Applying Petri Net Unfoldings for Verification of Mobile Systems

Victor Khomenko, Maciej Koutny and Apostolos Niaouris

School of Computing Science, University of Newcastle
Newcastle upon Tyne, NE1 7RU, U.K.
`{victor.khomenko,maciej.koutny,apostolos.niaouris}@ncl.ac.uk`

Summary. Mobility is a central feature of many distributed systems of ever growing complexity. To make their formal analysis and verification feasible, process algebras — notably the π-calculus — have been introduced and extensively studied.

A well-established method of verifying the correctness of general distributed systems has been model-checking which is completely automatic and relatively fast compared to other alternatives, and so particularly attractive in industrial context. Mobile systems are highly concurrent causing state space explosion when applying model-checking techniques. To cope with this problem, techniques based on partial order semantics of concurrency seem to offer the desired level of efficiency. The aim of this paper is to investigate how one of such techniques — based of unfoldings of high-level Petri nets — could be used for verification of π-calculus terms.

Our starting point was an existing compositional translation from a finite fragment of the π-calculus into a class of high-level Petri nets. We developed a prototype tool based on this theoretical translation and an existing efficient unfolder and verifier. In this paper, we describe initial experimental results in support of specific design choices. Crucially, developing the prototype was not a straightforward task since the theoretical translation does not produce nets which conform to the input format required by the verifier. The paper states how this mismatch has been overcome and draws conclusions for future uses of unfoldings technique in the model-checking of mobile systems.

Keywords: mobile systems, π-calculus, model-checking, Petri net unfoldings, high-level Petri nets.

1 Introduction

Mobility has now become a central feature of many real life concurrent and distributed computing systems. To model it and reason about its properties, process algebras such as π-calculus [14, 15] have been introduced and studied. It allows, in particular, to express a dynamic change in the process ability to communicate with the external environment, by passing *references* (or a *channels*) through interactions on previously known channels. It is widely accepted as a *fundamental* formal model within which mobility can be succinctly expressed and reasoned about.

The complexity of verification of concurrent and distributed systems is widely recognised as a major stumbling block in this key area of computer system design. One way of coping with the complexity problem is to use formal methods supported by computer aided verification tools. Within this approach, a well-established method is model-checking [6] which is completely automatic and relatively fast compared to other alternatives. It is therefore particularly attractive in industrial context as it can contribute successfully to the reduction of product development costs [16].

Model-checking is a technique in which the verification of a system is carried out using a finite representation of its state space. Basic properties, such as the absence of a deadlock or satisfaction of a state invariant (e.g., mutual exclusion), can be verified by inspecting individual states. An important pragmatic feature of model-checking algorithms is that they produce counterexamples which can be used for debugging [16]. Industrial strength model-checkers are beginning to have an impact on practical designs and design methodologies. For example, in a "classical" reactive system application to the call processing software of a telephone switch at Bell Labs, model extraction combined with model-checking revealed ten times as many concurrency related defects in the target code as the conventional system testing did [9]. Such an approach is particularly effective in detecting inter-process communication problems at an early stage of system design, helping to resolve the issue of design productivity.

Model-checking of concurrent systems is intrinsically hard, and exhibits a trade-off between the compactness of the representation of the system and resources it takes to verify behavioural properties. For example, the classical deadlock detection problem is PSPACE-complete for a compact (bounded) Petri net or equivalent process algebra representation, but polynomial for transition system representation. However, the latter is often exponentially larger, and soon becomes too large to be stored in the main memory. That is, even a relatively small system specification can (and often does) yield a very large state space, and this phenomenon is called the *state space explosion* problem [17].

The problem of model-checking mobile systems is that they are highly concurrent causing a state space explosion when applying model-checking techniques. One should therefore use approach which alleviates this problem, in our case, based on partial order semantics of concurrency and the corresponding Petri net unfoldings [13]. A finite and complete unfolding prefix of a Petri net is a finite acyclic net which implicitly represents all the reachable states of the original net together with transitions enabled at those states. Efficient algorithms exist for building such prefixes [10], and complete prefixes are often exponentially smaller than the corresponding state graphs, especially for highly concurrent systems, because they represent concurrency directly rather than by multidimensional "diamonds" as it is done in state graphs.

1.1 About this paper

The aim of this paper is to investigate how the model-checking technique based on Petri net unfoldings could be used in the verification of π-calculus terms. Our starting point was a compositional translation from a finite fragment of the π-calculus into *p-nets* — a class of high-level Petri nets — proposed in [7], which is not directly susceptible to verification due to the presence of a place with infinitely many tokens.

We developed prototype tools based on this 'theoretical' translation of [7], and an existing efficient unfolder and verifier [10, 12]. It should be stressed that developing the prototype was not a straightforward task. The first problem is that p-nets do not conform to the input format required by the unfolder (though it accepts high-level nets with inscriptions used by p-nets, the latter also use read arcs and non-safe places which are not supported). In particular, the so-called *tag-places* used by p-net for managing coloured tokens representing π-calculus channels have infinite markings and so are not directly implementable. Other problems concerned the infinite branching in the operational semantics of π-calculus expressions, and an efficient implementation of read arcs used by the theoretical translation. We describe both the problems and concrete solutions we adopted in order to deal with them, as well as give initial experimental results in support of our specific design choices.

The model-checking approach outlined in this paper is suitable for the task of model-checking of finite π-calculus terms. To lift it to the recursive (or iterative) case requires further investigation as this cannot in general be done (sice the full π-calculus is Turing powerful), and so one needs to identify sufficiently rich yet still manageable fragment of the full process algebra. Note that an alternative way of using the technique proposed here would be to unroll a recursive π-calculus expression to a certain pre-determined depth, and then apply the model-checking developed for the resulting finite π-calculus terms.

