A Practical Approach to Verification of Mobile Systems Using Net Unfoldings[⋆]

Roland Meyer[1], Victor Khomenko[2], and Tim Strazny[1]

[1] Department of Computing Science, University of Oldenburg
D-26129 Oldenburg, Germany
e-mail: {Roland.Meyer, Tim.Strazny}@informatik.uni-oldenburg.de

[2] School of Computing Science, University of Newcastle
Newcastle upon Tyne, NE1 7RU, U.K.
e-mail: Victor.Khomenko@ncl.ac.uk

Abstract. In this paper we propose a technique for verification of mobile systems. We translate *finite control processes,* which are a well-known subset of π-Calculus, into Petri nets, which are subsequently used for model checking. This translation always yields bounded Petri nets with a small bound, and we develop a technique for computing a non-trivial bound by static analysis. Moreover, we introduce the notion of *safe processes,* which are a subset of finite control processes, for which our translation yields safe Petri nets, and show that every finite control process can be translated into a safe one of at most quadratic size. This gives a possibility to translate every finite control process into a safe Petri net, for which efficient unfolding-based verification is possible. Our experiments show that this approach has a significant advantage over other existing tools for verification of mobile systems in terms of memory consumption and runtime.
Keywords: finite control processes, safe processes, π-Calculus, mobile systems, model checking, Petri net unfoldings.

1 Introduction

Mobile systems permeate our lives and are becoming ever more important. Ad-hoc networks, where devices like mobile phones, PDAs and laptops form dynamic connections are common nowadays, and the vision of pervasive (ubiquitous) computing, where several devices are simultaneously engaged in interaction with the user and each other, forming dynamic links, is quickly becoming a reality. This leads to the increasing dependency of people on the correct functionality of mobile systems, and to the increasing cost incurred by design errors in such systems. However, even the conventional concurrent systems are notoriously difficult to design correctly because of the complexity of their behaviour, and mobile systems add another layer of complexity due to their dynamical nature. Hence formal methods, especially computer-aided verification tools implementing model

[⋆] Appeared in Proc. of ATPN, volume 5062 of LNCS, pages 327-347. Springer-Verlag, 2008. http://dx.doi.org/10.1007/978-3-540-68746-7_22.

checking (see, e.g., [CGP99]), have to be employed in the design process to ensure correct behaviour.

The π-Calculus is a well-established formalism for modelling mobile systems [Mil99,SW01]. It has an impressive modelling power, but, unfortunately, is difficult to verify. The full π-Calculus is Turing complete, and hence, in general, intractable for automatic techniques. A common approach is to sacrifice a part of the modelling power of π-Calculus in exchange for the possibility of fully automatic verification. Expressive fragments of π-Calculus have been proposed in the literature. In particular *finite control processes (FCPs)* [Dam96] combine an acceptably high modelling power with the possibility of automatic verification.

In this paper, we propose an efficient model checking technique for FCPs. We translate general FCPs into their syntactic subclass, called *safe processes.* In turn, safe processes admit an efficient translation into safe Petri nets — a well-investigated model for concurrent systems, for which efficient model checking techniques have been developed.

This approach has a number of advantages, in particular it does not depend on a concrete model checking technique, and can adapt any model checker for safe Petri nets. Moreover, Petri nets are a *true concurrency* formalism, and so one can efficiently utilise partial-order techniques. This alleviates the main drawback of model checking — the *state explosion* problem [Val98]; that is, even a small system specification can (and often does) yield a huge state space.

Among partial-order techniques, a prominent one is McMillan's (finite prefixes of) Petri Net unfoldings (see, e.g., [ERV02,Kho03,McM92]). They rely on the partial-order view of concurrent computation, and represent system states implicitly, using an acyclic net, called a *prefix.* Many important properties of Petri nets can be reformulated as properties of the prefix, and then efficiently checked, e.g., by translating them to SAT. Our experiments show that this approach has a significant advantage over other existing tools for verification of mobile systems in terms of memory consumption and runtime. The proofs of the results and other technical details can be found in the technical report [MKS08].

2 Basic Notions

In this section, we recall the basic notions concerning π-Calculus and Petri nets.

The π-Calculus We use a π-Calculus with parameterised recursion as proposed in [SW01]. Let the set $\mathcal{N} \overset{\mathrm{df}}{=} \{a, b, x, y \ldots\}$ of *names* contain the channels (which are also the possible messages) that occur in communications. During a process execution the *prefixes* π are successively consumed (removed) from the process to communicate with other processes or to perform silent actions:

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

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

To denote recursive processes, we use *process identifiers* from the set $PIDS \overset{\text{df}}{=} \{H, K, L, \ldots\}$. A process identifier is defined by an equation $K(\tilde{x}) := P$, where $\tilde{x}$ is a short-hand notation for $x_1, \ldots, x_k$. When the identifier is *called*, $K\lfloor\tilde{a}\rfloor$, it is replaced by the process obtained from P by replacing the names $\tilde{x}$ by $\tilde{a}$. More precisely, a *substitution* $\sigma = \{\tilde{a}/\tilde{x}\}$ is a function that maps the names in $\tilde{x}$ to $\tilde{a}$, and is the identity for all the names not in $\tilde{x}$. The *application of substitution*, $P\sigma$, is defined in the standard way [SW01]. A π-Calculus process is either a call to an identifier, $K\lfloor\tilde{a}\rfloor$, a *choice process* deciding between prefixes, $\sum_{i\in I} \pi_i.P_i$, a *parallel composition* of processes, $P_1 \mid P_2$, or the *restriction* of a name in a process, $\nu a.P$:

$$P ::= K\lfloor\tilde{a}\rfloor \;\shortmid\; \textstyle\sum_{i\in I} \pi_i.P_i \;\shortmid\; P_1 \mid P_2 \;\shortmid\; \nu a.P.$$

The *set of all processes* is denoted by $\mathcal{P}$. We abbreviate empty sums (i.e., those with $I = \emptyset$) by $\mathbf{0}$ and use M or N to denote arbitrary sums. We also use the notation $\Pi_{i=1}^{n} P_i$ for iterated parallel composition. Processes that do not contain the parallel composition operator are called *sequential*. We denote sequential processes by $P_{\mathcal{S}}$, $Q_{\mathcal{S}}$ and the identifiers they use by $K_{\mathcal{S}}$. An identifier $K_{\mathcal{S}}$ is defined by $K_{\mathcal{S}}(\tilde{x}) := P_{\mathcal{S}}$ where $P_{\mathcal{S}}$ is a sequential process. W.l.o.g., we assume that every process either is $\mathbf{0}$ or does not contain $\mathbf{0}$. To see that this is no restriction consider the process $\overline{a}\langle b\rangle.\mathbf{0}$. We transform it to $\overline{a}\langle b\rangle.K\lfloor -\rfloor$ with $K(-) := \mathbf{0}$.

The input action $a(b)$ and the restriction $\nu c.P$ *bind* the names b and c, 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 α-conversion of bound names. Therefore, w.l.o.g., we assume that a name is bound at most once in a process and $bn\,(P) \cap fn\,(P) = \emptyset$. Moreover, if a substitution $\sigma = \{\tilde{a}/\tilde{x}\}$ is applied to a process P, we assume $bn\,(P) \cap (\tilde{a} \cup \tilde{x}) = \emptyset$.

We use the *structural congruence* relation in the definition of the behaviour of a process term. It is the smallest congruence where α-conversion of bound names is allowed, $+$ and $\mid$ are commutative and associative with $\mathbf{0}$ as the neutral element, and the following laws for restriction hold:

$$\nu x.\mathbf{0} \equiv \mathbf{0} \qquad \nu x.\nu y.P \equiv \nu y.\nu x.P \qquad \nu x.(P \mid Q) \equiv P \mid (\nu x.Q), \text{ if } x \notin fn\,(P).$$

The last rule is called *scope extrusion*. The behaviour of π-Calculus processes is then determined by the *reaction relation* $\rightarrow \subseteq \mathcal{P} \times \mathcal{P}$ defined by:

$$\text{(Par)}\ \frac{P \rightarrow P'}{P \mid Q \rightarrow P' \mid Q} \qquad \text{(Tau)}\ \tau.P + M \rightarrow P \qquad \text{(Res)}\ \frac{P \rightarrow P'}{\nu a.P \rightarrow \nu a.P'}$$

$$\text{(React)}\ (x(y).P + M) \mid (\overline{x}\langle z\rangle.Q + N) \rightarrow P\{z/y\} \mid Q$$

$$\text{(Const)}\ K\lfloor\tilde{a}\rfloor \rightarrow P\{\tilde{a}/\tilde{x}\}, \text{ if } K(\tilde{x}) := P$$

$$\text{(Struct)}\ \frac{P \rightarrow P'}{Q \rightarrow Q'}, \text{ if } P \equiv Q \text{ and } P' \equiv Q'.$$

By *Reach* (P) we denote the *set of all processes reachable from* P by the reaction relation. We use a client-server system to illustrate the behaviour of a π-Calculus process. It will serve us as a running example throughout the paper.

