})$ and $N(S_{Init,2})$. For the substitution net, we only give the places on which the marking has changed. The following is immediate from the definition of implementation:

$$M'([y_2{=}n_2]) = 1 \qquad M'([y_2{\neq}n_2]) = 0 \qquad M'([r{=}n_r]) = 1 \qquad M'([r_*{\neq}n_r]) = 0.$$

$M' = M_1' \cup M_2'$ is stable; moreover, marking $M_1'$ of $N_{Subst}$ is indeed a substitution marking.

We now argue that (1.a) there is $G' \in Reach(F)/_{\equiv}$ with $\underline{G} \hookrightarrow \underline{G}'$ and (1.b) $(\underline{G}', M') \in \mathcal{B}$.

**Claim 1.a: There is $G' \in Reach(F)/_{\equiv}$ so that $\underline{G} \hookrightarrow \underline{G}'$** We assume that $\underline{G}$ and $M_1 \cup M_2$ are related by $\mathcal{B}$. Hence, there is a process in normal form that satisfies

$$G \equiv \nu \tilde{r}.(S_1 \sigma_1 \mid S_2 \sigma_2 \mid \ldots \mid S_n \sigma_n).$$

From marking $M_1 \cup M_2$, we now derive the following information:

$$S_1 = \overline{x_1}\langle y_1 \rangle.S_1' + \ldots \quad S_2 = x_2(y_2).\nu r.S_2' + \ldots \quad \sigma_1(x_1) = \sigma_2(x_2) \in \tilde{r} \quad \sigma_1(y_1) \in \tilde{r}.$$

The equalities on $S_1$ and $S_2$ are due to the markings of the nets $N(S_{Init,1})$ and $N(S_{Init,2})$. For the substitution, we make use of the fact that $M_1$ corresponds to $\sigma_1 \cup \ldots \cup \sigma_n$. We have $M_1([x_1{=}n_1]) = 1 = M_1([x_2{=}n_1])$ with $n_1 \in \mathcal{N}$. Since $x_1$ and $x_2$ are bound to at most one name by **(SM2)**, this allows us to conclude that the markings of $[x_1{=}n]$ and $[x_2{=}n]$ coincide for all names $n \in \mathcal{N}$. Hence, we get $\sigma_1(x_1) = \sigma_2(x_2)$ by **(COR4)**. By **(DOM)**, we have that $\sigma_1(x_1) \in \tilde{r} \cup \mathcal{P}$. If $\sigma_1(x_1)$ was in $\mathcal{P}$, we had $M_1([x_1{=}\sigma_1(x_1)]) = 1$ by **(COR2)**. This is not the case, hence $\sigma_1(x_1) \in \tilde{r}$. For $y_1$, the reasoning is similar. We already mentioned above that $\{y_2, r\} \notin dom(\sigma_1 \cup \ldots \cup \sigma_n)$.

The normal form process has a reaction to

$$G' \stackrel{\mathrm{df}}{=} \nu \tilde{r}.a_r.(S_1' \sigma_1 \mid S_2' \sigma_2' \mid \ldots \mid S_n \sigma_n) \quad \text{with} \quad \sigma_2' \stackrel{\mathrm{df}}{=} \sigma_2\{\sigma_1(y_1)/y_2\}\{a_r/r\}.$$

Hence, $\underline{G} \hookrightarrow \underline{G}'$. Since $G$ was reachable from $F$, we have $G'$ reachable from $F$. Moreover, we already argued in the proof of Lemma 6.4 that $G'$ has the required normal form.

**Claim 1.b:** $(\underline{G'}, M_1' \cup M_2') \in \mathcal{B}$    For the threads, the reasoning is as in Lemma 6.4. We check that $M_1'$ corresponds to $\sigma' \stackrel{\mathrm{df}}{=} \sigma_1 \cup \sigma_2' \cup \ldots \cup \sigma_n$. **(COR1)** to **(COR3)** are as before. For **(COR4)**, we consider $r \neq x \in dom(\sigma')$. We have $\sigma'(r) \neq \sigma'(x)$, which coincides with the fact that $M_1'([r{=}n_r]) = 1$ and $M_1'([x{=}n_r]) = 0$.    $\square$

*Proof of Theorem 6.2.* We show that $\mathcal{B}$ relates $\underline{F}$ and $M_0$. The transitions can be mimicked due to Lemmas 6.4 and 6.5. By our assumptions, we have $F = \nu\tilde{r}.(S_{Init,1}\sigma_1 \mid \ldots \mid S_{Init,n}\sigma_n)$. Here, $S_{Init,i}$ are choices or calls that have been derived from artificial defining equations. Moreover, $\sigma_i : \tilde{f}_{Init,i} \to \tilde{r} \cup \mathcal{P}$ with $dom(\sigma_i) = \tilde{f}_{Init,i} = fn(S_{Init,i}) \cup \emptyset$. This shows **(DOM)**, and concludes the proof that $F$ is in normal form. For the initial marking $M_0 = M_{0,1} \cup M_{0,2}$ of $N(F)$, we have that $M_{0,1}$ corresponds to $\sigma_1 \cup \ldots \cup \sigma_n$ as needed. In the control flow nets $N(S_{Init,i})$, we have the necessary tokens on $(S_{Init,i}, \varepsilon)$.    $\square$

## 7. Optimisation of the translation

In this section, we propose optimisations of the translation, which can significantly reduce the size of the resulting safe PN and increase the efficiency of subsequent model checking.