The paper is structured as follows. The next three sections recall the details of the used variant of π-calculus, p-nets, and the translation of [7]. The main case study used in the experiments is also introduced. Sections 5 and 6 describes implementation issues encountered in our work and outlines various decisions (together with experimental support) which were adopted to address them. Finally, Section 7 presents some modifications of the main case study together with the corresponding experimental results.

2 π-calculus

The concrete syntax of the (finite) π-calculus [14, 15] we use is given below, assuming that $\mathbb{C}$ is a countably infinite set of *channels* ranged over by the first few lower case Roman letters, and τ denotes the invisible action:

$$\ell ::= \bar{a}b \mid ac \mid \tau \qquad \text{(output / input / internal prefixes)}$$
$$P ::= \mathsf{0} \mid \ell\,.\,P \mid P+P \mid P|P \mid (\boldsymbol{\nu}c)P \qquad \text{(agents)}$$

Agents are defined up to the *alpha-conversion* (i.e., bound variables can be changed in a consistent way provided that name clashes are avoided); $ac\,.\,P$ (input) and $(\boldsymbol{\nu} c)P$ (restriction) *bind* the channel c in P; and we denote by $fn(P)$ the free (i.e., not bound) channels of P. Moreover, $\{b/c\}P$ denotes the agent obtained from P by replacing all free occurrences of c by b, possibly after alpha-converting P in order to avoid name clashes; for instance, $\{b/c\}\bar{a}c\,.\,cd\,.\,\mathsf{0} = \bar{a}b\,.\,bd\,.\,\mathsf{0}$.

The translation of [7] worked with operational semantics of process expressions presented in [3, 4] and reproduced with minor adjustments in Table 1. It is based on transition steps of the form

$$A \vdash P \xrightarrow{\ \ell\ } B \vdash Q\,,$$

where ℓ is a prefix and $A, B \subset \mathbb{C}$ are finite sets of *indexing* channels. Its intuitive meaning is that

"in a state where the channels A may be known by agent P and by its environment, P can do ℓ to become agent Q and the channels B may be known to Q and its environment" [4].

Note that Q may know more channels than P, e.g., an input $\ell = ab$ adds b when $b \notin A$ (intuitively, b is a new channel received from the environment – see the In rule in Table 1). An expression $A \vdash P$ is *well-formed* if: $fn(P) \subseteq A$; no bound channel of P is present in A; and each bound channel only generates a single binding, i.e., there are no name clashes (using alpha-conversion one can always convert a given expression to a well-formed one).

The implemented prototype requires as input a well-formed π-calculus expression of the form

$$A \vdash (\boldsymbol{\nu} c_1) \ldots (\boldsymbol{\nu} c_n)\ P$$

where P has no subexpression of the form $(\boldsymbol{\nu} c)P'$ (a given expression can always be converted to this format).

2.1 An example

In our experiments we used simple 'classroom' specifications of the following form ($n \geq 2$):

$$\textsc{Ness}(n) \overset{\text{df}}{=} \{a, done\} \vdash (\boldsymbol{\nu} h)(\boldsymbol{\nu} h_1) \ldots (\boldsymbol{\nu} h_n)\ (T|S_1| \ldots |S_n)$$

where T represents a 'teacher' process, and each S_i represents a 'student' process. Their respective definitions are as follows (since we use longer, more meaningful, channel names to enhance the readability of the code, the input prefixes are denoted as $a?b$ and the output ones as $a!b$):

$$T \overset{\text{df}}{=} a?ness\,.\,(h_1!ness\,.\,h_1?x_1\,.\,\mathsf{0} \mid \cdots \mid h_n!ness\,.\,h_n?x_n\,.\,\mathsf{0})$$

$$\begin{aligned} S_i \overset{\text{df}}{=} h_i?addr_i\,.\,(&h!h_i\,.\,h_i!done\,.\,\mathsf{0} \\ &+ h?another_i\,.\,addr_i!h_i\,.\,addr_i!another_i\,.\,h_i!done\,.\,\mathsf{0}) \end{aligned}$$

$$\textsc{Tau}\ \frac{}{A \vdash \tau \,.\, P \xrightarrow{\tau} A \vdash P} \qquad \frac{}{A \vdash ac \,.\, P \xrightarrow{ab} A \cup \{b\} \vdash \{b/c\}P}\ \textsc{In}$$

$$\textsc{Out}\ \frac{}{A \vdash \bar{a}b \,.\, P \xrightarrow{\bar{a}b} A \vdash P} \qquad \frac{A, c \vdash P \xrightarrow{\bar{a}c} A, c \vdash P' \qquad a \neq c}{A \vdash (\boldsymbol{\nu} c)P \xrightarrow{\bar{a}c} A \cup \{c\} \vdash P'}\ \textsc{Open}$$

$$\textsc{Par}\ \frac{A \vdash P \xrightarrow{\ell} A' \vdash P'}{A \vdash P|Q \xrightarrow{\ell} A' \vdash P'|Q} \qquad \frac{A, c \vdash P \xrightarrow{\ell} A', c \vdash P' \qquad c \notin ns(\ell)}{A \vdash (\boldsymbol{\nu} c)P \xrightarrow{\ell} A' \vdash (\boldsymbol{\nu} c)P'}\ \textsc{Res}$$

$$\textsc{Sum}\ \frac{A \vdash P \xrightarrow{\ell} A' \vdash P'}{A \vdash P{+}Q \xrightarrow{\ell} A' \vdash P'} \qquad \frac{A \vdash P \xrightarrow{\bar{a}c} A' \vdash P' \qquad A \vdash Q \xrightarrow{ac} A'' \vdash Q'}{A \vdash P|Q \xrightarrow{\tau} A \vdash (\boldsymbol{\nu} c \setminus A)(P'|Q')}\ \textsc{Com}$$

Table 1. Operational semantics of π-expressions, where: $ns(\tau) \overset{\text{df}}{=} \emptyset$; $ns(ab) = ns(\bar{a}b) \overset{\text{df}}{=} \{a, b\}$; the notation A, c stands for the disjoint union $A \uplus \{c\}$; and $(\boldsymbol{\nu} c \setminus A)P$ is P if $c \in A$ and $(\boldsymbol{\nu} c)P$ otherwise. Symmetric versions of Sum, Par and Com are omitted.

