Verifying π-calculus Processes by Promela Translation

Hosung Song and Kevin J. Compton

Department of Electrical Engineering and Computer Science
University of Michigan, Ann Arbor, MI 48109, USA
{hosungs,kjc}@umich.edu

Abstract. In this paper, the possibility of verifying π-calculus processes via Promela translation is investigated. A general translation method from π-calculus processes to Promela models is presented and its usefulness is shown by performing verification tasks with translated π-calculus examples and SPIN. Model checking translated π-calculus processes in SPIN is shown to overcome shortcomings of the Mobility Workbench, which implements a theorem-proving style μ-calculus model checking algorithm for the π-calculus.

1 Introduction

Processes in Promela models achieve concurrency through global variable sharing or message passing. Promela's message passing primitives have a unique feature: they can pass channels as messages in communication. The π-calculus [4, 5] is a canonical process algebra especially designed for mobile concurrent computation in a very abstract way. The π-calculus achieves this goal with the same idea: passing channels as messages in communication. This striking similarity leads us to think about the possibility of performing π-calculus verification tasks in Promela and SPIN environments.

Verification in the π-calculus is usually done by checking bisimulation equivalence. A more complex process representing an implementation is shown to be bisimilar to a simpler process representing a specification. The simpler process should be so clear that it can be regarded as satisfying correctness requirements in an intuitive sense, not with rigorous mathematical proof. This may not be the case for every occasion, and we believe there should also be a usable model checking verification method with temporal logic specification. There are model checking algorithms for the π-calculus with μ-calculus style specification logic [2, 1], but the logic itself is very complicated to describe or understand, and sometimes results are not obtained in reasonable amount of time even for the proofs of very simple correctness requirements.

In this paper, we suggest an alternative way of model checking π-calculus processes via Promela translation. We present a translation method from π-calculus processes to Promela models. A strategy to unify the model checking logics and styles between the π-calculus and Promela/SPIN is also suggested.

Experimental study with simple examples is presented to show the effectiveness of our approach to π-calculus model checking. Visualized counterexamples for negative results are the most immediate benefit we can get through our approach. Also, we find that some π-calculus problems, which fail to terminate in several hours using the usual π-calculus model checking method, can be solved with SPIN and Promela translation in less than a second. With these experiments, we believe our translation-based approach to π-calculus model checking can be a better alternative to the existing one.

This paper is organized as follows. In Section 2, we review necessary background knowledge: the π-calculus, the μ-calculus extension for the model checking, and an existing tool for this purpose. In Section 3, the translation method is presented for general π-calculus processes. In Section 4, various experiment results are presented to show the usefulness of our approach. Section 5 concludes the paper with interesting thoughts about the extension of this research.

2 Preliminaries

We briefly review the π-calculus, the π-μ-calculus and the Mobility Workbench (MWB) in this section.

2.1 The π-calculus

The π-calculus [4, 5] is a canonical process algebra which can describe mobile concurrent computation in an abstract way. It provides a way of defining labeled transition systems which can exchange communication channels as messages. The π-calculus used in this paper consists of following components.

- *Action.* The set $\mathcal{N}$ of names (for actions). The set of co-names is $\overline{\mathcal{N}} = \{\overline{x} \mid x \in \mathcal{N}\}$. The set of labels $\mathcal{L}$ is $\mathcal{N} \cup \overline{\mathcal{N}}$. The set of actions Act is $\mathcal{L} \cup \{\tau\}$, where τ is the label for silent actions.
- *Action prefix.* The *action prefixes* π of the π-calculus are a generalization of the actions introduced above; an action prefix represents either sending or receiving a message (a name), or making a silent transition. The syntax is

$$\begin{array}{rll} \pi ::= & x(y) & \text{receive } y \text{ along } x \\ & \overline{x}\langle y \rangle & \text{send } y \text{ along } x \\ & \tau & \text{unobservable action} \end{array}$$

  where $x, y \in \mathcal{N}$.
- *Process.* The set P^π of π-calculus *process expressions* is defined by the following syntax:

$$P ::= \mathbf{0} \;\mid\; \sum_{i \in I} \pi_i.P_i \;\mid\; P_1|P_2 \;\mid\; (\nu a)\, P \;\mid\; [x = y]P \;\mid\; !P$$

  where $a \in \mathcal{N}$, π_i is an action prefix, and I is any finite indexing set.