### 7.1. **Communication splitting.**
Recall the size of the PN resulting from our translation is dominated by the number of transitions modelling communication. We now propose a method to significantly decrease this number. It actually reduces the asymptotic worst case size from $O(\|F\|^4)$ down to $O(\|F\|^3)$. Furthermore, its straightforward generalisation yields a polynomial translation from polyadic $\pi$-calculus to safe PNs, see Sect. 8.

The idea is to model the communication between potentially synchronisable actions $\overline{a}\langle b \rangle$ and $x(y)$ not by a single step but by a pair of steps. The first checks that $a$ and $x$ are mapped to the same value by the substitution, and the second maps $y$ to the value of $b$.

Assume $\overline{a}\langle b \rangle$ and $x(y)$ correspond to stub transitions $t'$ and $t''$. To implement the decomposition, we create a control place $p_{middle}$ 'in the middle' of the communication and two sets of transitions, $t_i^1$ and $t_j^2$. The transitions $t_i^1$, created for each $i \in dom(a) \cap dom(x)$, work as follows. Each $t_i^1$

- consumes tokens from the input places of $t'$ and $t''$ and produces a token on $p_{middle}$;
- checks by read arcs that $[a = i]$ and $[x = i]$ are marked (i.e. the substitution maps $a$ and $x$ to the same value $i$ and thus the synchronisation is possible).

The transitions $t_j^2$, created for each $j \in dom(b) \cap dom(y)$, work as follows. Each $t_j^2$

- consumes a token from $p_{middle}$ and produces tokens on the output places of $t'$ and $t''$;
- checks by a read arc that $[b = j]$ is marked, consumes a token from $[y \neq j]$ (if this place exists) and produces a token on $[y = j]$ (mapping thus in the substitution $y$ to $j$, i.e. to the value of $b$).

If the synchronisation is possible in the current state of the system (i.e. $a$ and $x$ have the same value), exactly one of the transitions $t_i^1$ is enabled; else none of these transitions is enabled. Once some $t_i^1$ fires, exactly one of the transitions $t_j^2$ becomes enabled.

7.2. **Abstractions of names.** In the substitution net described in Sect. 3.1, each bound name and formal parameter is represented by a separate row of places. In practice, it is often the case that some bound names and formal parameters can never be simultaneously active, and so can share the same row of places.

We have implemented a simple sharing scheme by introducing an equivalence $\sim$ on the set of bound names and formal parameters, such that if two names are equivalent then they cannot be simultaneously active. Then the rows of the substitution table will correspond to the equivalence classes of $\sim$, and for each bound name and formal parameter we will introduce the *abstraction* operator *abs*, mapping the name to the corresponding equivalence class of $\sim$. Now the operations on the substitution (initialisation of a restricted name, remapping and unmapping) can be performed on the abstractions of names rather than the names themselves.

A possible choice of equivalence $\sim$ and the related abstraction is as follows. For each name $b \in \mathcal{R} \cup \mathcal{I} \cup \mathcal{F}$, we denote by $thread(b)$ the thread where $b$ is defined (note that due to **(NC)** and the assumption that threads do not share defining equations, $thread(b)$ is unique). Furthermore, we define

$$type(b) \stackrel{\mathrm{df}}{=} \left\{ \begin{array}{ll} 0 & \text{if } b \in \mathcal{R} \\ 1 & \text{otherwise (i.e. if } b \in \mathcal{I} \cup \mathcal{F}), \end{array} \right.$$

and $depth(b)$ to be the number of names $b' \in \mathcal{R} \cup \mathcal{I} \cup \mathcal{F}$ in whose scope $b$ resides and such that $type(b) = type(b')$. Then the abstraction of $b$ can be defined as a tuple

$$abs(b) \stackrel{\mathrm{df}}{=} \Big( thread(b), type(b), depth(b) \Big),$$

and two names are considered equivalent wrt. $\sim$ iff their abstractions coincide. This equivalence ensures that two distinct names related by it belong to the same thread, and that their scopes lie within either different defining equations or different branches of some choice operator, and so the names cannot be simultaneously active. Other choices of $\sim$ and *abs* are also possible, and we plan to explore them in our future work.

7.3. **Better overapproximations for name domains.** Recall that the domain of a bound name or formal parameter is an overapproximation of the set of values from $\mathcal{P} \cup \mathcal{N}$ that it can take. While the rough overapproximation proposed in Sect. 3 is sufficient to make the translation polynomial, its quality can be improved by static analysis, resulting in a much smaller PN. In particular, the number of synchronisations between communication actions as well as the number of transitions implementing each communication may be reduced significantly. Furthermore, the number of transitions implementing parameter passing in calls and the number of places in $N_{Subst}$ can also decrease substantially.

We outline a simple iterative procedure to compute better overapproximations. We start by setting $dom(p) \stackrel{\mathrm{df}}{=} \{p\}$ for each public name $p$. For each restricted name $r$ we set $dom(r) \stackrel{\mathrm{df}}{=} \{r\}$, interpreting this as that the values of $r$ are taken from the set $\mathcal{N}_r$ of unique values (the procedure never looks inside $\mathcal{N}_r$, and only exploits the fact that the names from $\mathcal{N}_r$ are different from all the other names). The domains of these names are fixed and will not be changed by the procedure. The domains of all other names occurring in the FCP are initialised to $\emptyset$; they will grow monotonically during the run of the procedure, converging to some overapproximations.