The idea is that the teacher first receives from the school electronic submission system[1] a channel *ness* using which the students are supposed to submit their work for assessment. The teacher passes this channel to all the students (using n parallel sub-processes), and (also in parallel) then waits for the confirmation that the students have finished working on the assignment before terminating. A student's behaviour is somewhat more complicated. After receiving the *ness* channel, students are supposed to organise themselves to work on the assignment in pairs and, after finishing, exactly one of them sends to the support system (using the previously acquired *ness* channel) two channels which give access to their completed joint work. The students finally notify the teacher about the completion of their work. The property of the system we attempted to verify was that all the processes involved in the computation successfully terminate by reaching the end of their individual code (as distinguished from a deadlock where some processes could still be stuck in the middle of their intended behaviour).

To illustrate the operational semantics describe above, we may observe that the following move is possible for the initial expression:

$$\begin{aligned}
&\{a, \mathit{done}\} \vdash (\boldsymbol{\nu} h)(\boldsymbol{\nu} h_1) \ldots (\boldsymbol{\nu} h_n)\ (T|S_1| \ldots |S_n) \\
&\xrightarrow{\bar{a}b} \\
&\{a, \mathit{done}, b\} \vdash (\boldsymbol{\nu} h)(\boldsymbol{\nu} h_1) \ldots (\boldsymbol{\nu} h_n)\ (T'|S_1| \ldots |S_n)
\end{aligned}$$

[1] Called Ness in Newcastle.

where b is the channel on which links to the completed pieces of coursework are to be submitted, and

$$T' \overset{\mathrm{df}}{=} h_1!b\,.\,h_1?x_1\,.\,\mathsf{0} \mid \cdots \mid h_n!b\,.\,h_n?x_n\,.\,\mathsf{0}$$

The above example allowed us both to have easily scalable specifications, and also ones which should satisfy the correctness property (deadlock-freeness) for all even n and failing to satisfy it for all odd values of n (as it is impossible to organise all the students in pairs in such a case). For example, one of the possible executions of NESS(3) is when the teacher acquires a channel from the school submission system (the $a?ness$ action), passes it on to the three students ($h_i!ness$ actions of the teacher process synchronise with the $h_i?addr_i$ actions of the student processes), then the first student student sends his part of work to the second students (the actions $h!h_1$ and $h?another_2$ synchronise) and the confirmation to the teacher (the actions $h_1!done$ and $h_1?x_1$ synchronise) and terminates, and the second student sends his part of work (the $addr_2!h_2$ action) followed by the first student's part of work (the $addr_2!another_2$ action) via the channel from the school submission system acquired via the teacher, and then sends the confirmation to the teacher (the actions $h_2!done$ and $h_2?x_2$ synchronise) and terminates; the third student, on the other hand, can execute neither $h!h_3$ nor $h?another_3$ since h is not known to the environment and there is no pair for internal communication. Hence the process has reached a deadlock, as the third student and the teacher have not properly terminated.

Finally, the example is interesting since it exhibits two completely *different* sources of state space explosion. The first is due to the way in which the students can be paired and decide which of the two students will be communicating with the coursework submission system. With n students, the number of ways in which this can be done is exponential (e.g., for even n there are $n!/(n/2)!$ different final states the system can reach). The other potential source of state space explosion is the high level of concurrency in communications with the external environment as well as between the processes. This kind of explosion is typically avoided by the unfolding based model-checking techniques, whereas techniques based on interleaving suffer from it.

2.2 Context-based expressions

The translation to nets starts with a more convenient, *context-based*, representation of π-terms, which uses two fresh sets of *restricted channels* $\mathbb{R}$ ranged over by the upper case Greek letters, and *channel holders* $\mathbb{H}$ ranged over by the first few lower case Greek letters. A *context* itself is a partial mapping $\varsigma : \mathbb{H} \to \mathbb{C} \uplus \mathbb{R}$ with a finite domain. The basic idea is to represent an expression like $\{b, d\} \vdash (\boldsymbol{\nu} c)ba\,.\,\bar{a}c\,.\,\bar{c}b\,.\,\mathsf{0}$ as a context based expression $\mathcal{P}{:}\varsigma$, where

$$\mathcal{P} \overset{\mathrm{df}}{=} \beta\alpha\,.\,\bar{\alpha}\gamma\,.\,\bar{\gamma}\beta\,.\,\mathsf{0}$$

is a *restriction-free* agent based solely on channel holders and

$$\varsigma \overset{\text{df}}{=} [\beta \mapsto b, \delta \mapsto d, \gamma \mapsto \Delta]$$

is a context providing their interpretation. From such a context we can read off that: α is a *type-I* channel holder corresponding to the binding channel in an *input* prefix (since α is not in the domain of the context mapping though it occurs in $\mathcal{P}$); β and δ are *type-K* channel holders corresponding respectively to the *known* channels b and d; and γ is a *type-R* channel holder corresponding to the *restricted* channel Δ.

3 An algebra of nets

The high-level Petri net model of [7] which is the target of the translation, called *p-nets*, was inspired by the box algebras developed in order to provide a compositional Petri net semantics of concurrent programming languages (see, e.g., [2, 1, 8]). Transitions in p-nets have three different kinds of labels:

- UV, Uv and $\bar{U}V$ for communication with the external environment. (Note that U, V and v are variables, and when a transition is executed, concrete values assigned to these variables generate actions of the form used by the rules of the π-calculus operational semantics.)
- τ for internal actions.
- *rcv* and *snd* for inter-process communication.