In the above definition of P^π, there are 6 kinds of processes.

- *Null.* $\mathbf{0}$ is the deadlocked process which cannot involve with any transition.
- *Prefixed sum.* $\sum_{i\in I} \pi_i.P_i$ can proceed to P_i by taking the transition of the action prefix π_i. Transitions and nondeterministic choices are described as prefixed sums.
- *Parallel composition.* $P|Q$ is a process consisting of P and Q which will operate concurrently, but may interact with each other through actions/co-actions.
- *Restriction.* $(\nu a)\ P$ means that the action/co-action a or $\overline{a}$ in P can neither be observed outside, nor react with $\overline{a}$ or a outside the scope of P. This operator is for the description of restricted internal actions/co-actions. Also the effect of any restriction requires a new instantiation of the local name a inside the restriction.
- *Match.* $[x = y]P$ behaves like P if the names x and y are identical, and otherwise like $\mathbf{0}$.
- *Replication.* $!P$ means that the behavior of P can be arbitrarily replicated.

Structural operational semantics of the π-calculus is given by reaction and transition rules. One of the rules is the reaction rule:

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

where $\{z/y\}P$ means syntactic substitution of names for names. Every free occurrence of y in P is substituted with z. The meaning of this reaction is as follows. $x(y).P$ (input to y along x, then P) is chosen for the left component, and $\overline{x}\langle z\rangle.Q$ (output of z along x, then Q) is chosen for the right component. The result of this reaction is the parallel composition of the remaining parts of each component with free occurrences of y in P substituted by z.

For more details of the π-calculus, we refer to [4, 5].

2.2 The π-μ-calculus

A logic for specifying correctness requirements for π-calculus processes was suggested in [2] and is called the π-μ-calculus. The π-μ-calculus extends the μ-calculus logic to reason about the messages that accompany any π-prefix. In addition to the usual components of the μ-calculus (base propositions, conjunction, disjunction, possibility modality, necessity modality, fixpoint expression), the π-μ-calculus also has the constructs for quantifying message contents. For more details, we refer to [2, 1]. In this paper, we briefly reproduce from [1] an example of the π-μ-calculus logic formula for the correctness requirement of no message loss. The target π-process is any buffer with input channel i and output channel o.

$$\nu L.([t]L \wedge [\overline{o}]\Sigma u.L \wedge [i]\Pi u.(\nu I(u).([t]I(u) \wedge [i]\Pi z.I(u) \wedge [\overline{o}]\Sigma z.z = u))(u))$$

Note that the usage of square/angle brackets is different from SPIN's LTL. Square brackets are used for necessity modality, not the "always" operator, and

angle brackets are used for possibility modality, not the "eventuality" operator. The meaning of the formula is also reproduced from [1].

> L("nothing has been input yet") must hold after all transitions, except after an input $i(u)$ when $I(u)$ holds. $I(u)$("u has been input but not output") holds after all transitions, except after an output $\overline{o}\langle u\rangle$ (then nothing more is required) or after an output $\overline{o}\langle z\rangle$ with $z \neq u$ (then it is false). So $I(u)$ means nothing can be emitted before u, and L means that nothing can be emitted before the first received item.

In general, the π-μ-calculus specification of correctness requirements is much more complicated than the μ-calculus specification (not to mention CTL or LTL) because message contents need to be quantified accordingly. In the above example, Σ and Π are for that purpose. This is due to the model checking style– there is no environment process and the correctness requirement should consider every possible configuration of any possible environment. Since we are going to translate π-calculus processes into Promela, in which models under verification should be closed by suitable environment processes, we will not encounter such complicated name-quantified π-μ-calculus formulas if we also add environment processes to models in the π-calculus.

2.3 The Mobility Workbench

The Mobility Workbench (MWB) [6] started as a bisimulation equivalence checker. It contains everything needed for π-process analysis: π-grammar, parser, abstract representation, executor and bisimulation checker. A model checking algorithm [2] for π-μ-calculus temporal logic formulas was also included and later improved with another algorithm [1]. To the best of our knowledge, the MWB is the only tool available for simulation and verification of the π-calculus. Its source code, written in SML/NJ, has all the useful functionalities, so it is very natural to implement our translation program as an add-on to the MWB utilizing MWB basis.