Each iteration of the procedure identifies two actions, $\overline{a}\langle b\rangle$ and $x(y)$, that satisfy the following conditions: (i) the actions belong to different threads of the FCP; (ii) $dom(a)$ and $dom(x)$ overlap; and (iii) $dom(b) \not\subseteq dom(y)$. If no two such actions can be found, the procedure stops, returning the current values of the domains. Otherwise, $dom(y)$ is replaced by its union with $dom(b)$. Intuitively, the above conditions check that the two actions can potentially synchronise, and so $y$ can be mapped to the value of $b$, and so its domain has to include that of $b$.

7.4. **Better overapproximation for** $|\mathcal{N}|$. The cardinality of the set $\mathcal{N}$ is an important parameter of the translation, affecting the efficiency of almost all its aspects. While the rough overapproximation proposed in Sect. 3 (taking $|\mathcal{N}|$ to be the total number of bound names and formal parameters) is sufficient to make the translation polynomial, a better one can make the translation much more practical and amenable to model checking.

In the worst case, all names from $\mathcal{R} \cup \mathcal{I} \cup \mathcal{F}$ can be assigned different values from $\mathcal{N}$. To improve the overapproximation of Sect. 3, we observe that in many cases not all such names can be simultaneously active, i.e. it is enough to overapproximate the number of such names that can be simultaneously active. Hence we propose the following improved overapproximation of $|\mathcal{N}|$. If there are no occurrences of the restriction operator in the FCP, $|\mathcal{N}| \overset{\mathrm{df}}{=} 0$. Else, for each thread we compute the maximal number of names from $\mathcal{R} \cup \mathcal{I} \cup \mathcal{F}$ that can be simultaneously active in it,

$$\max\{|\mathit{fn}\left(S'\right) \cup \lambda| \mid (S, \varepsilon) \text{ is a defining process and } (S, \varepsilon) \twoheadrightarrow^* (S', \lambda)\},$$

and set $|\mathcal{N}|$ to the sum of these numbers.

The number of names from $\mathcal{R} \cup \mathcal{I} \cup \mathcal{F}$ that can be simultaneously active in a thread can be computed by separately determining this parameter for each of the defining equations belonging to this thread, as well as the subterm of the main process corresponding to this thread, and taking the maximum of these values. Since these $\pi$-calculus expressions are sequential, their parse trees can only have the $+$ operator in every node where a branching occurs, and so the sought value is simply the maximum of the numbers of active names from $\mathcal{R} \cup \mathcal{I} \cup \mathcal{F}$ in the leafs of this parse tree. Furthermore, the names whose domains contain no restricted names can be ignored by this analysis.

7.5. **Sharing subnets for unmapping names.** When we call $K\lfloor\tilde{a}\rfloor$, some names have to be unmapped in the substitution. The subnet for unmapping a particular name can be shared by all points where such unmapping is necessary. This reduces the size of the resulting PN. This optimisation is especially effective when name abstractions (see above) are used, as the sharing increases significantly in such a case.

7.6. **Re-ordering parameters of calls.** Consider the FCP $K\lfloor a, b\rfloor$ with:

$$\begin{array}{lcl} K(f_1, f_2) := L\lfloor f_2, f_1\rfloor & & K(f_1, f_2) := L\lfloor f_1, f_2\rfloor \\ L(f_3, f_4) := K\lfloor f_4, f_3\rfloor & \Rightarrow & L(f_4, f_3) := K\lfloor f_4, f_3\rfloor \end{array}$$

With the definition on the left, when name abstractions are computed, the equivalence relation has two classes, $\{f_1, f_3\}$ and $\{f_2, f_4\}$. Hence, the substitution has to be modified every time the calls are performed, as the call parameters keep getting flipped. If the order of the formal parameters in one of the defining equations is changed (together with the order

of the factual parameters in the corresponding calls), as shown on the right, the substitution would not require any changes. This significantly reduces the size of the resulting net.

This example illustrates that the order of formal parameters in the defining equations matters, and the translation can gain savings by changing this order. Searching for the best order of formal parameters can be formulated as an optimisation problem, with the cost function being the total number of changes required in the substitution for all the calls.

7.7. **Dropping the restrictions in the main term of the FCP.** All restrictions in the main term of the FCP can be dropped, making the formerly restricted names public. Note that due to **(NC)**, this does not introduce name clashes. The transformation yields a bisimilar $\pi$-calculus process, but the corresponding PN becomes smaller.

7.8. **Separate pools of values for restricted names.** Creation of names introduces arbitration between the values in $\mathcal{N}$ (see Sect. 3.2): a name that is currently unused has to be chosen to initialise the given restricted name. Such arbitration can adversely affect the efficiency of some model checking methods.

It is possible to eliminate such arbitration by splitting $\mathcal{N}$ into several pools, one for each thread, and initialise restricted names only from the corresponding pool, by sequentially looking for the first unused value. This however increases the size of the resulting PN. Moreover, if symmetry reduction is used in model checking, the problem vanishes.

7.9. **Using symmetries.** The translation introduces a number of symmetries in the PN: (i) the values in $\mathcal{N}$, and thus the corresponding columns of the substitution (see Fig. 3), are interchangeable; and (ii) when enforcing the assumption that threads do not share defining equations as explained in Sect. 2, some equations are replicated.

It is desirable to exploit these symmetries during model checking. In particular, this would efficiently handle the arbitration that arises when a value from $\mathcal{N}$ has to be chosen to initialise a restriction. If symmetries are used, all the immediate successor states of the arbitration are equivalent, and only one of them has to be explored further.

7.10. **Translation to different PN classes.** Our translation produces a safe PN, as this PN class is particularly suitable for algorithmic verification. However, if the model checking method can cope with more powerful PN classes, the following changes can be made.