Example 1. Consider the process $C\lfloor url\rfloor \mid C\lfloor url\rfloor \mid S\lfloor url\rfloor$ modelling two clients and a sequential server, with the corresponding process identifiers defined as

$$C(url) := \nu ip.\overline{url}\langle ip\rangle.ip(s).s(x).C\lfloor url\rfloor$$
$$S(url) := url(y).\nu ses.\overline{y}\langle ses\rangle.\overline{ses}\langle ses\rangle.S\lfloor url\rfloor.$$

The server is located at some URL, $S\lfloor url\rfloor$. To contact it, a client sends its ip address on the channel url, $\overline{url}\langle ip\rangle$. This ip address is different for every client, therefore it is restricted. The server receives the IP address of the client and stores it in the variable y, $url(y)$. To establish a private connection with the client, the server creates a temporary session, νses, which it passes to the client, $\overline{y}\langle ses\rangle$. Note that by rule (React), y is replaced by ip during the system execution. Thus, the client receives this session, $ip(s)$. Client and server then continue to interact, which is not modelled explicitly. At some point, the server decides that the session should be ended. It sends the session object itself to the client, $\overline{ses}\langle ses\rangle$, and becomes a server again, $S\lfloor url\rfloor$. The client receives the message, $s(x)$, and calls its recursive definition to be able to contact the server once more, $C\lfloor url\rfloor$. The model can contain several clients (two in our case), but the server is engaged with one client at a time. $\diamondsuit$

Our theory employs a standard form of process terms, the so-called *restricted form* [Mey07]. It minimises the scopes of all restricted names νa not under a prefix π. Then processes congruent with $\mathbf{0}$ are removed. For example, the restricted form of $P = \nu a.\nu d.(\overline{a}\langle b\rangle.Q \mid \overline{b}\langle c\rangle.R)$ is $\nu a.\overline{a}\langle b\rangle.Q \mid \overline{b}\langle c\rangle.R$, but the restricted form of $\overline{a}\langle b\rangle.P$ is $\overline{a}\langle b\rangle.P$ itself. A *fragment* is a process of the form

$$F ::= K\lfloor\tilde{a}\rfloor \;\mid\; \textstyle\sum_{i\in I\neq\emptyset}\pi_i.P_i \;\mid\; \nu a.(F_1 \mid \ldots \mid F_n),$$

where $P_i \in \mathcal{P}$ and $a \in fn(F_i)$ for all i. We denote fragments by F or G. A process P_ν is in the *restricted form*, if it is a parallel composition of fragments, $P_\nu = \Pi_{i\in I}G_i$. The *set of fragments in* P_ν is denoted by $Frag(P_\nu) \overset{\text{df}}{=} \{G_i \mid i \in I\}$. The *set of all processes in restricted form* is denoted by $\mathcal{P}_\nu$.

For every process P, the function $(-)_\nu$ computes a structurally congruent process $(P)_\nu$ in the restricted form [Mey07]. For a choice composition and a call to a process identifier $(-)_\nu$ is defined to be the identity, and $(P \mid Q)_\nu \overset{\text{df}}{=} (P)_\nu \mid (Q)_\nu$. In the case of restriction, $\nu a.P$, we first compute the restricted form of P, which is a parallel composition of fragments, $(P)_\nu = \Pi_{i\in I}F_i$. We then restrict the scope of a to the fragments F_i where a is a free name (i.e., $i \in I_a \subseteq I$): $(\nu a.P)_\nu \overset{\text{df}}{=} \nu a.(\Pi_{i\in I_a}F_i) \mid \Pi_{i\in I\setminus I_a}F_i$.

Lemma 1. *For every process $P \in \mathcal{P}$ it holds $(P)_\nu \in \mathcal{P}_\nu$ and $P \equiv (P)_\nu$. For $P_\nu \in \mathcal{P}_\nu$ we have $(P_\nu)_\nu = P_\nu$.*

If we restrict structural congruence to processes in restricted form, we get the *restricted equivalence* relation $\hat{\equiv}$. It is the smallest equivalence on processes in restricted form that permits (1) associativity and commutativity of parallel composition and (2) replacing fragments by structurally congruent ones, i.e., $F \mid P_\nu \hat{\equiv} G \mid P_\nu$ if $F \equiv G$. It characterises structural congruence [Mey07]:

Lemma 2. *$P \equiv Q$ iff $(P)_\nu \hat{\equiv} (Q)_\nu$.*

Petri Nets A *net* is a triple $N \overset{\text{df}}{=} (P, T, W)$ such that P and T are disjoint sets of respectively *places* and *transitions*, and $W : (P \times T) \cup (T \times P) \to \mathbb{N} \overset{\text{df}}{=} \{0, 1, 2, \ldots\}$ is a *weight function*. A *marking* of N is a multiset M of places, i.e., $M : P \to \mathbb{N}$. The standard rules about drawing nets are adopted in this paper, viz. places are represented as circles, transitions as boxes, the weight function by arcs with numbers (the absence of an arc means that the corresponding weight is 0, and an arc with no number means that the corresponding weight is 1), and the marking is shown by placing tokens within circles. As usual, ${}^\bullet z \overset{\text{df}}{=} \{y \mid W(y, z) > 0\}$ and $z^\bullet \overset{\text{df}}{=} \{y \mid W(z, y) > 0\}$ denote the *pre-* and *postset* of $z \in P \cup T$, and ${}^\bullet Z \overset{\text{df}}{=} \bigcup_{z \in Z} {}^\bullet z$ and $Z^\bullet \overset{\text{df}}{=} \bigcup_{z \in Z} z^\bullet$, for all $Z \subseteq P \cup T$. In this paper, the presets of transitions are restricted to be non-empty, ${}^\bullet t \neq \emptyset$ for every $t \in T$. A *net system* is a pair $\Upsilon \overset{\text{df}}{=} (N, M_0)$ comprising a finite net N and an *initial* marking M_0.

A transition $t \in T$ is *enabled* at a marking M, denoted $M[t\rangle$, if $M(p) \geq W(p, t)$ for every $p \in {}^\bullet t$. Such a transition can be *fired*, leading to the marking M' with $M'(p) \overset{\text{df}}{=} M(p) - W(p, t) + W(t, p)$, for every $p \in P$. We denote this by $M[t\rangle M'$ or $M[\rangle M'$ if the identity of the transition is irrelevant. The set of *reachable* markings of Υ is the smallest (w.r.t. $\subseteq$) set $[M_0\rangle$ containing M_0 and such that if $M \in [M_0\rangle$ and $M[\rangle M'$ then $M' \in [M_0\rangle$.

A net system Υ is *k-bounded* if, for every reachable marking M and every place $p \in P$, $M(p) \leq k$, and *safe* if it is 1-bounded. Moreover, Υ is *bounded* if it is k-bounded for some $k \in \mathbb{N}$. One can show that the set $[M_0\rangle$ is finite iff Υ is bounded. W.l.o.g., we assume that for net systems known to be safe the range of the weight function is $\{0, 1\}$.

3 A Petri Net Translation of the π-Calculus

We recall the translation of π-Calculus processes into Petri nets defined in [Mey07]. The translation is based on the observation that processes are connected by restricted names they share. Consider the fragment $\nu a.(K\lfloor a \rfloor \mid L\lfloor a \rfloor)$. As the scope of a cannot be shrunk using the scope extrusion rule, the restricted name a 'connects' the processes $K\lfloor a \rfloor$ and $L\lfloor a \rfloor$. The idea of the translation is to have a separate place in the Petri net for each reachable 'bunch' of processes connected by restricted names, i.e., the notion of fragments plays a crucial role in the proposed translation. The algorithm takes a π-Calculus process P and computes a Petri net $\mathcal{PN}[\![P]\!]$ as follows:

- The places in the Petri net are all the fragments of every reachable process (more precisely, the congruence classes of fragments w.r.t. $\equiv$).
- The transitions consist of three disjoint subsets:
  - Transitions $t = ([F], [Q])$ model reactions inside a fragment F, where Q is such that $F \to Q$ and $[F]$ is a place (i.e., F is a reachable fragment). These reactions are communications of processes within F, τ actions, or calls to process identifiers, $K\lfloor \tilde{a} \rfloor$. There is an arc weighted one from place $[F]$ to t.

  - Transitions $t = ([F \mid F], [Q])$ model reactions between two structurally congruent reachable fragments along public channels, i.e., $F \mid F \to Q$ and $[F]$ is a place. There is an arc weighted two from $[F]$ to t. If this transition is fired, F contains a sequential process sending on a public channel and another one receiving on that channel, and there are two copies (up to $\equiv$) of F in the current process.
  - Transitions $t = ([F_1 \mid F_2], [Q])$ model reactions between reachable fragments $F_1 \not\equiv F_2$ along public channels: $F_1 \mid F_2 \to Q$ and $[F_1]$ and $[F_2]$ are places. There are two arcs each weighted one from $[F_1]$ and $[F_2]$ to t.

  The postsets of each kind of transitions are the reachable fragments in the restricted form of Q. If the fragment G occurs (up to $\equiv$) $k \in \mathbb{N}$ times in $(Q)_\nu$, then there is an arc weighted k from $([F], [Q])$ to $[G]$. For example, from the transition $\left(\left[\tau.\Pi_{i=1}^{3} K\lfloor a \rfloor\right], \left[\Pi_{i=1}^{3} K\lfloor a \rfloor\right]\right)$ there is an arc weighted three to the place $[K\lfloor a \rfloor]$.
- The initial marking of place $[F]$ in $\mathcal{PN}[\![P]\!]$ equals to the number of fragments in the restricted form of P that are congruent with F.

Note that if it is known in advance that the resulting Petri net will be safe, then no transition incident to an arc of weight more than one can fire, and so they can be dropped by the translation (in particular, the second kind of transitions will never appear). This fact can be used to optimise the translation of *safe processes* defined in Section 5.

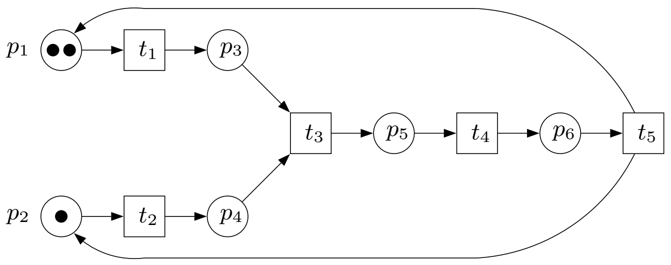
It turns out that a π-Calculus process and the corresponding Petri net obtained by this translation have isomorphic transition systems [Mey07]. Hence, one can verify properties specified for a process P using $\mathcal{PN}[\![P]\!]$. Returning to our running example, this translation yields the Petri net in Figure 1(a) for the process in Example 1.