Places used to model control flow (e.g., sequencing or choice between executed actions) are labelled by their status symbols (*internal* places by i, and *interface* places by e and x, for entry and exit, respectively); the tokens they hold are the standard 'black' ones. *Holder* places, on the other hand, carry coloured tokens representing channels; their borders are thicker and their labels are simply the corresponding channel holders. Arcs adjacent to the former are the standard directed arcs, while those adjacent to the latter are high-level directed arcs and read arcs (see, e.g., [5, 18]) annotated with *channel variables* u, U, v and V.[2]

Consider now a context-based expression $\mathcal{P}{:}\varsigma$ for which the translation to p-nets is designed. The basic idea behind the use of holder places is to represent each channel holder α appearing within P by a unique α-labelled holder place. Then we have two cases:

- If α is of type-K or type-R, then the initial marking inserts a single coloured token $\varsigma(\alpha)$ into it.

[2] Note that a read arc *tests* (without consuming) for the presence of specific token(s) in the place it is connected to. Moreover, multiple read arcs can concurrently test for the presence of the same token.

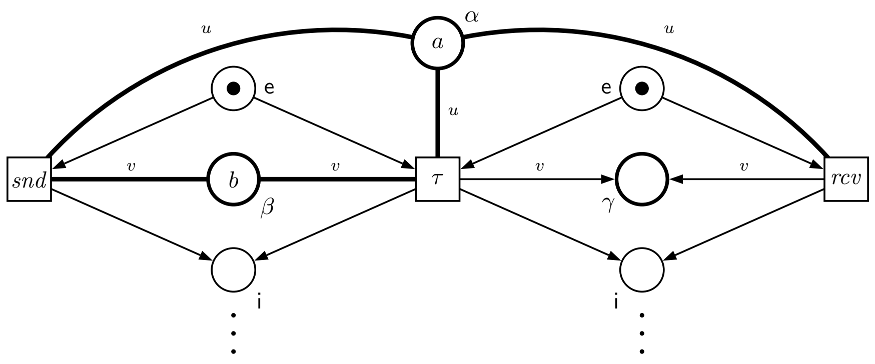

Fig. 1. A fragment of high-level Petri net.

- If α is of type-I, then it is initially empty and remains so until an internal communication or an input from the environment inserts a channel into it.

This idea is illustrated in Figure 1, which shows a fragment of the translation of the expression $(\bar{\alpha}\beta\ \ldots|\alpha\gamma\ \ldots) : \varsigma$, where $\varsigma \overset{\mathrm{df}}{=} [\alpha \mapsto a, \beta \mapsto b]$. The net fragment has also two black tokens in the entry places. One now can observe that the *rcv*-labelled transition is enabled since the right entry place contains a token and the α-labelled place contains the channel a. Similarly, the *snd*-labelled transition is also enabled and results in a transfer of the token from the left entry place to the left internal place. But what is really important is that the *fusion* of these two transitions, which is the τ-labelled transition, can also fire and the result is that the γ-labelled holder place acquires the channel b. In this way, the internal interprocess communication in π-calculus expressions can be implemented using net-theoretic devices.

The third and — from the point of view of our ultimate aim — the only troublesome kind of place, is a special *tag-place*. It is $\mathbb{T}$-labelled and indicated by a double border. It stores coloured *bookkeeping tokens*, each token having its special tag drawn from the set $\mathbb{T} \overset{\mathrm{df}}{=} \{\mathsf{K}, \mathsf{N}, \mathsf{R}\}$, as follows:

- K is for the known channels;
- N is for the new (yet unknown) channels;
- R is for the restricted channels.

The first case is rather complicated since a restricted channel, say Δ, may become known, say as c, due to being sent to the environment. This will be recorded by inserting a bookkeeping token $c.\Delta.\mathsf{K}$ into the tag-place, and then consulting it whenever necessary. A bookkeeping token may be of one of the following forms:

- $a.\mathsf{N}$ if a is a new (still unused) channel.
- $\Delta.\mathsf{R}$ if Δ is a restricted channel.

- $a.a.\mathsf{K}$ if a is a known channel (either a has always been known or a was initially new and then became known).
- $a.\varDelta.\mathsf{K}$ if the restricted $\varDelta$ has become known as a.

Consider, for example, the p-net fragment in Figure 2(a). The marking in the tag-place indicates that $\varDelta$ is a restricted channel; e is a new (yet unknown) channel, and a is a known one. The transition is enabled and its firing produces the marking shown in Figure 2(b). This firing illustrates how a restricted channel becomes known (which is represented by the insertion of the book-keeping token $e.\varDelta.\mathsf{K}$ in the tag-place), and corresponds to the OPEN rule in Table 1.

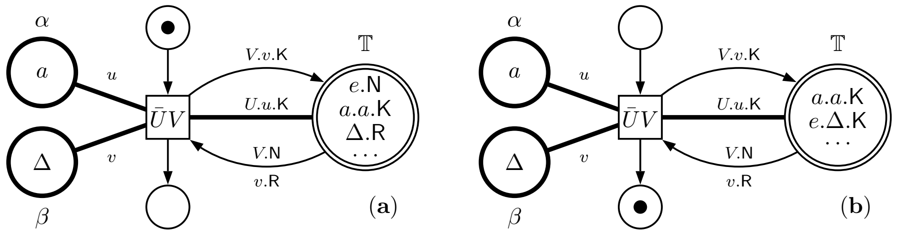

Fig. 2. Example showing how the tag-place is used.

Our brief presentation of the p-net model is completed by mentioning the composition operators which are defined for them. The concrete operations we need are prefixing, choice, parallel composition and restriction, as illustrated in Figure 3. Note that the restriction simply deletes the *rcv*- and *snd*-labelled transitions, while other operators combine control flow and/or holder places of the operand nets (note that the tag-places are always merged together into a single one).