**Translation to bounded PNs**   For each $val \in \mathcal{N}$, we can fuse the places $[var \neq val]$, where $var \in \{r_*\} \cup \mathcal{I} \cup \mathcal{F}$, into one place $[* \neq val]$. We thus replace $|\mathcal{N}| \cdot (|\mathcal{I}| + |\mathcal{F}| + 1)$ safe places with $|\mathcal{N}|$ places of capacity $|\mathcal{I}| + |\mathcal{F}| + 1$. It is still possible to perform all the necessary operations with the substitution. In particular, to find a value $val \in \mathcal{N}$ to which no bound name or formal parameter is currently mapped, and map a given restricted name $r_k$ to $val$, the PN transition performing the initialisation has to:

- consume by a weighted arc $|\mathcal{I}| + |\mathcal{F}| + 1$ tokens from $[* \neq val]$ (checking thus that $val$ is not assigned to any name) and return by a weighted arc $|\mathcal{I}| + |\mathcal{F}|$ tokens;
- produce a token at $[r_k = val]$.

**Translation to coloured PNs**   In this case, the symmetries present in the PN can be used to fold it. In particular:

- The values in $\mathcal{N}$ are interchangeable, and so the corresponding columns of the substitution can be folded into one column, by giving the tokens corresponding to the elements of $\mathcal{N}$ unique colours.
- Instead of enforcing the assumption that the threads do not share defining equations (see Sect. 2), one can use coloured control tokens that are unique for each thread.

## 8. EXTENSIONS

We now generalise the translation to some often used extensions of $\pi$-calculus, viz. to polyadic communication and synchronisation, and to match and mismatch operators.

**Polyadic communication**   Polyadic communication exchanges multiple names in one reaction. Intuitively, a sending prefix $\overline{a}\langle x_1 \ldots x_n \rangle$ and a receiving prefix $b(y_1 \ldots y_n)$ (with all $y_i$ being different names) can synchronise iff $\sigma(a) = \sigma(b)$. After synchronisation each $y_i$ gets the value of $x_i$. Formally,

$$a(\tilde{y}).S_1 + \ldots \mid \overline{a}\langle \tilde{x} \rangle.S_2 + \ldots \to S_1\{\tilde{x}/\tilde{y}\} \mid S_2 \quad \text{if } |\tilde{x}| = |\tilde{y}|.$$

A polynomial translation of this extension generalises the communication splitting idea described in Sect. 7. We perform the communication in stages. At the first step, one checks that $a$ and $b$ are mapped to the same value by the substitution. The subsequent steps map, one-by-one, $y_i$ to the value of $x_i$ in the substitution.

**Polyadic synchronisation**   Dual to polyadic communication is polyadic synchronisation [4]. When sending a message, this operation synchronises on multiple channels instead of just one. Formally,

$$\tilde{a}(y).S_1 + \ldots \mid \overline{\tilde{a}}\langle x \rangle.S_2 + \ldots \to S_1\{x/y\} \mid S_2.$$

Polyadic synchronisation captures, in a clean formalism, expressive features like locality. To extend our translation to polyadic synchronisation, we check, one-by-one, that the channel bindings of both processes match.

In the presence of polyadic synchronisation, the relationship between the FCP and its PN translation is subtle, since false deadlocks may be introduced. For example, in $a_1 \cdot a_2(y) \mid \overline{b_1 \cdot b_2}\langle x \rangle$ the evaluation may find $a_1$ and $b_1$ bound to the same name, $\sigma(a_1) = \sigma(b_1)$, while $a_2$ and $b_2$ do not match. In this case the resulting PN will get stuck in the middle of the evaluation. This does not happen in the original $\pi$-calculus process. Nevertheless, such false deadlocks can easily be distinguished from real ones, and so the resulting PN is still suitable for model checking. An alternative is to use the idea of the construction in Sect. 10, which avoids false deadlocks.

**Match and mismatch operators**   The match and mismatch operators are a common extension of $\pi$-calculus. Intuitively, the process $[x = y].S$ behaves as $S$ if $\sigma(x) = \sigma(y)$ and does nothing otherwise, and the process $[x \neq y].S$ behaves as $S$ if $\sigma(x) \neq \sigma(y)$ and does nothing otherwise. To handle these operators, we extend the construction of $N(S_{Init})$ with the following transitions. For each $a \in \mathcal{P} \cup \mathcal{N}$, we have

$$([x = y].S, \lambda) \xrightarrow[\{test([x=a]),test([y=a])\}]{\tau} (S, \lambda \cdot \lambda') \qquad ([x \neq y].S, \lambda) \xrightarrow[\{test([x=a]),test([y\neq a])\}]{\tau} (S, \lambda \cdot \lambda'),$$

where $\lambda'$ contains the names from $\{x, y\} \setminus \mathit{fn}(S)$. For the latter rule, new places $[x{\neq}a]$ complementing $[x{=}a]$ may have to be introduced to $N_{Subst}$. The relationship between FCP and PN is similar to the case of polyadic synchronisation.