Our translation is particularly suitable for verification because it represents an expressive class of processes (viz. FCPs) with potentially unbounded creation of restricted names as bounded Petri nets.

4 Boundedness of FCP Nets

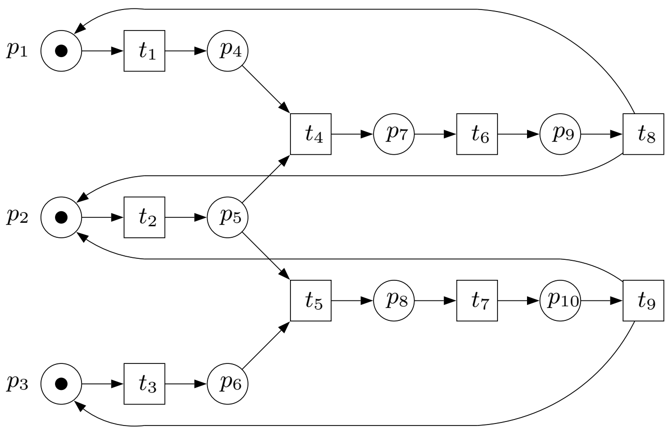
For general π-Calculus processes, the translation presented in the previous section may result in infinite Petri nets, and even when the result is finite, it can be unbounded, which is bad for model checking. (Model checking of even simplest properties of unbounded Petri nets is ExpSpace-hard.) To make verification feasible in practice, we need bounded nets, preferably even safe ones (the unfolding-based verification is especially efficient for safe nets), and so we have to choose an expressive subclass of π-Calculus which admits efficient verification.

In this section, we investigate the translation of the well-known *finite control processes (FCPs)*, a syntactic subclass of the π-Calculus [Dam96]. FCPs are parallel compositions of a finite number of sequential processes $P_{\mathcal{S}i}$, $P_{\mathcal{FC}} = \nu\tilde{a}.(P_{\mathcal{S}1} \mid \ldots \mid P_{\mathcal{S}n})$, and so new threads are never created and the degree of concurrency is bounded by n. The main result in this section states that the Petri net $\mathcal{PN}[\![P_{\mathcal{FC}}]\!]$ is bounded, and a non-trivial bound can be derived syntactically



$$
\begin{aligned}
p_1 &= [C\lfloor url \rfloor] \qquad p_2 = [S\lfloor url \rfloor]\\
p_3 &= \left[\nu ip.\overline{url}\langle ip\rangle.ip(s).s(x).C\lfloor url \rfloor\right]\\
p_4 &= [url(y).\nu ses.\overline{y}\langle ses\rangle.\overline{ses}\langle ses\rangle.S\lfloor url \rfloor]\\
p_5 &= \left[\nu ip.(ip(s).s(x).C\lfloor url \rfloor \mid \nu ses.\overline{ip}\langle ses\rangle.\overline{ses}\langle ses\rangle.S\lfloor url \rfloor)\right]\\
p_6 &= [\nu ses.(ses(x).C\lfloor url \rfloor \mid \overline{ses}\langle ses\rangle.S\lfloor url \rfloor)]
\end{aligned}
$$

(a)



$$
\begin{aligned}
p_1 &= \left[C^1\lfloor url \rfloor\right] \qquad p_2 = \left[S^3\lfloor url \rfloor\right] \qquad p_3 = \left[C^2\lfloor url \rfloor\right]\\
p_4 &= \left[\nu ip.\overline{url}\langle ip\rangle.ip(s).s(x).C^1\lfloor url \rfloor\right]\\
p_5 &= \left[url(y).\nu ses.\overline{y}\langle ses\rangle.\overline{ses}\langle ses\rangle.S^3\lfloor url \rfloor\right]\\
p_6 &= \left[\nu ip.\overline{url}\langle ip\rangle.ip(s).s(x).C^2\lfloor url \rfloor\right]\\
p_7 &= \left[\nu ip.(ip(s).s(x).C^1\lfloor url \rfloor \mid \nu ses.\overline{ip}\langle ses\rangle.\overline{ses}\langle ses\rangle.S^3\lfloor url \rfloor)\right]\\
p_8 &= \left[\nu ip.(ip(s).s(x).C^2\lfloor url \rfloor \mid \nu ses.\overline{ip}\langle ses\rangle.\overline{ses}\langle ses\rangle.S^3\lfloor url \rfloor)\right]\\
p_9 &= \left[\nu ses.(ses(x).C^1\lfloor url \rfloor \mid \overline{ses}\langle ses\rangle.S^3\lfloor url \rfloor)\right]\\
p_{10} &= \left[\nu ses.(ses(x).C^2\lfloor url \rfloor \mid \overline{ses}\langle ses\rangle.S^3\lfloor url \rfloor)\right]
\end{aligned}
$$

(b)

Fig. 1. The Petri nets corresponding to the FCP in Example 1 **(a)** and to the corresponding safe process in Example 3 **(b)**.

from the structure of $P_{\mathcal{FC}}$. The intuitive idea is that k tokens on a place $[F]$ require at least k processes $P_{\mathcal{S}i}$ in $P_{\mathcal{FC}} = \nu\tilde{a}.(P_{\mathcal{S}1} \mid \ldots \mid P_{\mathcal{S}n})$ that share some process identifiers.

To make the notion of sharing process identifiers precise we define *orbits*. The orbit of a sequential process $P_{\mathcal{S}i}$ consists of the identifiers $P_{\mathcal{S}i}$ calls (directly or indirectly). With this idea, we rephrase our result: if there are at most k orbits in $P_{\mathcal{FC}}$ whose intersection is non-empty then the net $\mathcal{PN}[\![P_{\mathcal{FC}}]\!]$ is k-bounded.

Generally, the result states that *the bound of* $\mathcal{PN}[\![P_{\mathcal{FC}}]\!]$ *is small.* If $P_{\mathcal{FC}} = \nu\tilde{a}.(P_{\mathcal{S}1} \mid \ldots \mid P_{\mathcal{S}n})$ then $\mathcal{PN}[\![P_{\mathcal{FC}}]\!]$ is trivially n-bounded, as the total number of orbits is n. Often, our method yields bounds which are better than n. This should be viewed in the light of the fact that for general bounded Petri nets the bound is double-exponential in the size of the net [Esp98]. This limits the state space in our translation and makes such nets relatively easy to model check.

The intuitive idea of the *orbit function* is to collect all process identifiers syntactically reachable from a given process. We employ the function $ident : \mathcal{P} \to \mathbb{P}(PIDS)$ which gives the set of process identifiers $ident(P)$ that are in the process $P \in \mathcal{P}$:

$$\begin{array}{ll} ident(K\lfloor\tilde{a}\rfloor) \overset{\text{df}}{=} \{K\} & ident(\sum_{i\in I}\pi_i.P_i) \overset{\text{df}}{=} \bigcup_{i\in I} ident(P_i) \\ ident(\nu a.P) \overset{\text{df}}{=} ident(P) & ident(P \mid Q) \overset{\text{df}}{=} ident(P) \cup ident(Q). \end{array}$$

The *orbit* of a process P, $orb(P)$, is the smallest (w.r.t. $\subseteq$) set such that $ident(P) \subseteq orb(P)$ and if a process identifier K with a defining equation $K(\tilde{x}) := Q$ is in $orb(P)$ then $ident(Q) \subseteq orb(P)$. The *maximal number of intersecting orbits of a process* $P_{\mathcal{FC}} = \nu\tilde{a}.(P_{\mathcal{S}1} \mid \ldots \mid P_{\mathcal{S}n})$ is

$$\#_\cap(P_{\mathcal{FC}}) \overset{\text{df}}{=} max\left\{|I| \mid I \subseteq \{1,\ldots,n\} \text{ and } \bigcap_{i\in I} orb(P_{\mathcal{S}i}) \neq \emptyset\right\}.$$

The main result of this section can now be stated as follows.

Theorem 1. *$\mathcal{PN}[\![P_{\mathcal{FC}}]\!]$ is $\#_\cap(P_{\mathcal{FC}})$-bounded.*

Example 2. Consider $P_{\mathcal{FC}} = C\lfloor url\rfloor \mid C\lfloor url\rfloor \mid S\lfloor url\rfloor$ in Example 1. We have $orb(S\lfloor url\rfloor) = \{S\}$ and $orb(C\lfloor url\rfloor) = \{C\}$ for both clients. Thus, $\#_\cap(P_{\mathcal{FC}}) = 2$, and so the corresponding Petri net $\mathcal{PN}[\![P_{\mathcal{FC}}]\!]$ in Figure 1(a) is 2-bounded. This is an improvement on the trivial bound of 3 (i.e., the number of concurrent processes in the system). $\Diamond$

We spend the rest of the section sketching the proof of this result. The Petri net $\mathcal{PN}[\![P_{\mathcal{FC}}]\!]$ is k-bounded iff in every reachable process $Q \in Reach(P_{\mathcal{FC}})$ there are at most k fragments that are structurally congruent. Thus, we need to show that the number of structurally congruent fragments is bounded by $\#_\cap(P_{\mathcal{FC}})$ in every reachable process Q. To do so, we assume there are k fragments $F_1 \equiv \ldots \equiv F_k$ in Q and conclude that there are at least k intersecting orbits in $P_{\mathcal{FC}}$, i.e., $\#_\cap(P_{\mathcal{FC}}) \geq k$. We argue as follows. From structural congruence we know that the identifiers in all F_i are equal. We now show that the identifiers of the F_i are already contained in the orbits of different $P_{\mathcal{S}i}$ in $P_{\mathcal{FC}}$. Thus, the intersection $orb(P_{\mathcal{S}1}) \cap \ldots \cap orb(P_{\mathcal{S}k})$ is not empty. This means that we have found k intersecting orbits, i.e., $\#_\cap(P_{\mathcal{FC}}) \geq k$.