4 From π-calculus terms to p-nets

Let $\mathcal{P}_0 : \varsigma_0$ be a context-based expression. The first stage of translation yields a compositionally derived p-net $\mathbb{K}(\mathcal{P}_0)$, in the following way. One first gives the translation for the base process $\mathsf{0}$ and the three prefixes, as shown in Figure 4. For example, each output prefix $\bar{\alpha}\beta$, when $\alpha \neq \beta$, is translated into the p-net $\mathbb{K}(\bar{\alpha}\beta)$ which may exhibit three kinds of behaviour, corresponding to the firing of three specific transitions:

- t^k: known output. A known channel matching V is sent through a channel matching U.
- t^n: new output. A new channel V is sent through a known channel U.

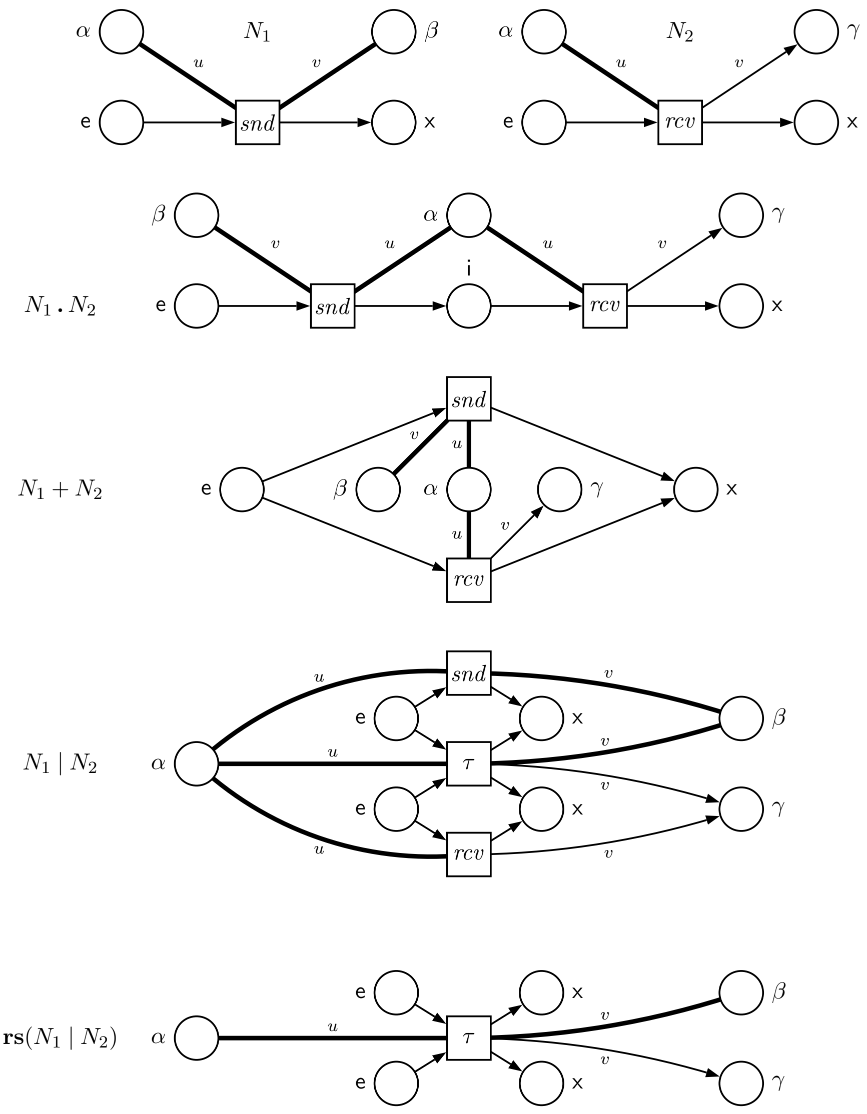

Fig. 3. Illustration of operators defined for p-nets: prefix, choice, parallel composition and restriction.

- t^c: communicating output. It is intended to synchronise with a corresponding communication input in order to provide the transfer of a channel v through the channel u, be it known or restricted.

The output prefix $\bar{\alpha}\alpha$ has a simpler translation, since α may not be both known and restricted, so that t^n is unnecessary. For compound sub-expressions, we

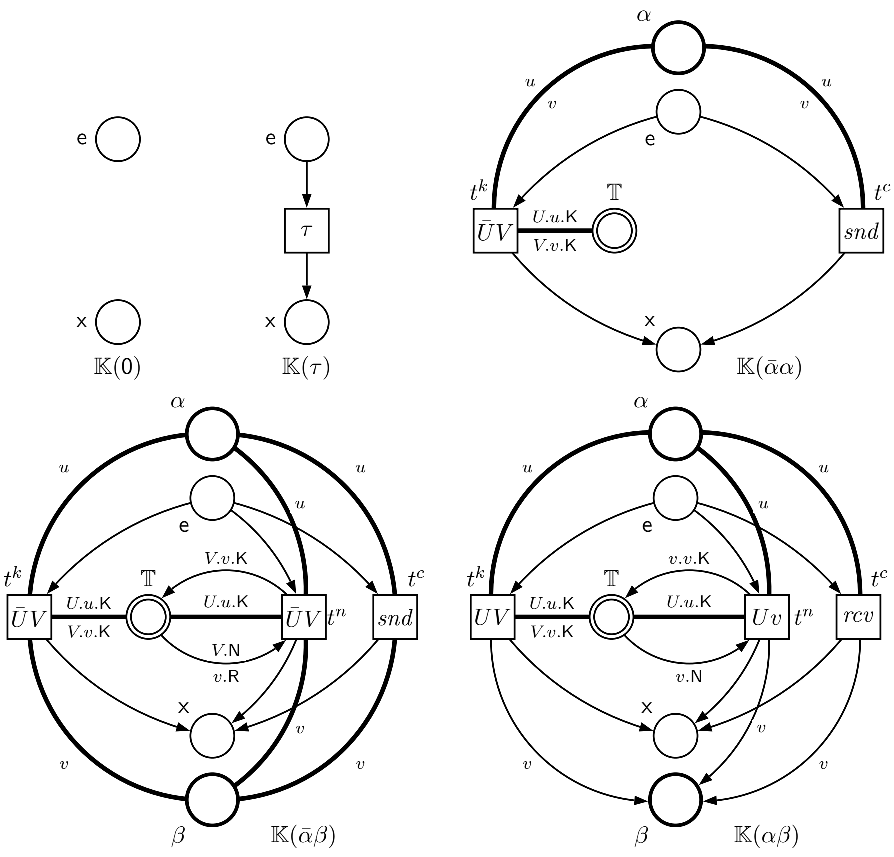

Fig. 4. The unmarked p-nets for 0 and π-calculus prefixes.