## 9. EXPERIMENTAL RESULTS

To demonstrate the practicality of our translation-based approach to $\pi$-calculus verification, we implemented the encoding of FCPs into safe PNs in the tool FCP2PN[2], and used the resulting nets for model checking a number of benchmark systems.[2]

The *NESS (Newcastle E-Learning Support System)* series of benchmarks models an electronic coursework submission system [12]. The model consists of a teacher process $T$ composed in parallel with $k$ students $S$ (the system can be scaled up by increasing the number of students) and an environment process $ENV$. Every student has its own local channel for communication, $h_i$, and all students share the channel $h$:

$$\nu h.\nu h_1 \ldots \nu h_k.\Big( T\lfloor nessc, h_1, \ldots, h_k \rfloor \mid \prod_{i=1}^{k} S\lfloor h, h_i \rfloor \mid ENV \lfloor nessc \rfloor \Big) \, .$$

The students are supposed to submit their work for assessment to *NESS*. The teacher passes the channel *nessc* of the system to all students, $\overline{h_i}\langle nessc \rangle$, and then waits for the confirmation that they have finished working on the assignment, $h_i(x_i)$. After receiving the *NESS* channel, $h_i(nsc)$, students non-deterministically organise themselves in pairs. To do so, they send their local channel $h_i$ on $h$ and at the same time listen on $h$ to receive a partner, $\overline{h}\langle h_i \rangle \ldots + h(x) \ldots$ When they finish, exactly one student of each pair sends two channels (the own channel $h_i$ and the channel received from the partner) to the support system, $\overline{nsc}\langle h_i \rangle.\overline{nsc}\langle x \rangle$, which give access to their completed joint work. These channels are received by the *ENV* process. The students finally notify the teacher about the completion of their work, $\overline{h_i}\langle fin \rangle$. Thus, the system is modelled by:

$$T(nessc, h_1, \ldots, h_k) := \prod_{i=1}^{k} \overline{h_i}\langle nessc \rangle.h_i(x_i).\mathbf{0}$$

$$S(h, h_i) := h_i(nsc).(\overline{h}\langle h_i \rangle.\overline{h_i}\langle fin \rangle.\mathbf{0} + h(x).\overline{nsc}\langle h_i \rangle.\overline{nsc}\langle x \rangle.\overline{h_i}\langle fin \rangle.\mathbf{0})$$

$$ENV(nessc) := nessc(y_1). \ \ldots \ .nessc(y_k).\mathbf{0}$$

To distinguish proper termination from deadlocks (where some processes are stuck waiting for a communication), a new transition is added to the PN that creates a loop at the state corresponding to successful termination. Obviously, the system successfully terminates iff the number of students is even, i.e. they can be organised into pairs.

The *DNESS* model is a refined version of *NESS*, with deterministic pairing of students. Thus, the number of students is always even, and these benchmarks are deadlock-free.

The $CS(m,n)$ series of benchmarks models a client-server system with one server, $n$ clients, and the server spawning $m$ sessions that handle the clients' requests:

$$CLIENT(url) := \nu ip.\overline{url}\langle ip \rangle.ip(s).s(x).CLIENT\lfloor url \rfloor$$

$$SERVER(url, getses) := url(y).getses(s).\overline{y}\langle s \rangle.SERVER\lfloor url, getses \rfloor$$

$$SESSION(getses) := \nu ses.\overline{getses}\langle ses \rangle.\overline{ses}\langle ses \rangle.SESSION\lfloor getses \rfloor$$

$$\nu getses\Big( SERVER(url, getses) \mid \prod_{i=1}^{m} SESSION(getses) \mid \prod_{i=1}^{n} CLIENT(url) \Big)$$

On a client's request, the server creates a new session using the *getses* channel, $getses(s)$. A session is modelled by a *SESSION* process. It sends its private channel $\nu ses$ along the *getses*

---

| | Process size | | | Safe PN | | Dlck | | | Process size | | | Safe PN | | Dlck |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Problem | FCP | nfFCP | $|\mathcal{N}|$ | $|P|$ | $|T|$ | [sec] | Problem | FCP | nfFCP | $|\mathcal{N}|$ | $|P|$ | $|T|$ | [sec] |
| *NESS* (04) | 110 | 110 | 0 | 137 | 145 | 0.02 | $CS(2,1)$ | 45 | 54 | 7 | 138 | 149 | 1.01 |
| *NESS* (05)† | 137 | 137 | 0 | 196 | 246 | 0.09 | $CS(2,2)$ | 48 | 68 | 10 | 243 | 320 | 0.16 |
| *NESS* (06) | 164 | 164 | 0 | 265 | 385 | 0.16 | $CS(3,2)$ | 51 | 80 | 11 | 284 | 431 | 1.28 |
| *NESS* (07)† | 191 | 191 | 0 | 344 | 568 | 0.45 | $CS(3,3)$ | 54 | 94 | 14 | 428 | 728 | 3.67 |
| *DNESS* (06) | 118 | 118 | 0 | 157 | 103 | 0.02 | $CS(4,4)$ | 60 | 120 | 18 | 663 | 1368 | 11.73 |
| *DNESS* (08) | 157 | 157 | 0 | 241 | 169 | 0.05 | $CS(5,5)$ | 66 | 146 | 22 | 948 | 2288 | 46.61 |
| *DNESS* (10) | 196 | 196 | 0 | 341 | 251 | 0.13 | *GSM* | 175 | 231 | 12 | 636 | 901 | 4.39 |
| *DNESS* (12) | 235 | 235 | 0 | 457 | 349 | 2.27 | *GSM*' | 174 | 230 | 0 | 355 | 503 | 3.09 |
| *DNESS* (14) | 274 | 274 | 0 | 589 | 463 | 1.71 | *PHONES* | 157 | 157 | 0 | 131 | 94 | 0.01 |

Table 1: Experimental results.

channel to the server. The server forwards the session to the client, $\overline{y}\langle s \rangle$, which establishes the private session, and becomes available for further requests. A communication on the channel *ses* terminates the private session. All these benchmarks are deadlock-free.