To show $ident(F_i) \subseteq orb(P_{\mathcal{S}i})$ we need to relate the processes in every reachable fragment with the initial process $P_{\mathcal{FC}} = \nu\tilde{a}.(P_{\mathcal{S}1} \mid \ldots \mid P_{\mathcal{S}n})$. To achieve this, we prove that every reachable process is a parallel composition of subprocesses of $P_{\mathcal{S}i}$. These subprocesses are in the set of *derivatives* of $P_{\mathcal{S}i}$, which are defined by removing prefixes from $P_{\mathcal{S}i}$ as if those prefixes were consumed.

Definition 1. *The function $der : \mathcal{P} \to \mathbb{P}(\mathcal{P})$ assigns to every process P the set $der(P)$ as follows:*

$$der(\mathbf{0}) \overset{\text{df}}{=} \emptyset \qquad der(K\lfloor\tilde{a}\rfloor) \overset{\text{df}}{=} \{K\lfloor\tilde{a}\rfloor\}$$
$$der(\textstyle\sum_{i\in I\neq\emptyset} \pi_i.P_i) \overset{\text{df}}{=} \{\textstyle\sum_{i\in I\neq\emptyset} \pi_i.P_i\} \cup \textstyle\bigcup_{i\in I} der(P_i)$$
$$der(\nu a.P) \overset{\text{df}}{=} der(P) \qquad der(P \mid Q) \overset{\text{df}}{=} der(P) \cup der(Q).$$

Consider an FCP $P_{\mathcal{FC}} = \nu\tilde{a}.(P_{\mathcal{S}1} \mid \ldots \mid P_{\mathcal{S}n})$. We assign to every $P_{\mathcal{S}i}$ the set of derivatives, *$derivatives(P_{\mathcal{S}i})$. It is the smallest (w.r.t. $\subseteq$) set such that $der(P_{\mathcal{S}i}) \subseteq derivatives(P_{\mathcal{S}i})$ and if $K_{\mathcal{S}}\lfloor\tilde{a}\rfloor \in derivatives(P_{\mathcal{S}i})$ then $der(P_{\mathcal{S}}) \subseteq derivatives(P_{\mathcal{S}i})$, where $K_{\mathcal{S}}(\tilde{x}) := P_{\mathcal{S}}$.* $\diamondsuit$

Using structural congruence, every process reachable from $P_{\mathcal{FC}}$ can be rewritten as a parallel composition of derivatives of the processes in $P_{\mathcal{FC}}$. This technical lemma relates every reachable process with the processes in $P_{\mathcal{FC}}$.

Lemma 3. *Let $P_{\mathcal{FC}} = \nu\tilde{a}.(P_{\mathcal{S}1} \mid \ldots \mid P_{\mathcal{S}n})$. Then every $Q \in Reach(P_{\mathcal{FC}})$ is structurally congruent with $\nu\tilde{c}.(Q_1\sigma_1 \mid \ldots \mid Q_m\sigma_m)$ such that there is an injective function $inj : \{1,\ldots,m\} \to \{1,\ldots,n\}$ with $Q_i \in derivatives(P_{\mathcal{S}inj(i)})$ and $\sigma_i : fn(Q_i) \to \tilde{c} \cup fn(P_{\mathcal{FC}})$.*

For the derivatives Q of $P_{\mathcal{S}}$ it holds that the identifiers in Q are in the orbit of $P_{\mathcal{S}}$. Combined with the previous lemma, this relates the identifiers in a reachable fragment and the orbits in the initial process.

Lemma 4. *If $Q \in derivatives(P_{\mathcal{S}})$ then $ident(Q) \subseteq orb(P_{\mathcal{S}})$.*

By an induction along the structure of processes we show that for all $P \in \mathcal{P}$ the following holds: if $Q \in der(P)$ then $ident(Q) \subseteq ident(P)$. With this observation, Lemma 4 follows by an induction on the structure of $derivatives(P_{\mathcal{S}})$.

We return to the argumentation on Theorem 1. Consider a reachable process $Q \equiv \Pi^k F \mid Q'$ for some Q'. By Lemma 3, $Q \equiv \nu\tilde{c}.(Q_1\sigma_1 \mid \ldots \mid Q_m\sigma_m)$ with $Q_i \in derivatives(P_{\mathcal{S}inj(i)})$. By transitivity, $\Pi^k F \mid Q' \equiv \nu\tilde{c}.(Q_1\sigma_1 \mid \ldots \mid Q_m\sigma_m)$. By Lemmata 1 and 2, $\Pi^k F \mid (Q')_\nu \,\hat{\equiv}\, \Pi_{i\in I} G_i = (\nu\tilde{c}.(Q_1\sigma_1 \mid \ldots \mid Q_m\sigma_m))_\nu$ for some fragments G_i.

By definition of $\hat{\equiv}$, k of the G_is are structurally congruent. As identifiers are preserved by $\equiv$, these G_is have the same identifiers. Each G_i is a parallel composition of some $Q_i\sigma_i$s. With Lemma 4, $Q_i \in derivatives(P_{\mathcal{S}inj(i)})$ implies $ident(Q_i) \subseteq orb(P_{\mathcal{S}inj(i)})$. Since every G_i consists of different Q_is and inj is injective, we have k processes $P_{\mathcal{S}inj(i)}$ sharing identifiers, i.e., Theorem 1 holds.

In the case the orbits of all $P_{\mathcal{S}i}$ in $P_{\mathcal{FC}} = \nu\tilde{a}.(P_{\mathcal{S}1} \mid \ldots \mid P_{\mathcal{S}n})$ are pairwise disjoint, Theorem 1 implies the safeness of the Petri net $\mathcal{PN}[\![P_{\mathcal{FC}}]\!]$. In the following section we show that every FCP can be translated into a bisimilar process with disjoint orbits.

5 From FCPs to Safe Processes

Safe nets are a prerequisite to apply efficient unfolding-based verification techniques. According to Theorem 1, the reason for non-safeness of the nets of arbitrary FCPs is the intersection of orbits. In this section we investigate a translation of FCPs into their syntactic subclass called *safe processes,* where the sequential processes comprising an FCP have pairwise disjoint orbits. The idea of translating $P_{\mathcal{FC}} = \nu\tilde{a}.(P_{\mathcal{S}1} \mid \ldots \mid P_{\mathcal{S}n})$ to the safe process $\mathit{Safe}(P_{\mathcal{FC}})$ is to create copies of the process identifiers that are shared among several $P_{\mathcal{S}i}$, i.e., of those that belong to several orbits. (The corresponding defining equations are duplicated as well.) The intuition is that every $P_{\mathcal{S}i}$ gets its own set of process identifiers (together with the corresponding defining equations) which it can call during system execution. Hence, due to Theorem 1, safe processes are mapped to safe Petri nets.

The main result in this section states that the processes $P_{\mathcal{FC}}$ and $\mathit{Safe}(P_{\mathcal{FC}})$ are bisimilar, and, moreover, that the fragments are preserved in some sense. Furthermore, the size of the specification $\mathit{Safe}(P_{\mathcal{FC}})$ is at most quadratic in the size of $P_{\mathcal{FC}}$, and this translation is optimal.

Definition 2. *An FCP $P_{\mathcal{FC}} = \nu\tilde{a}.(P_{\mathcal{S}1} \mid \ldots \mid P_{\mathcal{S}n})$ is a* safe process *if the orbits of all $P_{\mathcal{S}i}$ are pairwise disjoint, i.e., for all $i, j \in \{1, \ldots, n\}$: if $i \neq j$ then $orb(P_{\mathcal{S}i}) \cap orb(P_{\mathcal{S}j}) = \emptyset$.* $\Diamond$

To translate an FCP $P_{\mathcal{FC}} = \nu\tilde{a}.(P_{\mathcal{S}1} \mid \ldots \mid P_{\mathcal{S}n})$ into a safe process $\mathit{Safe}(P_{\mathcal{FC}})$, we choose unique numbers for every sequential process, say i for $P_{\mathcal{S}i}$. We then rename every process identifier K in the orbit of $P_{\mathcal{S}i}$ to a fresh identifier K^i using the unique number i. We use the functions $ren_k : \mathcal{P} \to \mathcal{P}$, defined for every $k \in \mathbb{N}$ by

$$\begin{aligned} ren_k(K) &\overset{\text{df}}{=} K^k & ren_k(K\lfloor\tilde{a}\rfloor) &\overset{\text{df}}{=} ren_k(K)\lfloor\tilde{a}\rfloor \\ ren_k(\textstyle\sum_{i\in I} \pi_i.P_i) &\overset{\text{df}}{=} \textstyle\sum_{i\in I} \pi_i.ren_k(P_i) & ren_k(P \mid Q) &\overset{\text{df}}{=} ren_k(P) \mid ren_k(Q) \\ ren_k(\nu a.P) &\overset{\text{df}}{=} \nu a.ren_k(P). \end{aligned}$$

Employing the ren_k function, the FCP $P_{\mathcal{FC}} = \nu\tilde{a}.(P_{\mathcal{S}1} \mid \ldots \mid P_{\mathcal{S}n})$ is translated into a safe process as follows:

$$\mathit{Safe}(P_{\mathcal{FC}}) \overset{\text{df}}{=} \nu\tilde{a}.(ren_1(P_{\mathcal{S}1}) \mid \ldots \mid ren_n(P_{\mathcal{S}n})),$$

where the defining equation of $K_{\mathcal{S}}^k$ is $K_{\mathcal{S}}^k(\tilde{x}) := ren_k(P_{\mathcal{S}})$ if $K_{\mathcal{S}}(\tilde{x}) := P_{\mathcal{S}}$. The original defining equations $K_{\mathcal{S}}(\tilde{x}) := P_{\mathcal{S}}$ are then removed. We demonstrate this translation on our running example.

