

A Logical Encoding of the π -Calculus: Model Checking Mobile Processes Using Tabled Resolution^{*}

Ping Yang, C. R. Ramakrishnan, Scott A. Smolka

Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY, 11794-4400, USA
E-mail: {pyang, cram, sas}@cs.sunysb.edu

Abstract. We present MMC, a model checker for mobile systems specified in the style of the π -calculus. MMC's development builds on our experience gained in developing XMC, a model checker for an extension of Milner's value-passing calculus implemented using the XSB tabled logic-programming system. MMC, however, is not simply an extension of XMC; rather it is virtually a complete re-implementation that addresses the salient issues that arise in the π -calculus, including scope extrusion and intrusion, and dynamic generation of new names to avoid name capture. We show that tabled logic programming is especially suitable as an efficient implementation platform for model checking π -calculus specifications, and can be used to obtain an exact encoding of the π -calculus's transitional semantics. Moreover, MMC is easily extended to handle process expressions in the spi-calculus. Our experimental data shows that MMC outperforms other known tools for model checking the π -calculus.

1 Introduction

In [?], we showed that logic programming with tabulation can be used to construct an efficient model checker for concurrent systems. In particular, we presented XMC, a model checker supporting XL (an extension of Milner's value-passing CCS [?]) as the system specification language, and the alternation-free fragment of the modal μ -calculus as the property specification language.

XMC is written in XSB Prolog, where XSB [?] is a logic-programming system that extends Prolog-style SLD resolution with *tabled resolution*. The principal merits of this extension are that XSB terminates more often than Prolog (e.g. for all datalog programs), avoids redundant sub-computations, and computes the well-founded model of normal logic programs.

XMC is written in a highly declarative fashion. The model checker is encoded in less than 200 lines of XSB Prolog using a binary predicate `models/2` which defines when an XL term satisfies a modal μ -calculus formula. This definition

^{*} This work was supported in part by NSF grants EIA-9705998, CCR-9876242, CCR-9988155, and CCR-0205376; ONR grant N000140110967; and ARO grants DAAD190110003, DAAD190110019.

uses a ternary predicate `trans/3` which represents the transition relation of the labeled transition system corresponding to an XL specification.

Our experience with XMC raises the following question: Can tabled logic programming be brought to bear on the problem of verifying *mobile systems* and what new insights are required? In this paper we present *MMC*, a practical model checker for mobile systems specified in the style of the π -calculus [?].¹ The main technical difficulties that we encountered are due to the ability to express *channel passing* in the π -calculus, which give rise to a variety of issues that were not present in XMC, including scope extrusion and intrusion, and the generation of new names to avoid name capture.

Logic programming with tabulation turns out to be an ideal framework in which to implement a model checker for mobile systems. In particular, π -calculus names are represented as Prolog *variables* in MMC, which enables us to treat scope extrusion and intrusion, renaming, name restriction, etc., in a direct and efficient manner. The result is that the MMC version of the `trans` relation, when applied to a π -calculus expression p , generates the labeled transition system for p as prescribed by the π -calculus' transitional semantics [?].

By using the ability of a logic-programming engine to manipulate terms and to perform unification, we can encode monadic and polyadic versions of the π -calculus in a single framework. We can also treat the encryption/decryption constructs of the spi-calculus [?], an extension of the π -calculus for cryptographic protocols, as syntactic sugar. In fact, we can evaluate the operational semantics of spi-calculus processes without changing the `trans` relation in MMC. Thus MMC can also be viewed as a model checker for the spi-calculus.

Related work. A number of analysis techniques have been developed for the π - and spi-calculi, and many of them have been incorporated in tools. The Mobility Workbench (MWB) [?] provided the first model-checking tool for the polyadic π -calculus and the π - μ -calculus [?,?]. In addition to the model checker, MWB consists of a bisimulation checker and a prover based on sequent calculus [?]. Picasso [?] is a static analyzer for the π -calculus that focuses on checking secrecy of information such as process-level leaks and insecure communications. Cryptyc [?] uses static type checking to find security violations, such as secrecy and authenticity errors, in cryptographic protocols specified in the spi-calculus. More recently, techniques have been proposed for verifying secrecy and authenticity of cryptographic protocols specified in an extension of spi-calculus and with the intruder modeled using Prolog rules [?,?]. These techniques support the verification of an unbounded number of sessions of a protocol. In contrast, MMC can verify only a finite number of concurrent sessions, but, being a full-fledged model checker, can be used to verify other properties such as deadlock freedom and lossless transmission. Recent extensions to the Maude system [?], which uses equational and rewrite logic as a general framework for executable specifications, support cryptographic protocol analysis [?], mobile computation [?], and may-testing equivalence of non-recursive π -calculus processes [?]. MMC, in

¹ Please see [?] for MMC's source code, the source code of the examples used in this paper, and an extended version of this paper.

contrast, is a more traditional model checker for recursive mobile processes encoded in the π -calculus. There are also some other well-known tools for analyzing security protocols without using spi-calculus, such as FDR [?] and NRL [?].