The *GSM* benchmark is a specification of the handover procedure in the GSM Public Land Mobile Network. We use the well-known $\pi$-calculus model from [22], with one mobile station, two base stations, and one mobile switching. We also studied a variant *GSM*' where a restriction in the sender process is dropped: the sender keeps sending the same message instead of generating a new one every time. Since the content of the message is not important, this change is inconsequential from the modelling point of view. However, it significantly reduces the size of the PN. Indeed, the modified FCP is restriction-free, and so $\mathcal{N} = \emptyset$.

The *PHONES* benchmark is a classical example taken from [19], modelling a handover procedure for mobile phones communicating with fixed transmitters, where the phones have to switch their transmitters on the go.

The experimental results are given in Table 1, with the columns showing from left to right: name of the case study († indicates deadlocks); sizes of the original FCP and its normal form (see Sect. 2), together with the cardinality of $\mathcal{N}$ determined by static analysis; number of places and transitions in the resulting safe PN; and deadlock checking time.

The experiments were conducted on a PC with an Intel Core 2 Quad Q9400 2.66 GHz processor (a single core was used) and 4G RAM. The deadlock checking was performed with the LoLA tool,[3] configured to assume safeness of the PN (`CAPACITY 1`), use the stubborn sets and symmetry reductions (`STUBBORN`, `SYMMETRY`), compress states using P-invariants (`PREDUCTION`), use a light-weight data structure for states (`SMALLSTATE`), and check for deadlocks (`DEADLOCK`). The FCP to PN translation times were negligible ($< 0.1$ sec in all cases) and so are not reported.

The experiments indicate that the sizes of the PNs grow moderately with the sizes of the FCPs, and that the PNs are suitable for efficient verification: deadlock checking took less than a minute in all examples.

---

[3] Available from `http://service-technology.org/tools/lola`.

## 10. PN to FCP translation

We now study a translation in the reverse direction. We translate safe PNs into FCPs that are weakly bisimilar and thus do not introduce false deadlocks. Moreover, no livelocks are introduced. This improves over the translations in [1, 15].

The motivation for proposing this translation is twofold. First, we obtain a PSPACE lower bound for verification problems on FCPs, see Sect. 1.1. Second, the idea can be adapted to translate polyadic communication and match/mismatch operators into safe PNs in a faithful way, see Sect. 8.

The main difficulty is faithful modelling of $n$-ary synchronisations in PNs by a sequence of binary synchronisations in $\pi$-calculus. We address it in three steps. First, we give a folklore translation of PNs into FCPs [1, 15], whose advantage is its simplicity. The drawback is that it can introduce false deadlocks. To fix this problem, we develop a second translation, which yields a weakly bisimilar FCP. This in particular implies the absence of false deadlocks. The translation may, however, introduce livelocks. We eliminate these livelocks in our third translation using scheduling.

All these translations are linear. Moreover, they use a very restricted set of $\pi$-calculus capabilities: communications do not pass information (in particular, no reference passing is used), no restricted names are used, the calls do not have any parameters, no $\tau$ actions are used, and the result is a safe FCP (see Sect. 2). As a consequence, the translations can be adopted to process calculi with weaker communication capabilities, such as CCS [18].

We fix $N = (P, T, F, M_0)$ as the safe PN to be translated. All communications will pass a fixed public name $\varepsilon$, which is not used for any other purpose. Thus, the simplified syntax $\overline{a}$ and $x$ is used instead of $\overline{a}\langle\varepsilon\rangle$ and $x(y)$, respectively. Similarly, since calls do not pass parameters, we write $K$ for $K\lfloor-\rfloor$.

**Blocking translation**   The following translation is inspired by [1, 15]. For each place $p \in P$, there is a separate public channel, also denoted by $p$, and a thread with two defining equations corresponding to the presence and absence of a token in $p$:

$$\text{Marked}_p := \overline{p}.\text{Empty}_p$$

$$\text{Empty}_p := p.\text{Marked}_p$$

A marked place can send a message over channel $p$, which models token consumption, and become empty. Similarly, an empty place can receive a message over $p$, which models token production, and become marked.

For each transition $t \in T$ with $^\bullet t = \{p_1, p_2, \ldots, p_m\}$ and $t^\bullet = \{q_1, q_2, \ldots, q_n\}$, we create a thread with the following defining equation:

$$\text{Tran}_t := p_1.p_2.\ \ldots\ .p_m.\overline{q_1}.\overline{q_2}.\ \ldots\ .\overline{q_n}.\text{Tran}_t$$

Intuitively, the transition consumes tokens, one-by-one, from the places in $^\bullet t$ by receiving messages over the corresponding channels. Then it produces tokens, one-by-one, in the places in $t^\bullet$, by sending messages over the corresponding channels. Since the PN is safe, it is guaranteed that the places in $t^\bullet$ were empty.

The initial term of the FCP is as follows:

$$\prod_{p \in M_0} \text{Marked}_p \ \Big|\ \prod_{p \in P \setminus M_0} \text{Empty}_p \ \Big|\ \prod_{t \in T} \text{Tran}_t$$

Clearly, the size of the FCP is linear in the size of the original PN. However, this basic translation can introduce false deadlocks. Indeed, it is possible for a thread $\text{Tran}_t$ to

consume some, but not all tokens from ${}^\bullet t$, and become blocked. In such a case the already consumed tokens are not returned back to the corresponding places in ${}^\bullet t$, which can prevent other transitions from firing. In practice, one can deal with this problem by declaring the process in which some $\mathrm{TRAN}_t$ is not in the beginning of its control flow as unstable, and considering only stable processes.