Example 3. Consider the FCP $P_{\mathcal{FC}} = C\lfloor url\rfloor \mid C\lfloor url\rfloor \mid S\lfloor url\rfloor$ in Example 1. The translation is $\mathit{Safe}(P_{\mathcal{FC}}) = C^1\lfloor url\rfloor \mid C^2\lfloor url\rfloor \mid S^3\lfloor url\rfloor$, where

$$C^i(url) := \nu ip.\overline{url}\langle ip\rangle.ip(s).s(x).C^i\lfloor url\rfloor, \quad i = 1, 2$$
$$S^3(url) := url(y).\nu ses.\overline{y}\langle ses\rangle.\overline{ses}\langle ses\rangle.S^3\lfloor url\rfloor.$$

The equations for C and S are removed. $\Diamond$

In the example, we just created another copy of the equation defining a client. In fact, the following result shows that the size of the translated system is at most quadratic in the size of the original specification. We measure the *size* of a π-Calculus process as the sum of the sizes of all the defining equations and the size of the main process. The size of a process is the number of prefixes, operators, and identifiers (with parameters) it uses. So the size of $\mathbf{0}$ is 1, the size of $K\lfloor\tilde{a}\rfloor$ is $1 + |\tilde{a}|$, the size of $\sum_{i\in I\neq\emptyset} \pi_i.P_i$ is $2|I| - 1 + \sum_{i\in I} \mathit{size}\,(P_i)$ (as there are $|I|$ prefixes and $|I| - 1$ pluses), the size of $P \mid Q$ is $1 + \mathit{size}\,(P) + \mathit{size}\,(Q)$, and the size of $\nu a.P$ is $1 + \mathit{size}\,(P)$.

Proposition 1 (Size). *Let $P_{\mathcal{FC}} = \nu\tilde{a}.(P_{\mathcal{S}1} \mid \ldots \mid P_{\mathcal{S}n})$ be an FCP. Then $\mathit{size}(\mathit{Safe}(P_{\mathcal{FC}})) \leq n \cdot \mathit{size}(P_{\mathcal{FC}})$.*

Note that since $n \leq \mathit{size}(P_{\mathcal{FC}})$, this result shows that the size of $\mathit{Safe}(P_{\mathcal{FC}})$ is at most quadratic in the size of $P_{\mathcal{FC}}$.

$\mathit{Safe}(P_{\mathcal{FC}})$ is a safe process; this follows from the compatibility of the renaming function with the function *orb*: $\mathit{orb}(\mathit{ren}_k(P)) = \mathit{ren}_k(\mathit{orb}(P))$.

Proposition 2 (Safeness). *Let $P_{\mathcal{FC}} = \nu\tilde{a}.(P_{\mathcal{S}1} \mid \ldots \mid P_{\mathcal{S}n})$ be an FCP. Then $\mathit{Safe}(P_{\mathcal{FC}})$ is a safe process.*

The translation of $P_{\mathcal{FC}}$ into $\mathit{Safe}(P_{\mathcal{FC}})$ does not alter the behaviour of the process: both processes are bisimilar with a meaningful bisimulation relation. This relation shows that the processes reachable from $P_{\mathcal{FC}}$ and $\mathit{Safe}(P_{\mathcal{FC}})$ coincide up to the renaming of process identifiers. Thus, not only the behaviour of $P_{\mathcal{FC}}$ is preserved by $\mathit{Safe}(P_{\mathcal{FC}})$, but also the structure of the reachable process terms, in particular their fragments. Technically, we define the relation $\mathcal{R}_i$ by $(P, Q) \in \mathcal{R}_i$ iff there are names $\tilde{a}$ and sequential processes $P_{\mathcal{S}1}, \ldots, P_{\mathcal{S}n}$, where the topmost operator of every $P_{\mathcal{S}i}$ is different from ν, such that

$$P \equiv \nu\tilde{a}.(P_{\mathcal{S}1} \mid \ldots \mid P_{\mathcal{S}n}) \text{ and } Q \equiv \nu\tilde{a}.(P_{\mathcal{S}1} \mid \ldots \mid \mathit{ren}_i(P_{\mathcal{S}i}) \mid \ldots \mid P_{\mathcal{S}n}).$$

Note that it is obvious from this definition that $\nu\tilde{a}.(P_{\mathcal{S}1} \mid \ldots \mid P_{\mathcal{S}n})$ and $\nu\tilde{a}.(P_{\mathcal{S}1} \mid \ldots \mid \mathit{ren}_i(P_{\mathcal{S}i}) \mid \ldots \mid P_{\mathcal{S}n})$ are related by $\mathcal{R}_i$.

Theorem 2. *For any $i \in \{1, \ldots, n\}$, the relation $\mathcal{R}_i$ is a bisimulation that relates the FCPs $\nu\tilde{a}.(P_{\mathcal{S}1} \mid \ldots \mid P_{\mathcal{S}n})$ and $\nu\tilde{a}.(P_{\mathcal{S}1} \mid \ldots \mid \mathit{ren}_i(P_{\mathcal{S}i}) \mid \ldots \mid P_{\mathcal{S}n})$.*

By transitivity of bisimulation, Theorem 2 allows for renaming several $P_{\mathcal{S}i}$ and still gaining a bisimilar process. In particular, renaming all n processes in $P_{\mathcal{FC}} = \nu\tilde{a}.(P_{\mathcal{S}1} \mid \ldots \mid P_{\mathcal{S}n})$ yields the result for the safe system $\mathit{Safe}(P_{\mathcal{FC}})$.

Consider a process Q which is reachable from $P_{\mathcal{FC}}$. We argue that the structure of Q is essentially preserved (1) by the translation of $P_{\mathcal{FC}}$ to the safe process $\mathit{Safe}(P_{\mathcal{FC}})$ and then (2) by the translation of $\mathit{Safe}(P_{\mathcal{FC}})$ to the safe Petri net $\mathcal{PN}[\![\mathit{Safe}(P_{\mathcal{FC}})]\!]$. With this result we can reason about the *structure of all processes reachable from* $P_{\mathcal{FC}}$ using $\mathcal{PN}[\![\mathit{Safe}(P_{\mathcal{FC}})]\!]$.

According to Theorem 2, $P_{\mathcal{FC}}$ and $\mathit{Safe}(P_{\mathcal{FC}})$ are bisimilar via the relation $\mathcal{R}_1 \circ \cdots \circ \mathcal{R}_n$, e.g., a process $Q{=}\nu a.\nu b.(H\lfloor a\rfloor \mid K\lfloor a\rfloor \mid L\lfloor b\rfloor)$ reachable from $P_{\mathcal{FC}}$ corresponds to $Q'{=}\nu a.\nu b.(H^1\lfloor a\rfloor \mid K^2\lfloor a\rfloor \mid L^3\lfloor b\rfloor)$ reachable from $\mathit{Safe}(P_{\mathcal{FC}})$. Hence, one can reconstruct the fragments of Q form those of Q'. Indeed, compute the restricted forms: $(Q)_\nu = \nu a.(H\lfloor a\rfloor \mid K\lfloor a\rfloor) \mid \nu b.L\lfloor b\rfloor$ and $(Q')_\nu = \nu a.(H^1\lfloor a\rfloor \mid K^2\lfloor a\rfloor) \mid \nu b.L^3\lfloor b\rfloor$. Dropping the superscripts in $(Q')_\nu$ yields the fragments in $(Q)_\nu$, since only the restricted names influence the restricted form of a process, not the process identifiers. The transition systems of $\mathit{Safe}(P_{\mathcal{FC}})$ and $\mathcal{PN}[\![\mathit{Safe}(P_{\mathcal{FC}})]\!]$ are isomorphic, e.g., Q' corresponds to the marking $M = \{[\nu a.(H^1\lfloor a\rfloor \mid K^2\lfloor a\rfloor)], [\nu b.L^3\lfloor b\rfloor]\}$ [Mey07, Theorem 1]. Thus, from a marking of $\mathcal{PN}[\![\mathit{Safe}(P_{\mathcal{FC}})]\!]$ one can obtain the restricted form of a reachable process in $\mathit{Safe}(P_{\mathcal{FC}})$, which in turn corresponds to the restricted form in $P_{\mathcal{FC}}$ (when the superscripts of process identifiers are dropped). Furthermore, the bisimulation between $P_{\mathcal{FC}}$ and $\mathcal{PN}[\![\mathit{Safe}(P_{\mathcal{FC}})]\!]$ allows one to reason about the *behaviour* of $P_{\mathcal{FC}}$ using $\mathcal{PN}[\![\mathit{Safe}(P_{\mathcal{FC}})]\!]$. (This bisimulation follows from the bisimulation between $P_{\mathcal{FC}}$ and $\mathit{Safe}(P_{\mathcal{FC}})$ and the isomorphism of the transition systems of $\mathit{Safe}(P_{\mathcal{FC}})$ and $\mathcal{PN}[\![\mathit{Safe}(P_{\mathcal{FC}})]\!]$).

We discuss our choice to rename all $P_{\mathcal{S}i}$ in $\nu\tilde{a}.(P_{\mathcal{S}1} \mid \ldots \mid P_{\mathcal{S}n})$ to gain a safe process. One might be tempted to improve our translation by renaming only a subset of processes $P_{\mathcal{S}i}$ whose orbits intersect with many others, in hope to get a smaller specification than $\mathit{Safe}(P_{\mathcal{FC}})$. We show that this idea does not work, and the resulting specification will be of the same size, i.e., our definition of $\mathit{Safe}(P_{\mathcal{FC}})$ is *optimal.* First, we illustrate this issue with an example.