The first user interface of the MWB would be the process description command. The detailed grammar and command description can be found in [6, 1]. The following are examples of MWB process descriptions.

```
agent Buf1(i,o) = i(x).'o<x>.Buf1(i,o)
agent Buf2(i,o) = (^m)(Buf1(i,m) | Buf1(m,o))
agent Buf20(i,o) = i(x).Buf21(i,o,x)
agent Buf21(i,o,x) = i(y).Buf22(i,o,x,y) + 'o<x>.Buf20(i,o)
agent Buf22(i,o,x,y) = 'o<x>.Buf21(i,o,y)
```

`agent` is the keyword for defining a π-process in the MWB. This differs from formal π-calculus grammar in that the output prefix $\overline{o}\langle x\rangle$ is typed as `'o<x>` and the restriction $(\nu m)\ P$ is typed as `(^m) P`. Also, the silent action τ is typed as `t`. No replication is allowed for π-process description in the MWB because of the decidability issue. In the above examples, `Buf1(i,o)` is a 1-cell FIFO (First In

First Out) buffer, `Buf2(i,o)` is a 2-cell FIFO buffer with parallel composition, and `Buf20(i,o)` is a 2-cell FIFO buffer without parallel composition. All free names in any process description should be parameterized as above. Primitive verification tools are the model executor (simulator) with the command `step` and the bisimulation checker with the commands `eq`, `weq`, `eqd` and `weqd`. The different bisimulation checking commands are for different bisimulation flavors and name distinctions.

Model checking π-μ-calculus formulas with π-processes is also available in the MWB. The π-μ-calculus formula in the previous subsection can be typed in the MWB as follows.

```
nu L.(([t]L) & (['o]Sigma u.L) & ([i]Pi u.(nu I(u).(([t]I(u))
     & ([i]Pi z.I(u)) & (['o]Sigma z.z=u)))(u)))
```

This differs from the formal π-μ-calculus in using the Greek letter representation and the output transition representation with `'`. The interface for the more recent model checking algorithm [1] is:

```
MWB> prove {pi-process} {pi-mu-formula}
```

3 Translating π-calculus processes to Promela

With the background knowledge of the π-calculus as described in the previous section, we present the details of the translation of π-calculus processes into the Promela language. The following translation methods are implemented as an add-on to the MWB in several hundred lines of SML/NJ codes.

3.1 π-names

π-calculus names are used not only for communication channels but also for messages. We may first be tempted to define some π-names as `mtype` Promela constants and others as Promela channels (`chan` type). This strategy was used for the mobile phone handover example in the SPIN distribution and seemingly worked fine. This strategy forces a channel to deal with communication of not only messages, but also channels. This causes quite a few type clash errors during simulation runs of the mobile phone handover example, but the simulation results as well as the verification results seem all fine due to the similarity of the internal representation of `mtype` constants and channels (`chan`). In our translation, we require the strong type consistency. That is, we define every π-name as a Promela channel with capacity 0. This will force the Rendez-Vous message passing in SPIN verification, which gives the same semantics for π-calculus synchronous communication.

One restriction here is that "polyadic" π-calculus cannot be supported. Polyadicity means a channel can be used for as many messages as needed at once. This contradicts to the concept of communication in Promela. Thus, our translation supports monadic π-calculus, which restricts that exactly one message can be exchanged in any message passing. This will result in the translation of any instantiated π-name `n` to Promela channel definition:

```
chan n = [0] of { chan };
```

Uninstantiated (e.g. bound or parameterized) names are just declared as Promela channels.

3.2 π-processes

A π-calculus process is translated into a Promela `proctype` process in a straightforward manner. Parameters in a π-process description are naturally matched with parameters in the corresponding Promela `proctype` definition. There are two more things to be taken care of. One is the declaration of bound names unparameterized in π-process description. In the π-calculus, there are bound names which are not parameterized at all, such as names for incoming messages. In `agent Buf(in,out)=in(x).'out<x>.Buf(in,out)`, `x` is not parameterized, but it is bound. This kind of names cannot be used in Promela translation without proper declarations, so we need to declare these names in the head of actual process definition. The other is to give a label for possible looping. In the above example, the process is just a loop expressed in a recursive way. To enhance the quality of the translation, such a self-loop is detected and process instantiation (`run`) is replaced by a `goto` statement. To achieve this looping for recursion, a label for the process should be given. We give a label for every process translation.