have:

$$\begin{aligned}\mathbb{K}(p\,.\,\mathcal{P}) &\overset{\text{df}}{=} \mathbb{K}(p)\,.\,\mathbb{K}(\mathcal{P})\\ \mathbb{K}(\mathcal{P}\;+\;\mathcal{P}') &\overset{\text{df}}{=} \mathbb{K}(\mathcal{P})\;+\;\mathbb{K}(\mathcal{P}')\\ \mathbb{K}(\mathcal{P}\mid\mathcal{P}') &\overset{\text{df}}{=} \mathbb{K}(\mathcal{P})\mid\mathbb{K}(\mathcal{P}')\,.\end{aligned}$$

The target p-net $\mathbb{PN}(\mathcal{P}_0{:}\varsigma_0)$ is then obtained by first applying the restriction operator, and then inserting one black token into each entry place, one $\varsigma(\alpha)$ token into each α-labelled type-K holder place, and the following coloured tokens into the tag place:

- $a.a.$K for each initially known channel;
- $n.$N for each unknown channel (note that there are infinitely many tokens of this type);
- $\varDelta.$R for each type-R channel holder $\varDelta$.

The translation results in a p-net with a very close behavioural relationship with the original process expression. More precisely, [7] showed that the labelled transition system of $A \vdash P$ is *strongly bisimular* to the labelled transition system of the p-net $\mathbb{PN}(\mathcal{H})$, where $\mathcal{H}$ is any well-formed context based expression corresponding to $A \vdash P$.

5 Model-checking p-nets resulting from the translation

The theoretical translation described above is not directly implementable (due to the infinite number of tokens on the tag place), and in our work we set out to modify it in such a way that it can be model-checked by existing tools. Moreover, the specific model-checking technique we decided to use was right from the beginning the unfolding-based verification of Petri nets, which already proved to be efficient in dealing with, e.g., digital asynchronous systems and distributed programs [10].

There are several variations of the unfolding-based model-checking technique, including unfolders and property verifiers. The way p-nets are defined suggests that one should employ a variant which is capable to deal efficiently with coloured tokens and high-level arc annotations. In particular, one should avoid the expansion of the high-level Petri net to its low-level representation and unfolding the latter, since such an approach may yield a huge intermediate low-level net, rendering the whole attempt practically useless (see [12] for a thorough discussion of this issue). We therefore decided to use the Punf model-checker (complemented by the MPSat property verifier based on a SAT-solver), which directly applies unfolding to high-level nets without expanding them to low-level nets, as described in [12]. Having decided on the particular unfolding approach, we then had to address a number of implementation issues, as described in the rest of this paper.

5.1 Key implementation issues

A basic problem we faced right from the outset was that p-nets are not compatible with the input required by the unfolder, in that Punf requires as input a high-level net which: (i) is *strictly* safe in the sense that no place can ever hold more than one coloured token; and (ii) does not include any read-arcs. Another, even more fundamental, issue is that the p-net resulting from the translation will in all but the simplest cases have infinite state space. The reason is that the set of potentially known (or new) channel names is countably infinite, and so any input prefix which receives a name from the environment will necessarily give rise to infinite branching. As a result, a naïve state space exploration through exhaustive enumeration is bound to fail.
The way in which addressed these problems is described next.

Infinity of new channels.

We are primarily interested in checking state properties of mobile systems expressed using π-calculus expressions. For this reason, the main property of channels we are interested in is whether a channel which has been received from the environment is brand new (i.e., fresh) or the same as one of the already known channels. As far as sending to the outside of channels previously restricted is concerned, they are always brand new (fresh), i.e., different from those already known. As the precise identity of a channels which has just become known is irrelevant for our purposes, we proceed as follows: (i) if a restricted Δ is sent outside it becomes known simply as Δ; and (ii) if input into a channel holder α happened and the inserted channel is a fresh channel, its identity is set to α, otherwise it is one of the existing known channels. In this way, the number of known channels other than those present initially is bounded by the total number of type-I and type-R channel holders. The resulting model can therefore be made bounded and then model-checked. It is worth observing that this treatment of newly known channels is a kind of *symmetry reduction* employed at the level of system modelling.

Read arcs.

We use the standard simulation of a read arc using two directed arcs pointing in the opposite directions. Intuitively, this replaces a non-destructive read operation by a destructive one, followed by a re-write. One can observe that this transformation preserves the interleaving semantics of the net.

Non-safeness.

Holder places in p-nets resulting from translation are strictly safe,[3] and so conform to the input required by PUNF. The tag-place, on the other hand, is never safe (actually, in the theoretical translation its marking is always infinite). However, with the decisions about the modelling of known channels made above, we can simulate the working of the tag-place by introducing a *status* place for each type-R and type-I channel holder α. The status place holds always one token, 0 or 1, and its meaning is as follows. For a type-R status place, 0 means that α is restricted and 1 means that is has become known. For a type-I status place, 1 means that the α is non-empty and received a fresh channel from the environment; otherwise it contains 0. Initially, all the status places contain 0's.

Partial transition expansion.

An immediate side-effect of replacing the tag-place by a finite set of status places is that each of the transitions used in the original translation accessing

[3] This is not true for an arbitrary p-net, and so our model-checking approach works only for those nets which result from the translation.

the tag-place needs to be replaced by a set of transition jointly simulating the effect of a single transition in the original translation (note that transition modelling internal communication do not need to be modified). The overall effect was that the number of transition increased but their arc annotations became simpler.