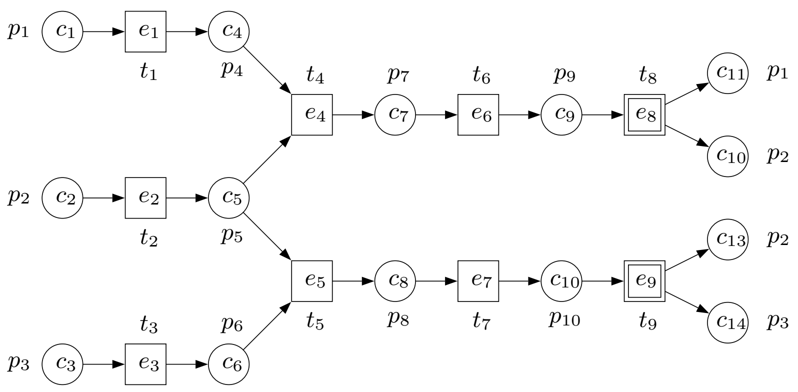
Example 4. Let $P = \tau.K\lfloor\tilde{a}\rfloor + \tau.L\lfloor\tilde{a}\rfloor$, $R = \tau.K\lfloor\tilde{a}\rfloor$ and $S = \tau.L\lfloor\tilde{a}\rfloor$, where $K(\tilde{x}) := \mathit{Def}_1$ and $L(\tilde{x}) := \mathit{Def}_2$. Consider the process $P \mid R \mid S$. The orbits of P and R as well as the orbits of P and S intersect.

The renaming of P yields $\mathit{ren}_1(P) \mid R \mid S = \tau.K^1\lfloor\tilde{a}\rfloor + \tau.L^1\lfloor\tilde{a}\rfloor \mid R \mid S$, where K and L are defined above and $K^1(\tilde{x}) := \mathit{ren}_1(\mathit{Def}_1), L^1(\tilde{x}) := \mathit{ren}_1(\mathit{Def}_2)$. This means we create additional copies of the shared identifiers K and L.

The renaming of R and S yields the process $P \mid \mathit{ren}_1(R) \mid \mathit{ren}_2(S) = P \mid \tau.K^1\lfloor\tilde{a}\rfloor \mid \tau.L^2\lfloor\tilde{a}\rfloor$, where we create new defining equations for the identifiers K^1 and L^2. The size of our translation is the same. $\Diamond$

This illustrates that any renaming of processes $P_{\mathcal{S}i}$ where the orbits overlap results in a specification of the same size. To render this intuition precisely, we call $K^k_{\mathcal{S}}(\tilde{x}) := \mathit{ren}_k(P_{\mathcal{S}})$ a *copy of the equation* $K_{\mathcal{S}}(\tilde{x}) := P_{\mathcal{S}}$, for any $k \in \mathbb{N}$. We also count $K_{\mathcal{S}}(\tilde{x}) := P_{\mathcal{S}}$ as a copy of itself.

Proposition 3 (Necessary condition for safeness). *The number of copies of an equation $K_{\mathcal{S}}(\tilde{x}) := P_{\mathcal{S}}$ necessary to get a safe process from $P_{\mathcal{FC}}$ equals to the number of orbits that contain $K_{\mathcal{S}}$.*



(a)

$$(\neg\mathsf{conf}_{e_4}\vee\mathsf{conf}_{e_1})\wedge(\neg\mathsf{conf}_{e_4}\vee\mathsf{conf}_{e_2})\wedge(\neg\mathsf{conf}_{e_5}\vee\mathsf{conf}_{e_2})\wedge(\neg\mathsf{conf}_{e_5}\vee\mathsf{conf}_{e_3})\wedge (\neg\mathsf{conf}_{e_6}\vee\mathsf{conf}_{e_4})\wedge(\neg\mathsf{conf}_{e_7}\vee\mathsf{conf}_{e_5})\wedge(\neg\mathsf{conf}_{e_4}\vee\neg\mathsf{conf}_{e_5})\ .$$

(b)

$$\mathsf{conf}_{e_1}\wedge\mathsf{conf}_{e_2}\wedge\mathsf{conf}_{e_3}\wedge(\neg\mathsf{conf}_{e_1}\vee\neg\mathsf{conf}_{e_2}\vee\mathsf{conf}_{e_4}\vee\mathsf{conf}_{e_5})\wedge(\neg\mathsf{conf}_{e_2}\vee\neg\mathsf{conf}_{e_3}\vee \mathsf{conf}_{e_4}\vee\mathsf{conf}_{e_5})\wedge(\neg\mathsf{conf}_{e_4}\vee\mathsf{conf}_{e_6})\wedge(\neg\mathsf{conf}_{e_5}\vee\mathsf{conf}_{e_7})\wedge\neg\mathsf{conf}_{e_6}\wedge\neg\mathsf{conf}_{e_7}\ .$$

(c)

Fig. 2. A finite and complete unfolding prefix of the Petri net in Figure 1(b) **(a)**, the corresponding configuration constraint $\mathcal{CONF}$ **(b)**, and the corresponding violation constraint $\mathcal{VIOL}$ expressing the deadlock condition **(c)**.

Now we show that our translation provides precisely this minimal number of copies of defining equations for every identifier, i.e., that it is optimal.

Proposition 4 (Optimality of our translation). *Our translation $Safe(P_{\mathcal{FC}})$ provides as many copies of a defining equation $K_{\mathcal{S}}(\tilde{x}) := P_{\mathcal{S}}$ as there are orbits containing $K_{\mathcal{S}}$.*

Remark 1. Note that one can, in general, optimise the translation by performing some dynamic (rather than syntactic) analysis, and produce a smaller process whose corresponding Petri net is safe; however, our notion of a safe process is syntactic rather than dynamic, and so the resulting process will not be safe according to our definition. $\Diamond$

6 Net Unfoldings

A *finite and complete unfolding prefix* of a bounded Petri net Υ is a finite acyclic net which implicitly represents all the reachable states of Υ together with transitions enabled at those states. Intuitively, it can be obtained through *unfolding* Υ, by successive firing of transitions, under the following assumptions: (i) for each new firing a fresh transition (called an *event*) is generated; (ii) for each newly produced token a fresh place (called a *condition*) is generated. For example, a finite and complete prefix of the Petri net in Figure 1(b) is shown in Figure 2(a). Due to its structural properties (such as acyclicity), the reachable states of Υ can be represented using *configurations* of its unfolding. A configuration C is a downward-closed set of events (being downward-closed means that if $e \in C$ and f is a causal predecessor of e then $f \in C$) without *choices* (i.e., for all distinct events $e, f \in C$, ${}^\bullet e \cap {}^\bullet f = \emptyset$). For example, in the prefix in Figure 2(a), $\{e_1, e_2, e_4\}$ is a configuration, whereas $\{e_1, e_2, e_6\}$ and $\{e_1, e_2, e_3, e_4, e_5\}$ are not (the former does not include e_4, which is a predecessor of e_6, while the latter contains a choice between e_4 and e_5). Intuitively, a configuration is a partially ordered execution, i.e., an execution where the order of firing some of its events (viz. concurrent ones) is not important; e.g., the configuration $\{e_1, e_2, e_4\}$ corresponds to two totally ordered executions reaching the same *final* marking: $e_1e_2e_4$ and $e_2e_1e_4$. Since a configuration can correspond to multiple executions, it is often much more efficient in model checking to explore configurations rather than executions. We will denote by $[e]$ the *local* configuration of an event e, i.e., the smallest (w.r.t. $\subseteq$) configuration containing e (it is comprised of e and its causal predecessors).

The unfolding is infinite whenever the original Υ has an infinite run; however, since Υ is bounded and hence has only finitely many reachable states, the unfolding eventually starts to repeat itself and can be truncated (by identifying a set of *cut-off* events) without loss of information, yielding a finite and complete prefix. Intuitively, an event e can be declared cut-off if the already built part of the prefix contains a configuration C^e (called the *corresponding* configuration of e) such that its final marking coincides with that of $[e]$ and C^e is smaller than $[e]$ w.r.t. some well-founded partial order on the configurations of the unfolding, called an *adequate order* [ERV02].

Efficient algorithms exist for building such prefixes [ERV02,Kho03], which ensure that the number of non-cut-off events in a complete prefix never exceeds the number of reachable states of the original Petri net. Moreover, complete prefixes are often exponentially smaller than the corresponding state graphs, especially for highly concurrent Petri nets, because they represent concurrency directly rather than by multidimensional interleaving 'diamonds' as it is done in state graphs. For example, if the original Petri net consists of 100 transitions which can fire once in parallel, the state graph will be a 100-dimensional hypercube with 2^{100} vertices, whereas the complete prefix will coincide with the net itself. Also, if the example in Figure 1(b) is scaled up (by increasing the number of clients), the size of the prefix is linear in the number of clients, even though the number of reachable states grows exponentially. Thus, unfolding prefixes significantly alleviate the state explosion in many practical cases.

A fundamental property of a finite and complete prefix is that each reachable marking of Υ is a final marking of some configuration C (without cut-offs) of the prefix, and, conversely, the final marking of each configuration C of the prefix is a reachable marking in Υ. Thus various reachability properties of Υ (e.g., marking and sub-marking reachability, fireability of a transition, mutual exclusion of a set of places, deadlock, and many others) can be restated as the corresponding properties of the prefix, and then checked, often much more efficiently.

Most of 'interesting' computation problems for safe Petri nets are PSPACE-complete [Esp98], but the same problems for prefixes are often in NP or even P. (Though the size of a finite and complete unfolding prefix can be exponential in the size of the original Petri net, in practice it is often relatively small, as explained above.) A reachability property of Υ can easily be reformulated for a prefix, and then translated into some canonical problem, e.g., Boolean satisfiability (SAT). Then an off-the-shelf solver can be used for efficiently solving it. Such a combination 'unfolder & solver' turns out to be quite powerful in practice [KKY04].

Unfolding-Based Model Checking This paper concentrates on the following approach to model checking. First, a finite and complete prefix of the Petri net unfolding is built. It is then used for constructing a Boolean formula encoding the model checking problem at hand. (It is assumed that the property being checked is the unreachability of some 'bad' states, e.g., deadlocks.) This formula is unsatisfiable iff the property holds, and such that any satisfying assignment to its variables yields a trace violating the property being checked.