To summarize, a π-process:

```
agent Proc(p1,...,pn)= ... (process body containing
                            bound names b1,...,bm)
```

will be translated into a Promela process:

```
proctype Proc(chan p1,...,pn)
{
    chan b1,...,bm;
looplabel__Proc:
    ... (translation of the process body)
}
```

One final point to make is that π-processes without parameters are translated into `active proctype` processes in Promela. This is because Promela requires some process(es) to be initially instantiated and π-processes without parameters are the best candidates for this purpose.

3.3 π-prefixes

Prefixes in π-calculus processes are communication primitives and they are naturally matched with message passing primitives in Promela. A message sending prefix `'out<msg>` is translated into `out!msg`, and a message receiving prefix `in(msg)` is translated into `in?msg`. To extend our translation a little further by incorporating CCS processes, prefixes without any messages are translated with a dummy message, which is globally defined for each translation as

`chan dummy__msg = [0] of { chan };`. Output prefix `'out` will be translated into `out!dummy__msg` and input prefix `in` will be translated into `in?dummy__msg`.

Finally, a silent internal transition τ in the π-calculus (`t` in the MWB grammar) is translated into one line of `skip` command in Promela language.

3.4 π-match

A π-match `[a=b]` is simply translated into a Promela boolean expression and implication `a==b ->`. This preserves the meaning of the match.

3.5 π-restriction

Restriction in π-calculus is the only way of defining and instantiating new names, which are also channels. Other names are either passed from the calling process or bound by message input prefixes. By passing locally instantiated restricted names to other processes, the π-calculus achieves powerful scope extrusion and mobility support.

New names should be introduced before other process behavior descriptions. But the π-calculus grammar allows introduction of new names in the middle of arbitrary process behavior descriptions. To make translation simpler, we rewrite such processes so that rewritten processes contain new name introduction always in front of their process definition. For example, a π-process

```
agent Gen(in) = in(out).(^new)'out<new>.Gen(in)
```

is first rewritten to (only in the translation algorithm):

```
agent Gen(in) = in(out).Gen__1(in,out)
agent Gen__1(in,out) = (^new)'out<new>.Gen(in)
```

and then translated into:

```
proctype Gen(chan in)
{
    chan out;
    in?out; run Gen__1(in,out)
}
proctype Gen__1(in,out)
{
    chan new = [0] of { chan };
    out!new; run Gen(in)
}
```

Notice that rewriting breaks the loop structure and relies on recursive procedure calls. The best way of dealing with this problem would be to do alpha conversion and move such new name introduction in front of process definition. But such alpha conversion works only for parallel composition and prefixes, and there can be new name introduction in the middle of summed process definition, for which alpha conversion is impossible. Rather than applying different techniques, we decided to use same simpler translation for all occasions of new name introduction. This idea preserves the meaning of new name introduction in a clearer way.

3.6 π-sum

π-sum is essentially about nondeterministic choices, which are best matched with the `if` construct in Promela. The translation is also straightforward. If a π-process definition is:

```
agent P(a1,...,ak) = ... (body for choice 1)
                   + ... (body for choice 2)
                   ...
                   + ... (body for choice n)
```

then the translation will be:

```
proctype P(chan a1,...,ak)
{
... (new name definition, bound name declaration, loop label)
  if
  :: ... (translation for choice 1)
  :: ... (translation for choice 2)
  ...
  :: ... (translation for choice n)
  fi
}
```

Any sub-choice can have another sum and the `if` construct can be nested without any problem.