**Non-blocking translation**    To fix the blocking translation, we change the specification of the place and transition processes. The idea is to let transitions detect that some tokens in the preset are missing, and return the already consumed tokens in such a case.

The initial term is the same as above. For each place $p \in P$ we create another public channel $p^\times$, over which the place process can communicate a failure to consume a token if it is empty. The specification of the place process is amended as follows:

$$\mathrm{EMPTY}_p := p.\mathrm{MARKED}_p + \overline{p^\times}.\mathrm{EMPTY}_p$$

The definition of $\mathrm{MARKED}_p$ remains the same.

The specification of the transition process is now as follows:

$$\mathrm{TRAN}_t := \mathrm{TRYCONS}_t^1$$

$$\mathrm{TRYCONS}_t^1 := p_1.\mathrm{TRYCONS}_t^2 + p_1^\times.\mathrm{RET}_t^0$$
$$\mathrm{TRYCONS}_t^2 := p_2.\mathrm{TRYCONS}_t^3 + p_2^\times.\mathrm{RET}_t^1$$
$$\ldots$$
$$\mathrm{TRYCONS}_t^m := p_m.\mathrm{PROD}_t + p_m^\times.\mathrm{RET}_t^{m-1}$$

$$\mathrm{RET}_t^0 := \mathrm{TRAN}_t$$
$$\mathrm{RET}_t^1 := \overline{p_1}.\mathrm{RET}_t^0$$
$$\ldots$$
$$\mathrm{RET}_t^{m-1} := \overline{p_{m-1}}.\mathrm{RET}_t^{m-2}$$

$$\mathrm{PROD}_t := \overline{q_1}.\overline{q_2}.\ \ldots\ .\overline{q_n}.\mathrm{TRAN}_t$$

Initially, $\mathrm{TRAN}_t$ tries to consume all the tokens from ${}^\bullet t = \{p_1, p_2, \ldots, p_m\}$, one-by-one, by calling $\mathrm{TRYCONS}_t^1$. $\mathrm{TRYCONS}_t^i$ communicates with the thread for $p_i$ and either consumes a token from this place (by receiving a message from channel $p_i$) and then calls $\mathrm{TRYCONS}_t^{i+1}$ to consume the remaining tokens, or detects that the place is empty (by receiving a message over channel $p_i^\times$), in which case it calls $\mathrm{RET}_t^{i-1}$ to return the previously consumed tokens, if any. Note that the tokens are returned in the reverse order of their consumption. Once all tokens have been successfully consumed, $\mathrm{PROD}_t$ is called to communicate with the processes for the places in $t^\bullet = \{q_1, q_2, \ldots, q_n\}$, one-by-one, to produce tokens on them.

The size of the FCP is still linear in the size of the original PN, and there are no false deadlocks. In fact, this FCP is weakly bisimilar to the original PN. It may, however, introduce livelocks, e.g. a disabled transition can perpetually try and fail to fire.

**Scheduling translation**    To solve the problem of livelocks, we augment the non-blocking translation with the process $\mathrm{SCHEDULER}$ responsible for the global operation of the net.

Hence, the initial term of the FCP becomes

$$\prod_{p \in M_0} \text{MARKED}_p \mid \prod_{p \in P \setminus M_0} \text{EMPTY}_p \mid \prod_{t \in T} \text{TRAN}_t \mid \text{SCHEDULER}$$

The simulation is performed in rounds. In each round the scheduler tries to execute a single transition, trying them one-by-one in some non-deterministically chosen order. If none of them can be executed, the scheduler blocks. Otherwise, some transition is executed and a new round starts.

Assuming that $T = \{t_1, t_2, \ldots, t_k\}$, the specification of the scheduler is as follows:

$$\text{SCHEDULER} := \overline{go}.(failure.\text{SCHEDULER} + success.\overline{reset_{t_1}}.\overline{reset_{t_2}}.\ \ldots\ .\overline{reset_{t_k}}.\text{SCHEDULER})$$

In each round, the scheduler non-deterministically chooses a transition by communicating with a process $\text{TRAN}_t$ over the public channel $go$. Upon receiving a message from the selected transition about a failed execution attempt, the scheduler chooses another available transition, or blocks if there is none (in which case the simulated PN has reached a deadlock state). Upon receiving a message about successful execution of a transition, the scheduler communicates over channels $reset_t$ with every transition process in some fixed order to make them available again. The transition processes that are still available also participate in this communication, but do nothing in response. Then a new round starts.

The specification of the place process remains the same, and that of the transition process is amended as follows:

$$\text{TRAN}_t := reset_t.\text{TRAN}_t + go.\text{TRYCONS}_t^1$$
$$\text{RET}_t^0 := \overline{failure}.reset_t.\text{TRAN}_t$$
$$\text{PROD}_t := \overline{q_1}.\overline{q_2}.\ \ldots\ .\overline{q_n}.\overline{success}.\text{TRAN}_t$$

The definitions of $\text{TRYCONS}_t^i$ and $\text{RET}_t^j$ for $j \neq 0$ remain the same.

Initially, $\text{TRAN}_t$ may receive a reset request over the public channel $reset_t$ and return to the initial state. Alternatively, it competes with the other transitions to communicate over the $go$ channel, which results in a non-deterministic selection of a transition to be executed. Note that the non-deterministic selection is essential here, as any fixed order of selection may perpetually ignore some enabled transition by always executing preceding transitions. The selected transition process then works as in the non-blocking translation above, but additionally communicates its failure or success to the scheduler. In case of failure, the transition is blocked until it is reset by the scheduler with a communication over $reset_t$.