Typically such a formula would have for each non-cut-off event e of the prefix a variable conf_e (the formula might also contain other variables). For every satisfying assignment A, the set of events $C \overset{\mathrm{df}}{=} \{e \mid \mathsf{conf}_e = 1\}$ is a configuration whose final marking violates the property being checked. The formula often has the form $\mathcal{CONF} \wedge \mathcal{VIOL}$. The role of the property-independent *configuration constraint* $\mathcal{CONF}$ is to ensure that C is a configuration of the prefix (not just an arbitrary set of events). $\mathcal{CONF}$ can be defined as the conjunction of the formulae

$$\bigwedge_{e \in E \setminus E_{cut}} \bigwedge_{f \in {}^{\bullet\bullet}e} (\neg\mathsf{conf}_e \vee \mathsf{conf}_f) \quad \text{and} \quad \bigwedge_{e \in E \setminus E_{cut}} \bigwedge_{f \in Ch_e} (\neg\mathsf{conf}_e \vee \neg\mathsf{conf}_f)\ ,$$

where $Ch_e \overset{\mathrm{df}}{=} \{(({}^{\bullet}e)^{\bullet} \setminus \{e\}) \setminus E_{cut}\}$ is the set of non-cut-off events which are in the direct choice relation with e. The former formula is basically a set of implications ensuring that if $e \in C$ then its immediate predecessors are also in C, i.e., C is downward closed. The latter one ensures that C contains no choices. $\mathcal{CONF}$ is given in the *conjunctive normal form (CNF)* as required by most SAT solvers. For example, the configuration constraint for the prefix in Figure 2(a) is shown in part (b) of this figure. The size of this formula is quadratic in the size of the prefix, but can be reduced down to linear by introducing auxiliary variables.

The role of the property-dependent *violation constraint* $\mathcal{VIOL}$ is to express the property violation condition for a configuration C, so that if a configuration C satisfying this constraint is found then the property does not hold, and C can be translated into a violation trace. For example, for the deadlock condition $\mathcal{VIOL}$

can be defined as

$$\bigwedge_{e\in E}\Big(\bigvee_{f\in{}^{\bullet\bullet}e}\neg\mathsf{conf}_f \vee \bigvee_{f\in({}^{\bullet}e)^{\bullet}\setminus E_{cut}}\mathsf{conf}_f\Big)\ .$$

This formula requires for each event e (including cut-off events) that some event in ${}^{\bullet\bullet}e$ has not fired or some of the non-cut-off events (including e unless it is cut-off) consuming tokens from ${}^{\bullet}e$ has fired, and thus e is not enabled. This formula is given in the CNF. For example, the violation constraint for the deadlock checking problem formulated for the prefix in Figure 2(a) is shown in part (c) of this figure. The size of this formula is linear in the size of the prefix.

If $\mathcal{VIOL}$ is a formula of polynomial size (in the size of the prefix) then one can check $\mathcal{CONF}\wedge\mathcal{VIOL}$ for satisfiability in non-deterministic polynomial time. In particular, every polynomial size (w.r.t. the prefix) formula F over the places of the net can be translated into a $\mathcal{VIOL}$ formula that is polynomial in the size of the prefix. Here, an atomic proposition p of F holds iff place p carries a token (we deal with safe nets). This covers reachability of markings and submarkings, deadlocks, mutual exclusion, and many other properties. Furthermore, an unfolding technique for model checking state-based LTL-X is presented in [EH01]. State-based means that the atomic propositions in the logic are again the places of the Petri net.

7 Experimental Results

To demonstrate the practicality of our approach, we implemented the translation of π-Calculus to Petri nets discussed in Section 3 and the translation of FCPs to safe processes presented in Section 5. In this section, we apply our tool chain to check three series of benchmarks for deadlocks. We compare the results with other well-known approaches and tools for π-Calculus verification.

The *NESS (Newcastle E-Learning Support System)* example models an electronic coursework submission system. This series of benchmarks is taken from [KKN06], where the only other unfolding-based verification technique for the π-Calculus is presented. The approach described in [KKN06] is limited to the *finite* π-Calculus, a subset of π-Calculus allowing to express only finite behaviours. It translates finite π-Calculus terms into high-level Petri nets and model checks the latter. The translation into Petri nets used in [KKN06] is very different from our approach, and a high-level net unfolder is used there for verification, while our technique uses the standard unfolding procedure for safe low-level nets. Moreover, our technique is not limited to the finite π-Calculus.

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:

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

The idea is that the students are supposed to submit their work for assessment to *NESS*. The teacher passes the channel *nessc* of the system to all students,

	FCP	**HLNet**		**Model Checking**				**mwb**	**hal**	**Struct**			**Safe**	**Struct**		**Model Checking**			
Mod.	**Size**	**|P|**	**|T|**	**unf**	**|B|**	**|E*|**	**sat**	**dl**	**π2fc**	**|P|**	**|T|**	**B**	**Size**	**|P|**	**|T|**	**unf**	**|B|**	**|E*|**	**sat**
dns4	84	1433	511	6	10429	181	< 1	10	93	22	47	8	98	32	50	< 1	113	38	< 1
dns6	123	3083	1257	46	28166	342	< 1	-	-	32	94	12	145	48	99	< 1	632	159	< 1
dns8	162	5357	2475	354	58863	551	< 1	-	-	42	157	16	192	64	164	< 1	3763	745	< 1
dns10	201	8255	4273	-				-	-	52	236	20	271	80	239	1	22202	3656	2
dns12	240	11777	6791	-				-	-	62	331	24	324	96	286	56	128295	18192	62
ns2	61	157	200	1	5553	127	< 1	< 1	< 1	18	28	4	67	26	40	< 1	61	27	< 1
ns3	88	319	415	7	22222	366	< 1	1	8	37	91	6	98	56	141	< 1	446	153	< 1
ns4	115	537	724	69	101005	1299	1	577	382	68	229	8	129	102	364	< 1	5480	1656	< 1
ns5	142	811	1139	532	388818	4078	58	-	-	119	511	10	160	172	815	17	36865	7832	3
ns6	169	1141	1672	-				-	-	206	1087	12	191	282	1722	1518	377920	65008	84
ns7	196	1527	2335	-				-	-	361	2297	14	222	646	3605	-			
ns2-r	61							n/a	n/a	16	24	4	67	24	36	< 1	51	22	< 1
ns3-r	88							n/a	n/a	29	70	6	98	48	117	< 1	292	99	< 1
ns4-r	113							n/a	n/a	45	123	8	127	79	216	< 1	1257	392	< 1
ns5-r	140							n/a	n/a	66	241	10	158	119	435	2	10890	2635	1
ns6-r	167							n/a	n/a	91	418	12	189	167	768	123	107507	19892	31
ns7-r	194							n/a	n/a	120	666	14	220	223	1239	-			

Table 1. Experimental results I.

$\overline{h_i}\langle nessc\rangle$, and then waits for the confirmation that they have finished working on the assignment, $h_i(x)$. After receiving the ness channel, $h_i(nsc)$, students 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 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:

$$
\begin{aligned}
T(nessc, h_1, \ldots, h_k) :=& \Pi_{i=1}^{k} \overline{h_i}\langle nessc\rangle.h_i(x_i).\mathbf{0} \\
S(h, h_i) :=& h_i(nsc).(\overline{h}\langle h_i\rangle.\overline{h_i}\langle fin\rangle.\mathbf{0} + h(x).\overline{nsc}\langle h_i\rangle.\overline{nsc}\langle x\rangle.\overline{h_i}\langle fin\rangle.\mathbf{0}) \\
ENV(nessc) :=& nessc(y_1).\ \ldots\ .nessc(y_k).\mathbf{0}
\end{aligned}
$$

In the following Table 1, the row **ns**k gives the verification results for the *NESS* system with $k \in \mathbb{N}$ students. The property we verified was whether all processes successfully terminate by reaching the end of their individual code (as distinguished from a deadlock where some processes are stuck in the middle of their intended behaviour, waiting for a communication to occur). Obviously, the system successfully terminates iff the number of students is even, i.e., they can be organised into pairs. The **dns**k entries refer to a refined *NESS* model where the pairing of students is deterministic; thus the number of students is even, and these benchmarks are deadlock-free.

The second example is the client-server system similar to our running example. For a more realistic model, we extend the server to spawn separate sessions that handle the clients' requests. We change the server process in Section 2 to a more concurrent *CONCS* and add separate session processes:

$$
\begin{aligned}
CONCS(url, getses) &:= url(y).getses(s).\overline{y}\langle s\rangle.CONCS\lfloor url, getses\rfloor \\
SES(getses) &:= \nu ses.\overline{getses}\langle ses\rangle.\overline{ses}\langle ses\rangle.SES\lfloor getses\rfloor
\end{aligned}
$$

On a client's request, the server creates a new session object using the *getses* channel, $getses(s)$. A session object is modelled by a *SES* process. It sends its

	FCP	mwb	hal	Struct			Safe	Struct		Model Checking			
Model	Size	dl	π2fc	|P|	|T|	B	Size	|P|	|T|	unf	|B|	|E*|	sat
gsm	194	-	18	374	138	1	194	148	344	< 1	345	147	< 1
gsm-r	194	n/a	n/a	60	72	1	194	75	110	< 1	150	72	< 1
1s1c	44	-	< 1	11	13	1	44	12	15	< 1	17	9	< 1
1s2c	47	-	6	12	15	2	58	22	30	< 1	35	17	< 1
2s1c	47	-	2	20	31	2	56	22	35	< 1	37	18	< 1
2s2c	50	-	138	31	59	2	70	40	66	< 1	73	33	< 1
3s2c	53	-	-	68	159	3	82	66	128	< 1	137	57	< 1
3s3c	56	-	-	85	217	3	96	100	194	< 1	216	87	< 1
4s4c	63	-	-	362	1202	4	122	216	484	< 1	537	195	< 1
5s5c	68	-	-	980	3818	5	148	434	1132	< 1	1238	403	< 1

Table 2. Experimental results II.

private channel νses along the *getses* 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. This case study uses recursion and is scalable in the number of clients and the number of sessions. In Table 2, e.g., the entry 5s5c gives the verification results for the system with five *SES* processes, five C processes and one server. All these benchmarks are deadlock-free.