3.7 π-application

Any non-finite π-process definition ultimately leads to process calls. The buffer process `agent Buf(i,o)=i(x).'o<x>.Buf(i,o)` finally calls itself recursively achieving infinite loop behavior. A process can be transformed to another by calling it at the end of a choice. The buffer process can be divided into two-staged buffer like:

```
agent Buf2(i,o)=i(x).Buf21(i,o,x)
agent Buf21(i,o,x)='o<x>.Buf2(i,o)
```

This is just a trivial example, but shows the idea clearly. This π-process calling is called "application" in the literature. π-application can be obviously achieved by the `run` construct in Promela. There is one exception to enhance the quality of translation slightly. If a process calls itself, then the translation is not a recursive call, but an unconditional jump to the loop label declared in the beginning. We believe these two are essentially same, but since the recursive call will result in different process instantiations in simulation and verification, the looping would be a slightly better translation. In this case of looping translation, parameters need to be adjusted if they do not match one by one. This is also taken care of by the translation algorithm.

The buffer process examples shown above will be translated into:

```
proctype Buf(chan i,o)
{
    chan x;
looplabel__Buf:
    i?x; o!x; goto looplabel__Buf
}
proctype Buf2(chan i,o)
{
    chan x;
looplabel__Buf2:
    i?x; run Buf21(i,o,x)
}
proctype Buf21(chan i,o,x)
{
looplabel__Buf21:
    o!x; run Buf2(i,o)
}
```

3.8 π-composition

Parallel composition is the method to describe concurrency in any process algebra. In Promela, such composition can be achieved by multiple `run` statements in an `atomic` group. For example, a two-cell buffer π-process example:

```
agent Buf1(i,o)=i(x).'o<x>.Buf1(i,o)
agent Buf2(i,o)=(^m)(Buf1(i,m)|Buf1(m,o))
```

will be translated into:

```
proctype Buf1(chan i,o)
{
    chan x;
looplabel__Buf1:
    i?x; o!x; goto looplabel__Buf1
}
proctype Buf2(chan i,o)
{
    chan m = [0] of { chan };
looplabel__Buf2:
    atomic { run Buf1(i,m); run Buf1(m,o) }
}
```

This idea requires that components in any parallel composition should only be process calls. But in the π-calculus, valid processes can be composed in parallel. That means for example sums and prefixes can be composed. Here, we have no other choice than rewriting the original π-process as we did for new name introduction translation in Section 3.5. For instance, the translation of the following π-process:

```
agent SomeBuf2(i,o) = i(x).'o<x>.0 | i(y).'o<y>.SomeBuf2(i,o)
```

will be in Promela:

```
proctype SomeBuf2(chan i,o)
{
looplabel__SomeBuf2:
    atomic { run SomeBuf2__1(i,o); run SomeBuf2__2(i,o) }
}
proctype SomeBuf2__1(chan i,o)
{
    chan x;
looplabel__SomeBuf2__1:
    i?x; o!x
}
proctype SomeBuf2__2(chan i,o)
{
    chan y;
looplabel__SomeBuf2__2:
    i?y; o!y; run SomeBuf2(i,o)
}
```

4 Verifying π-calculus processes using Promela translation and SPIN

In this section, we discuss verification of π-calculus processes using the translation described above and SPIN. We also present results of simple experiments and comparison with the π-μ-calculus model checking approach.

4.1 Incompatible verification styles between the π-μ-calculus and SPIN

Promela's power for model description exceeds that of the π-calculus, given the translation strategy described in the previous section. However, there are definite differences between verification styles of two formalisms. The first issue is whether models under verification are closed or not. In SPIN model checking, models under verification should be closed. That is, we should also specify necessary environment processes to make verification work. The SPIN model checker can inspect every variable or process status in a Promela model, and this makes verification with closed models possible. This is not the case with the π-calculus verification. If a set of processes in the π-calculus is closed, then all we can observe from outside are the silent internal transitions (τ). Since transitions are the only observables in the π-calculus, we cannot do any meaningful verification with τ transitions only. Thus, π-calculus models under verification should always be open. π-calculus transitions are also accompanied by messages and this makes the π-μ-calculus specification logic [2] for the π-calculus very complicated.