Reducing the number of holder places.

An important simplification that we applied right from the start was that only holder places used for inputting new channels have been retained; other holder places used in the theoretical translation have been hard-wired into transition guards and arc annotations. The overall effect was that the number of places decreased, some of arc annotations become simpler (constants in $\mathbb{C} \uplus \mathbb{R}$ replaced some of the variables used previously), and some transition guards changed from *true* to something like $a = u$, where $a \in \mathbb{C} \uplus \mathbb{R}$ and u is a variable used in an adjacent arc annotation.

6 Experimental results

Our experimental results are presented in Table 2. The meaning of the table entries is as follows. The first column identifies a specific instance being model-checked. After that we give the size of the high-level Petri nets derived for the input expression ($|P|$ and $|T|$ provide the numbers of places and transitions, respectively), as well as the size of finite prefix of its unfolding ($|B|$ and $|E|$ provide the numbers of conditions and events, respectively). The last two columns show the times (in seconds) needed to generate the unfolding using PUNF, and to verify the chosen correctness criterion (i.e., deadlock-freeness) using MPSAT.

6.1 Implementation I

The first implementation of the theoretical translation followed in a straightforward all the key choices and modification described above. The results are identified as the NESS(2):I and NESS(3):I entries in Table 2. However, when we tried to unfold NESS(4):I, the whole process became time consuming and we abandoned it after few hours, as it was clear that it is not reasonable to use this implementation in practice.

6.2 Implementation II

The reason behind the disappointing performance of the first implementation attempt was that the simulation of read arcs by pairs of consuming arcs degrade concurrency, quickly leading to an unacceptable growth of the resulting unfolding. Intuitively, this happens because concurrent non-destructive

Problem	Net		Prefix		Time, [s]	
	$\|P\|$	$\|T\|$	$\|B\|$	$\|E\|$	Punf	MPSat
Ness(2):I	38	200	2883	881	<1	<1
Ness(3):I	55	415	14235	4369	2	<1
Ness(2):II	667	200	5553	127	<1	< 1
Ness(3):II	1375	415	22222	366	2	< 1
Ness(4):II	2335	724	101005	1299	25	< 1
Ness(5):II	3547	1139	388818	4078	165	1
Ness(6):II	5011	1672	1180609	10431	1564	258
Ness(7):II	6727	2335	2971198	22662	25273	11
Ness(2):III	157	200	1413	127	<1	< 1
Ness(3):III	319	415	5458	366	1	<1
Ness(4):III	537	724	24561	1299	6	< 1
Ness(5):III	811	1139	93546	4078	46	<1
Ness(6):III	1141	1672	281221	10431	411	311
Ness(7):III	1527	2335	701898	22662	2904	8
sNess(2):II	1037	349	3419	57	<1	< 1
sNess(3):II	2145	710	14244	183	5	< 1
sNess(4):II	3649	1213	41423	423	22	< 1
sNess(5):II	5549	1870	96620	813	83	<1
sNess(6):II	7845	2693	194667	1389	239	243
sNess(7):II	10537	3694	353564	2187	616	<1
sNess(2):III	201	349	721	57	<1	< 1
sNess(3):III	409	710	2932	183	3	< 1
sNess(4):III	689	1213	8345	423	12	< 1
sNess(5):III	1041	1870	19156	813	28	<1
sNess(6):III	1465	2693	38137	1389	71	202
sNess(7):III	1961	3694	68636	2187	170	<1
detNess(4):II	1433	511	10429	181	3	<1
detNess(6):II	3083	1257	28166	342	36	<1
detNess(8):II	5357	2475	58863	551	289	<1
detNess(10):II	8255	4273	105976	808	2102	<1

Table 2. Experimental results.

read operations accessing the same place become sequentialised (in all possible ways) when they are replaced by destructive read-and-re-write operations (see [18] for a discussion of this phenomenon).

To allow a maximal level of concurrency in the resulting high-level net — which can significantly reduce the size of the unfolding prefix — we decided to replicate the status places described above as well as the type-I holder places. The intuition behind this transformation is to provide each reader of a place with its own replica of that place, which means that the read operations can happen concurrently and no sequentialising happens. Note that the transitions of the first implementations had to be suitably adjusted, but their overall number stayed the same as before.

The experimental results, identified by Ness(2):II–Ness(7):II in Table 2, improved dramatically and, in particular, confirmed that indeed Ness(n) does not have an illegal termination (deadlock) iff n is even, for $n \leq 7$. Note that the verification time for Ness(6) is much bigger that for Ness(7) since the latter contains a deadlock which is relatively quickly detected.

6.3 Implementation III

In a way, the second implementation used the maximal level of replication of status places and holder places one might wish to have. However, this in general may not be necessary. For example, one may observe that two transitions which can never be concurrently enabled can share the replicas of status and holder places as they will never need to access them concurrently. We therefore designed our third, and final, implementation which always uses the number of place replicas bounded by the maximal level of concurrency in π-calculus expressions modelling mobile system. A suitable bound can be determined statically, and for Ness(n) is equal to $2n$. The allocation of replicas to individual transitions is straightforward, and can be done by inspecting the parsing tree of the original π-calculus expression, i.e., statically.

The experimental results, identified by Ness(2):III–Ness(7):III in Table 2, again confirmed what was expected, i.e., that the resulting unfolding process is more efficient. Note that the number of events in the unfolding prefix is the same as for the second implementation, but the number of conditions and the prefix generation time have significantly decreased.