The last example is the well-known specification of the handover procedure in the GSM Public Land Mobile Network. We use the standard π-Calculus model with one mobile station, two base stations, and one mobile switching center presented in [OP92].

We compare our results with three other techniques for π-Calculus verification: the mentioned approach in [KKN06], the verification kit *HAL* [FGMP03], and the *mobility workbench* (*MWB*) [VM94]. *HAL* translates a π-Calculus process into a history dependent automaton (HDA) [Pis99]. This in turn is translated into a finite automaton which is checked using standard tools. The *MWB* does not use any automata translation, but builds the state space on the fly. These tools can verify various properties, but we perform our experiments for deadlock checking only, as it is the common denominator of all these tools.

We briefly comment on the role of the models with the suffix $-r$ in Table 1. One can observe that parallel compositions inside a fragment lead to interleaving 'diamonds' in our Petri net representation. Thus, restricted names that are known to a large number of processes can make the size of our Petri net translation grow exponentially. We demonstrate this effect by verifying some of the *NESS* benchmarks with and without (suffix $-r$ in the table) the restrictions on such critical names. Even with the critical restrictions our approach outperforms the other tools; but when such restrictions are removed, it becomes orders of magnitude faster. (Removing such critical restrictions does not alter the process behaviour: $\nu a.P$ can evolve into $\nu a.P'$ iff P can evolve into P'; thus, one can replace $\nu a.P$ by P for model checking purposes.)

The columns in Tables 1 and 2 are organised as follows. `FCP Size` gives the size of the process as defined in Section 5. The following two columns, `HLNet` and `Model Checking` (present only in Table 1), are the verification results when the approach in [KKN06] is applied. In the former column, $|P|$ and $|T|$ state the number of places and transitions in the high-level Petri net. The following column `unf` gives the time to compute the unfolding prefix of this net. (We measure all runtimes in seconds.) For this prefix, $|B|$ is the number of conditions, and $|E^*|$ is the number of events (excluding cut-offs). Like our technique, the [KKN06] employs a SAT solver whose runtime is given by `sat`. The following two columns, `mwb dl` and `hal` π`2fc`, give the runtimes for the deadlock checking algorithm in *MWB* and for converting a π-Calculus process into a finite automaton (via HDA). This time includes the translation of a π-Calculus process into an HDA, minimisation of this HDA, and the conversion of the minimised HDA into a finite automaton [FGMP03]. The remaining entries are the results of applying our model checking procedure. The column `Struct` gives the numbers of places and transitions and the bounds of the Petri nets corresponding to a direct translation of the FCPs. These nets are given only for comparison, and are not used for model checking. `Safe Size` gives the size of the safe process computed by the function *Safe* in Section 5, and the next column gives the numbers of places and transitions of the corresponding safe Petri nets. Note that these nets, unlike those in [KKN06], are the usual low-level Petri nets. The following columns give the unfolding times, the prefix sizes, and the times for checking deadlocks on the prefixes using a SAT solver. A '–' in the tables indicates the corresponding tool did not produce an output within 30 minutes, and an 'n/a' means the technique was not applicable to the example.

Table 1 illustrates the results for checking the *NESS* example with the different techniques. As the *MWB* requires processes where all names are restricted, we cannot check the $-r$ versions of the case studies. Our runtimes are by orders of magnitude smaller in comparison with *HAL* and *MWB*, and are much better compared with the approach in [KKN06]. Furthermore, they dramatically improve when the critical names are removed (the $-r$ models).

The approach in [KKN06] only applies to the finite π-Calculus, so one cannot check the client-server or the GSM benchmarks with that technique. Table 2 shows that the proposed technique dramatically outperforms both *MWB* and *HAL*, and handles the benchmark with five sessions and clients within a second.

8 Conclusions and Future Work

In this paper, we have proposed a practical approach for verification of finite control processes. It works by first translating the given FCP into a safe process, and then translating the latter into a safe Petri net for which unfolding-based model checking is performed. Our translation to safe processes exploits a general boundedness result for FCP nets based on the theory of orbits. Our experiments show that this approach has significant advantages over other existing tools for verification of mobile systems in terms of memory consumption and runtime. We plan to further develop this approach, and below we identify potential directions for future research.

It would be useful to extend some temporal logic so that it could express interesting properties of π-Calculus. (The usual LTL does not capture, at least in a natural way, the communication in dynamically created channels and dynamic connectivity properties.) Due to our fragment-preserving (i.e., preserving the local connections of processes) bisimulation result, one should be able to translate such properties into Petri net properties for verification. The hope is that since this Petri net has a rich structure (e.g., the connectivity information can be retrieved from place annotations), the resulting properties can be expressed in some standard logic such as state-based LTL and then efficiently model checked with existing techniques.

One can observe that after the translation into a safe process, some fragments differ only by the replicated process identifiers. Such fragments are equivalent in the sense that they react in the same way and generate equivalent places in the postsets of the transitions. Hence, it should be possible to optimise our translation procedure, because many structural congruence checks can be omitted and several computations of enabled reactions become unnecessary. Moreover, this observation allows one to use in the unfolding procedure a weaker (compared with the equality of final markings) equivalence on configurations, as explained in [Kho03, Section 2.5]. This would produce cut-off events more often and hence reduce the size of the unfolding prefix.

It seems to be possible to generalise our translation to a wider subclass of π-Calculus. For example, consider the process $S\lfloor url \rfloor \mid C\lfloor url \rfloor \mid C\lfloor url \rfloor$ modelling a *concurrent* server and two clients, with the corresponding process identifiers defined as

$$\begin{aligned} S(url) &:= url(y).(\nu ses.\overline{y}\langle ses\rangle.\overline{ses}\langle ses\rangle.\mathbf{0} \mid S\lfloor url \rfloor) \\ C(url) &:= \nu ip.\overline{url}\langle ip\rangle.ip(s).s(x).C\lfloor url \rfloor \end{aligned}$$

Intuitively, when contacted by a client, the server spawns a new session and is ready to serve another client, i.e., several clients can be served in parallel. Though this specification is not an FCP, it still results in a 2-bounded Petri net very similar to the one in Figure 1(a). Our method can still be used to convert it into a safe Petri net for subsequent verification.

Acknowledgements The authors would like to thank Maciej Koutny, Eike Best and Ernst-Rüdiger Olderog for helpful comments and for arranging the visit of the first author to Newcastle.

This research was supported by the German Research Council (DFG) as part of the Graduate School GRK 1076/1 (TrustSoft) and by the Royal Academy of Engineering/Epsrc post-doctoral research fellowship EP/C53400X/1 (Davac).

References

[CGP99] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 1999.

[Dam96] M. Dam. Model checking mobile processes. *Information and Computation*, 129(1):35–51, 1996.

[EH01] J. Esparza and K. Heljanko. Implementing LTL model checking with net unfoldings. In M.B. Dwyer, editor, *Proc. SPIN'01*, volume 2057 of *Lecture Notes in Computer Science*, pages 37–56. Springer-Verlag, 2001.

[ERV02] J. Esparza, S. Römer, and W. Vogler. An Improvement of McMillan's Unfolding Algorithm. *Formal Methods in System Design*, 20(3):285–310, 2002.

[Esp98] J. Esparza. Decidability and complexity of Petri net problems — an introduction. In W. Reisig and G. Rozenberg, editors, *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*, pages 374–428. Springer-Verlag, 1998.

[FGMP03] G.-L. Ferrari, S. Gnesi, U. Montanari, and M. Pistore. A model-checking verification environment for mobile processes. *ACM Transactions on Software Engineering and Methodology*, 12(4):440–473, 2003.

[Kho03] V. Khomenko. *Model Checking Based on Prefixes of Petri Net Unfoldings.* PhD thesis, School of Computing Science, Newcastle University, 2003.

[KKN06] V. Khomenko, M. Koutny, and A. Niaouris. Applying Petri net unfoldings for verification of mobile systems. In *Proc. Workshop on Modelling of Objects, Components and Agents (MOCA'06)*, Bericht FBI-HH-B-267/06, pages 161–178. University of Hamburg, 2006.

[KKY04] V. Khomenko, M. Koutny, and A. Yakovlev. Detecting state coding conflicts in STG unfoldings using SAT. *Fundamenta Informaticae*, 62(2):1–21, 2004.

[McM92] K. McMillan. Using unfoldings to avoid state explosion problem in the verification of asynchronous circuits. In *Proc. CAV'92*, volume 663 of *Lecture Notes in Computer Science*, pages 164–174. Springer-Verlag, 1992.

[Mey07] R. Meyer. A theory of structural stationarity in the π-Calculus. *Under revision*, 2007.

[Mil99] R. Milner. *Communicating and Mobile Systems: the π-Calculus.* Cambridge University Press, 1999.

[MKS08] R. Meyer, V. Khomenko, and T. Strazny. A practical approach to verification of mobile systems using net unfoldings. Technical Report CS-TR-1064, School of Computing Science, Newcastle University, 2008. URL: `http://www.cs.ncl.ac.uk/research/pubs/trs/abstract.php?number=1064`.

[OP92] F. Orava and J. Parrow. An algebraic verification of a mobile network. *Formal Aspects of Computing*, 4(6):497–543, 1992.

[Pis99] M. Pistore. *History Dependent Automata.* PhD thesis, Dipartimento di Informatica, Università di Pisa, 1999.

[SW01] D. Sangiorgi and D. Walker. *The π-Calculus: a Theory of Mobile Processes.* Cambridge University Press, 2001.

[Val98] A. Valmari. *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*, chapter The State Explosion Problem, pages 429–528. Springer-Verlag, 1998.

[VM94] B. Victor and F. Moller. The mobility workbench: A tool for the π-Calculus. In *Proc. CAV'94*, volume 818 of *Lecture Notes in Computer Science*, pages 428–440. Springer-Verlag, 1994.