The other discrepancy comes from the difference between the logics themselves. LTL is the logic for Promela verification, whereas the π-μ-calculus, an extension of the μ-calculus, is the logic for the π-calculus verification. It is well known that CTL formulas can be translated into μ-calculus formulas, but not the other way around. To the best of our knowledge, there is no general translation algorithm from LTL to the μ-calculus. Even for LTL formulas which are also CTL formulas, the textbook translation does not work because of the difference of meanings in base propositions in the π-μ-calculus. Base propositions in the π-μ-calculus are about future possibility—for example, $\langle a \rangle TT$ is satisfied by a process which can make an a transition, and $[a]FF$ is satisfied by a process which cannot make any a transition. This is clearly different meaning from that of SPIN LTL in which base propositions are about the current state, not about future possibility.

Due to these incompatibilities, model checking π-calculus processes cannot be made fully automatic by our translation strategy. One has to devise a proper environment for Promela-translated π-processes, and find valid LTL formulas for corresponding π-μ-calculus formulas. We could not think of any mechanical translation of these two logics, but there may be some way of achieving this if more time is spent in studying two logics.

4.2 Verification results with simple buffer examples

To claim the usefulness of the verification of π-processes by translation, we present the results of model checking of simple examples with the MWB and SPIN. The example systems are simple buffers (which may not be FIFO). Some of them are shown in Figure 1.

```
(* one cell buffer *)
agent Buf1(i,o) = i(x).'o<x>.Buf1(i,o)
(* two cell buffer with parallel composition *)
agent Buf2(i,o) = (^m)(Buf1(i,m) | Buf1(m,o))
(* three cell buffer with parallel composition *)
agent Buf3(i,o) = (^m)(Buf1(i,m) | (^n)(Buf1(m,n) | Buf1(n,o)))
(* lossy buffer *)
agent Buf1l(i,o) = i(x).('o<x>.Buf1l(i,o) + t.Buf1l(i,o))
(* bag with capacity 2 : no order preservation *)
agent Bag2(i,o) = Buf1(i,o) | Buf1(i,o)
```

Fig. 1. π-calculus description of simple buffers

The π-calculus models are self-explanatory with comments. Target properties for model checking are no message loss and order preservation, which are also used for a standard go-back-n flow control protocol verification in the SPIN

tutorial paper [3]. We also use the same environment process idea as in the paper: send arbitrarily many white messages, followed by a red message, arbitrarily many white messages again, a blue message and finally arbitrarily many white messages. Also, the same idea of global monitor variables (`sent` and `rcvd` to record which messages are sent and received) are used for our experiment. The same LTL formulas for the properties to be verified are used. For no message loss (NL), the correctness requirement is `[](sr -> <> rr)`, where `sr` is defined as `(sent == red)` and `rr` is defined as `(rcvd == red)`. The formula says that the sending of a red message, `sr`, implies its eventual reception, `rr`, and this condition should be true always. For order preservation (OP), the negated correctness requirement is `(!rr U rb)`, where `rb` is defined as `(rcvd == blue)`. This formula states that no reception of red messages (`!rr`) should hold until a reception of blue message (`rb`), which is the violation of order preservation property.

It is natural to expect that SPIN model checking of these properties with translated π-calculus simple buffers is trivial, because in the tutorial paper, these properties are verified with more complicated go-back-n flow control protocol. As expected, SPIN model checking of these properties with translated buffers gives correct answers in less than a second (excluding compilation time) for all cases. The lossy buffer `Buf1l` and the 2-cell bag `Bag2p` do not satisfy the two properties and SPIN gives negative answers with counterexamples, which clearly visualize the violating sequence in message sequence charts.

The above result does not seem to show superiority of model checking π-processes via Promela translation, because of the triviality. Advantages, however, can be shown when we compare the result with the π-μ-calculus model checking of same models. To make the comparison as fair as possible, the open π-calculus model should be incorporated with the environment as in SPIN model checking. The π-calculus model for the environment is as follows.

```
(* send w*,r,w*,b,w* and before sending r, raise the flag "sr" *)
agent Send(i,r,b,w,sr) = 'i<w>.Send (i,r,b,w,sr)
                       + sr.'i<r>.Send1(i,r,b,w)
agent Send1(i,r,b,w) = 'i<w>.Send1(i,r,b,w) + 'i<b>.Send2(i,r,b,w)
agent Send2(i,r,b,w) = 'i<w>.Send2(i,r,b,w)
(* receive messages. *)
(* for reception of r and b, raise the flag "rr" or "rb" *)
agent Recv(o,r,b,w,rr,rb) = o(x).(x.0|('w.Recv(o,r,b,w,rr,rb)
                                      +'r.rr.Recv(o,r,b,w,rr,rb)
                                      +'b.rb.Recv(o,r,b,w,rr,rb)))
```