One can see that the resulting FCP is weakly bisimilar with the original PN, and the problem of livelocks is solved.

## 11. CONCLUSIONS

We developed a polynomial translation from finite control $\pi$-calculus processes to safe low-level Petri nets. To our knowledge, this is the first such translation. There is a close correspondence between the control flow of the $\pi$-calculus specification and the resulting PN, and the latter is suitable for practical model checking. The translation has been implemented in the FCP2PN tool, and the experimental results are encouraging.

We have also proposed a number of optimisations allowing one to reduce the size of the resulting PN. Moreover, we have shown how to generalise the translation to more expressive

classes of processes. In particular, we discussed how to handle polyadic communication, polyadic synchronisation, and match/mismatch operators.

In future work, we plan to further improve the translation by a more thorough static analysis, and to incorporate it into different model checking tool-chains, in particular, ones based on PN unfolding prefixes and abstraction-refinement approaches. Moreover, it would be interesting to check if some analog of Theorem 6.2 holds for barbed bisimulation. Furthermore, we would like to consider the labelled semantics of $\pi$-calculus.

## References

[1] R. Amadio and C. Meyssonnier. On decidability of the control reachability problem in the asynchronous $\pi$-calculus. *Nord. J. Comp.*, 9(1):70–101, 2002.

[2] E. Best, R. Devillers, and M. Koutny. *Petri Net Algebra*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2001.

[3] N. Busi and R. Gorrieri. Distributed semantics for the $\pi$-calculus based on Petri nets with inhibitor arcs. *J. Log. Alg. Prog.*, 78(1):138–162, 2009.

[4] M. Carbone and S. Maffeis. On the expressive power of polyadic synchronisation in pi-calculus. *Nord. J. Comp.*, 10(2):70–98, 2003.

[5] L. Cardelli and A. Gordon. Mobile ambients. In *Proc. FOSSACS'98*, volume 1378 of *LNCS*, pages 140–155. Springer, 1998.

[6] M. Dam. Model checking mobile processes. *Inf. Comp.*, 129(1):35–51, 1996.

[7] R. Devillers, H. Klaudel, and M. Koutny. A compositional Petri net translation of general $\pi$-calculus terms. *For. Asp. Comp.*, 20(4–5):429–450, 2008.

[8] C. Dufourd, A. Finkel, and Ph. Schnoebelen. Reset nets between decidability and undecidability. In *Proc. of ICALP*, volume 1443 of *LNCS*, pages 103–115. Springer, 1998.

[9] J. Esparza. Decidability and complexity of Petri net problems—an introduction. In *Lectures on Petri Nets I: Basic Models*, volume 1491 of *LNCS*, pages 374–428. Springer, 1998.

[10] G.-L. Ferrari, S. Gnesi, U. Montanari, and M. Pistore. A model-checking verification environment for mobile processes. *ACM Trans. Softw. Eng. Methodol.*, 12(4):440–473, 2003.

[11] R. Gorrieri and R. Meyer. On the relationship between pi-calculus and finite place/transition Petri nets. In *Proc. of CONCUR*, volume 5170 of *LNCS*, pages 463–480. Springer, 2009.

[12] V. Khomenko, M. Koutny, and A. Niaouris. Applying Petri net unfoldings for verification of mobile systems. In *Proc. of MOCA*, Bericht FBI-HH-B-267/06, pages 161–178. University of Hamburg, 2006.

[13] V. Khomenko and R. Meyer. Checking pi-calculus structural congruence is graph isomorphism complete. In S. Edwards, R. Lorenz, and W. Vogler, editors, *Proc. ACSD'09*, pages 70–79. IEEE Computing Society Press, 2009.

[14] R. Meyer. On boundedness in depth in the $\pi$-calculus. In *Proc. of IFIP TCS*, volume 273 of *IFIP*, pages 477–489. Springer, 2008.

[15] R. Meyer. *Structural Stationarity in the $\pi$-calculus*. PhD thesis, Department of Computing Science, University of Oldenburg, 2009.

[16] R. Meyer. A theory of structural stationarity in the $\pi$-calculus. *Acta Inf.*, 46(2):87–137, 2009.

[17] R. Meyer, V. Khomenko, and T. Strazny. A practical approach to verification of mobile systems using net unfoldings. *Fundam. Inf.*, 94:439–471, 2009.

[18] R. Milner. *A Calculus of Communicating Systems*. Springer, 1980.

[19] R. Milner. *Communicating and Mobile Systems: the $\pi$-Calculus*. CUP, 1999.

[20] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, part I. *Inf. Comp.*, 100(1):1–40, 1992.

[21] U. Montanari and M. Pistore. Checking bisimilarity for finitary $\pi$-calculus. In *Proc. of CONCUR*, volume 962 of *LNCS*, pages 42–56. Springer, 1995.

[22] F. Orava and J. Parrow. An algebraic verification of a mobile network. *For. Asp. Comp.*, 4(6):497–543, 1992.

[23] F. Peschanski, H. Klaudel, and R. Devillers. A Petri net interpretation of open reconfigurable systems. In *Proc. of Petri nets'11*, volume 6709, pages 208–227. Springer, 2011.

[24] M. Pistore. *History Dependent Automata*. PhD thesis, Dipartimento di Informatica, Università di Pisa, 1999.

[25] D. Sangiorgi and D. Walker. *The $\pi$-calculus: a Theory of Mobile Processes*. CUP, 2001.

[26] B. Victor and F. Moller. The mobility workbench: A tool for the $\pi$-calculus. In *Proc. of CAV*, volume 818 of *LNCS*, pages 428–440. Springer, 1994.