7 Further experiments

During the series of experiments we conducted, a closer analysis of firing sequences leading to deadlocks detected what might be considered as a 'security breach'. More precisely, the specification allows the whole protocol to terminate successfully in such a way that the students inform the environment (rather than the teacher), and the teacher receives the completion messages from the environment (rather than from the students). Intuitively, this means that the environment acts as an 'intruder in the middle'. Moreover, on the verification side, the possibility of 'too many' communications with the environment should also increase the size of the resulting unfolding. One would therefore expect that the situation becomes different if most of the communication is done within the system's processes. To test this hypothesis, and to address the security issue described above, we re-designed the original specification, in the following way:

$$\text{sNess}(n) \overset{\text{df}}{=} \{a, done\} \vdash (\boldsymbol{\nu} h)(\boldsymbol{\nu} h_1) \cdots (\boldsymbol{\nu} h_n)(\boldsymbol{\nu} hres)\ (T'' \mid S_1'' \mid \cdots \mid S_n'')$$

where:

$$T'' \overset{\mathrm{df}}{=} a?ness\,.\,(h_1!ness\,.\,h_1!hres.hres?x_1\,.\,\mathsf{0} \\ \mid \cdots \mid h_n!ness\,.\,h_n!hres\,.\,hres?x_n.\mathsf{0})$$

$$S_i'' \overset{\mathrm{df}}{=} h_i?addr_i\,.\,h_i?report_i\,. \\ (h!h_i\,.\,report_i!done\,.\,\mathsf{0} \\ + h?another_i\,.\,addr_i!h_i\,.\,addr_i!another_i\,.\,report_i!done\,.\,\mathsf{0})$$

Intuitively, sNess(n) uses a new secret channel *hres* to ensure that the communication between the students and the teacher about the completion of coursework does not leak outside the system. The results, identified by sNess(2):II–sNess(7):II and sNess(2):III–sNess(7):III in Table 2, confirmed our initial hypothesis.

Finally, we investigated the effect of reducing the state explosion due the protocol for pairing the students present in the case study. We therefore model-checked the system assuming that the students know in advance what the pairing is, and who is to communicate with the environment. This, of course, was only possible if n is even. The results, identified by detNess(4):II–detNess(10):II in Table 2, indicate that the unfolding copes very well with the state space explosion which is due to concurrency present in the system specification.

8 Conclusions

The results presented in this paper indicate that model-checking based on Petri net unfoldings can be a successful technique to deal with distributed systems with mobility. We can also identify at least three challenging areas of future work leading to potentially significant improvements in efficiency and applicability of the present approach:

- To develop an unfolding technique dealing with the read arc in a direct way, rather than simulating them using pairs of directed arcs.
- To introduce a restricted form of π-calculus recursion (or iteration) still allowing one to use model-checking.
- To deal with the state space explosion problem caused by aspects other than a high level of concurrency; there are strong indications that the recently proposed *merged processes* [11] could offer an effective solution.

Acknowledgement. This research was supported by the EC IST grant 511599 (Rodin) and RAEng/EPSRC research fellowship EP/C53400X/1 (Davac).

References

1. E. Best, R. Devillers, and M. Koutny. *Petri Net Algebra.* EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 2001.

2. E. Best, W. Frączak, R. P. Hopkins, H. Klaudel, and E. Pelz. M-nets: an Algebra of High Level Petri Nets, with an Application to the Semantics of Concurrent Programming Languages. *Acta Informatica*, 35:813–857, 1998.
3. G. L. Cattani and P. Sewell. Models for Name-Passing Processes: Interleaving and Causal. In *LICS'2000*, pages 322–333. IEEE Computer Society Press, 2000.
4. G. L. Cattani and P. Sewell. Models for Name-Passing Processes: Interleaving and Causal. Technical Report TR-505, University of Cambridge, 2000.
5. S. Christensen and N. D. Hansen. Coloured Petri Nets Extended with Place Capacities, Test Arcs and Inhibitor Arcs. In *ICATPN'93*, volume 691 of *Lecture Notes in Computer Science*, pages 186–205. Springer-Verlag, 1993.
6. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
7. R. Devillers, H. Klaudel, and M. Koutny. Petri Net Semantics of the Finite pi-calculus Terms. *Fundamenta Informaticae*, 70:203–226, 2006.
8. R. Devillers, H. Klaudel, M. Koutny, and F. Pommereau. Asynchronous Box Calculus. *Fundamenta Informaticae*, 54:295–344, 2003.
9. G. Holzmann and M. Smith. Automating Software Feature Verification. *Bell Labs Technical Journal*, 5:72–87, 2000.
10. V. Khomenko. *Model Checking Based on Prefixes of Petri Net Unfoldings*. PhD thesis, School of Computing Science, University of Newcastle upon Tyne, 2003.
11. V. Khomenko, A. Kondratyev, M. Koutny, and V. Vogler. Merged Processes — a New Condensed Representation of Petri Net Behaviour. In *CONCUR 2005*, volume 3653 of *Lecture Notes in Computer Science*, pages 338–352, 2005.
12. V. Khomenko and M. Koutny. Branching Processes of High-Level Petri Nets. In *TACAS 2003*, volume 2619 of *Lecture Notes in Computer Science*, pages 458–472, 2003.
13. K. L. McMillan. Using Unfoldings to Avoid State Explosion Problem in the Verification of Asynchronous Circuits. In *CAV 1992*, volume 663 of *Lecture Notes in Computer Science*, pages 164–174, 1992.
14. R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes. *Information and Control*, 100:1–77, 1992.
15. J. Parrow. An Introduction to the π-calculus. In Bergstra, Ponse, and Smolka, editors, *Handbook of Process Algebra*, pages 479–543. Elsevier, 2001.
16. C. Pixley. Formal Verification in 2004. In *DATE 2004, EDA Tools Forum*, 2004.
17. A. Valmari. The State Explosion Problem. In W. Reisig and G. Rozenberg, editors, *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*, pages 429–528. Springer-Verlag, 1998.
18. W. Vogler, A. Semenov, and A. Yakovlev. Unfolding and Finite Prefix for Nets with Read Arcs. In *CONCUR 1998*, volume 1466 of *Lecture Notes in Computer Science*, pages 501–516, 1998.