The key transitions of sending a red message and receiving a red or a blue message should be observed from the outside model checker, so there are flags `sr` for sending a red message, `rr` for receiving a red message and `rb` for receiving a blue message. As stated earlier, the LTL formulas should be properly translated into the π-μ-calculus formulas, considering the different semantics of two logics. The following formulas are manually-translated μ-calculus formulas for the two properties.

```
nu X.([t]X & [sr] mu Y.(<rr>TT | (<t>TT & [t]Y)))
nu X.([t]X & [sr] mu Y.(<rr>TT | (<t>TT & [rb] FF & [t]Y)))
```

Note that `<>` is now used for the existential modality of the labeled transition rather than eventuality as in SPIN's LTL, and `[]` is used for the universal modality of the labeled transition rather than the always operator. The first formula states that the property `X` should hold always (greatest fixpoint `nu X`). It also states the property `X` should hold recursively after any silent internal transition (τ=`t`) and also after any `sr` (can send a red message) transition, property `Y` should hold eventually (least fixpoint `mu Y`). Property `Y` should hold if `rr` (can receive a red message) is possible, or else if a silent transition is possible, `Y` should hold recursively after any such transition. To summarize, the property states that after sending a red message, reception of the red message should be eventually possible. This means that the formula specifies NL property. The second formula is almost the same as the first one, except in the eventuality of the `rr` transition, there is one more condition that no `rb` (can receive a blue message) is possible. This effectively specifies the OP property: no reception of a blue message should be possible until a red message is received.

Example models	*Model checking results*			
	NL		OP	
	SPIN	MWB	SPIN	MWB
Buf1	Y, 0.1	Y, 0.24	Y, 0.1	Y, 0.37
Buf2	Y, 0.1	Y, 4.14	Y, 0.1	Y, 6.66
Buf3	Y, 0.1	Y, 473.12	Y, 0.1	Y, 776.4
Bag2	N, 0.1	N, 7.04	N, 0.1	N, 10.99
Buf1l	N, 0.1	?	N, 0.1	N, 1.82

(time in seconds, ? means not terminated after 8 hours)

Table 1. Experiment results with SPIN and MWB

Table 1 shows the execution times of model checking using the MWB and SPIN (with translated π-processes) for selected buffer processes and properties. Buf1, Buf2 and Buf3 are buffers with corresponding capacity. Bag2 is a 2-cell buffer without order preservation. It can also trap a message indefinitely, so NL does not hold for the Bag2. Buf1l loses input messages nondeterministically. These are defined in Figure 1 as π-calculus processes with MWB grammar. A Sun Ultra-10 (440 MHz) workstation with 512MB main memory was used for all experiments. As stated earlier, SPIN gave correct answers in less than a tenth second in all cases. For negative answers by SPIN, it also gave counterexamples as usual. The MWB spent more than 8 minutes to give answers for Buf3. Also, it failed to give an answer for the NL property of the lossy buffer even after 8 hours. It is also interesting to note that the answer for the OP property of the lossy buffer came out in less than two seconds and the OP seems more complicated

than the NL. Also note that the MWB gives no trace of the counterexample for negative answers.

We do not present the experiment results with a more substantial example, the mobile phone handover protocol. The MWB verification simply did not terminate in several hours, while the manual translation of the protocol by Gerard Holzmann in SPIN distribution is instantaneously verified.

The MWB theorem prover appears to require long execution times and sometimes results in nontermination, without any trace of counterexample in negative cases. The implementation in a very high level functional programming language (SML/NJ) might be an explanation for the inefficiency. Even though the advantages of model checking π-calculus processes with SPIN and Promela translation are due to the shortcomings of the MWB, we believe our approach can be a better alternative, considering the MWB is the only available π-calculus model checking tool.

5 Conclusion

In this paper, we studied the feasibility of model checking π-calculus processes using SPIN and Promela translation. The π-calculus is a canonical process algebra which can describe mobile concurrent computation. Because of the ability of passing channels to other processes in Promela description language, it is natural to think that any π-calculus process can be embedded in a Promela model by proper translation. We gave one such translation strategy in this paper. The translation procedure was implemented with SML/NJ as an add-on to the MWB and used to generate Promela translation of simple π-calculus buffer processes. With addition of environment processes, the first benefit of the translation is that the simulation experiment of π-calculus processes can be greatly improved graphically with the XSPIN interface. Also with manually translated π-μ-calculus formulas, we were able to do the verification of π-calculus processes for π-μ-calculus formulas more efficiently in the SPIN environment. It was not too surprising to find that SPIN could even give answers very quickly for seemingly non-terminating model checking problems with the π-μ-calculus and the MWB. SPIN also gives trace of counterexample for negative results as usual, whereas theorem-proving style π-μ-calculus model checker only gives a yes or no answer.

A number of interesting issues arose in this research. First of all, it was not easy at all to translate π-μ-calculus formulas into equivalent LTL formulas, or vice versa. The π-μ-calculus is for "labeled" transition systems, and its base propositions are about the future possibility. If we disregard all the labels, CTL formulas can be translated into general μ-calculus formulas, but no result is available for any other case. We believe that an automatic translation from μ-calculus to LTL would be almost impossible. It is also very difficult to specify and understand requirements written in the μ-calculus, so a better option would be just to translate π-calculus model only, and to specify the correctness requirements in SPIN and LTL formalism. In this regard, we believe our translation

routine will be useful for large scale π-calculus models, for which model checking type of verification is required and there have been no such attempts so far.

The translation procedure also needs a substantial improvement. Currently, its loop detection works only with self-looping π-processes. π-processes are usually mutually looping, and the current translation gives recursive calls for those looping processes. Since recursively spawned processes are regarded as different from parent processes, SPIN's cycle detection does not work and it keeps searching on and on until the depth limit is reached. However, detection of arbitrary mutual-looping behavior is also nontrivial. It requires complex data flow analysis and may lead to building a complete optimizing compiler. It may be a better way to make SPIN detect cycles from recursive calls.

As stated earlier, SPIN's verification models should be closed, while they should be open for the π-μ-calculus verification. To make closed verification work, we must also specify environment and verification should be done with that. Any such environment should be proved to be representative of any possible other configuration of environment. Open verification requires a complicated logic such as the π-μ-calculus, because it has to specify the quantification of messages along communications. It is, however, mathematically rigorous to let the logic specify every possible configuration. It may be interesting to extend SPIN's verification strategy to accommodate the open verification idea, even though it is much harder for explicit state-based model checking algorithms such as SPIN.

The initial verification effort in the process algebra community always starts from the bisimulation equivalence checking. The definition of bisimulation equivalence is always chosen so that two bisimilar processes satisfy the same set of temporal logic formulas. It would be also nice to have similar notion of equivalence in Promela so that equivalent Promela processes satisfy the same set of LTL formulas.

References

1. F. Beste. *The Model Prover – a Sequent-calculus Based Modal μ-calculus Model Checker Tool for Finite Control π-calculus Agents.* M.S. Thesis. Dept. of Computer Science, Uppsala University. January 1998. `ftp://ftp.docs.uu.se/pub/mwb/x4.ps.gz`.
2. M. Dam. Model checking mobile processes. In E. Best, editor, *CONCUR'93, 4th Intl. Conference on Concurrency Theory*, Vol. 715 of *Lecture Notes in Computer Science*, pp. 22–36. Springer-Verlag, 1993. Full version in Research Report R94:01, Swedish Institute of Computer Science, Kista, Sweden.
3. G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279-295, 1997.
4. R. Milner. The polyadic π-calculus: a tutorial. In F. L. Bauer, W. Brauer and H. Schwichtenberg, editors, *Logic and Algebra of Specification.* Springer-Verlag, 1993.
5. R. Milner, J. Parrow and D. Walker. A calculus of mobile processes, Parts I and II. *Journal of Information and Computation*, 100:1–77, September 1992.
6. B. Victor. *A Verification Tool for the Polyadic π-calculus.* DoCS Licentiate Thesis 94/50. Dept. of Computer Science, Uppsala University, May 1994.