Among these approaches, MMC is most closely related to the model checker implemented in the MWB. The property logic used in MMC is an expressive subset of the π - μ -calculus that is amenable to efficient implementation. The process language used in MMC, on the other hand, is more expressive than that of MWB, and permits encoding of spi-calculus specifications. The performance of MMC is considerably better than the model checkers and equivalence checkers of MWB reported in the literature [?,?]. Moreover, MMC is even comparable to that of the first versions of XMC where, as in MMC, labeled transition systems were generated by interpreting process terms (see Section 4).

In the following, Section 2 describes the computational basis of MMC: the encoding of the operational semantics of the π -calculus as a logic program, and the implementation of a model checker for a subset of π - μ -calculus in MMC. Section 3 describes extensions to MMC to support the spi-calculus. Experimental results appear in Section 4 and our concluding remarks are given in Section 5.

2 MMC: A Model Checker for the π -Calculus

In this section, we describe our model checker (called MMC for the *Mobility Model Checker*) for the π -calculus. Processes in the π -calculus are encoded as described in Section 2.1. The operational semantics of the π -calculus is encoded as a Prolog relation `trans`, which generates symbolic transition systems from agent definitions (Section 2.2). For simplicity, we first describe the encoding for the *monadic* π -calculus. This encoding is later optimized to reduce the size of the symbolic transition system and extended to the polyadic π -calculus. Finally, the semantics of π - μ -calculus is encoded as another Prolog relation `models` which determines whether a given π -calculus expression is in the model of a given alternation-free π - μ -calculus formula (Section 2.3).

2.1 Syntax of MMC Processes

We use \mathcal{P} to denote the set of all process (or agent) expressions, and P, P_1, P_2, \dots to range over individual process expressions. We use \mathcal{V} to denote an enumerable set of names, and X, X_1, X_2, \dots to range over elements of \mathcal{V} . In MMC, names are represented by Prolog variables. We use \mathcal{PN} to denote the enumerable set of process (agent) names, and p, p_1, p_2, \dots to range over process names. In MMC, process names are represented by Prolog function (i.e. data constructor) symbols. Finally, \mathcal{D} is used to denote the set of process definitions. Process expressions and process definitions in the monadic π -calculus are encoded in MMC using the language described by the following grammar.

$$\begin{aligned}
\mathcal{A} &::= \text{in}(\mathcal{V}, \mathcal{V}) \mid \text{out}(\mathcal{V}, \mathcal{V}) \mid \text{outbound}(\mathcal{V}, \mathcal{V}) \mid \text{tau} \\
\mathcal{P} &::= \text{zero} \mid \text{pref}(\mathcal{A}, \mathcal{P}) \mid \text{nu}(\mathcal{V}, \mathcal{P}) \mid \text{par}(\mathcal{P}, \mathcal{P}) \mid \text{choice}(\mathcal{P}, \mathcal{P}) \\
&\quad \mid \text{match}(\mathcal{V}=\mathcal{V}, \mathcal{P}) \mid \text{proc}(\mathcal{PN}(\vec{\mathcal{V}})) \\
\mathcal{D} &::= \text{def}(\mathcal{PN}(\vec{\mathcal{V}}), \mathcal{P})
\end{aligned}$$

Actions `in`, `out`, `outbound` and `tau` represent input, output, bound output and internal actions respectively. Among process expressions, `zero` is the process with no transitions; `pref(A, P)` is the process obtained by prefixing action A to P ; `nu(X, P)` is the process obtained from P by restricting the name X ; `match(X1=X2, P)` is the process that behaves as P if the names X_1 and X_2 match, and as `zero` otherwise. The operators `choice` and `par` represent non-deterministic choice and parallel composition respectively. The expression `proc(p(\vec{X}))` represents a *process invocation* where p is a process name (having a corresponding definition) and \vec{X} is a comma-separated list of names that are the actual parameters of the invocation. Process invocation may be used to define recursive processes. Each process definition of the form `def(p(\vec{X}), P)` associates a process name p and a list of formal parameters \vec{X} with process expression P .

It is easy to see that MMC's syntax simply encodes the standard syntax of π -calculus expressions used in [?]. This observation is formalized below.

Definition 1 *Given a one-to-one function θ that maps names of Prolog variables to names in π -calculus expressions, the function f_θ mapping process expressions in MMC's syntax to standard π -calculus syntax is defined as follows:*

$$\begin{aligned}
f_\theta(\mathbf{zero}) &= 0 & f_\theta(X) &= \theta(X) \\
f_\theta(\mathbf{match}(X_1=X_2, P)) &= [\theta(X_1) = \theta(X_2)]f_\theta(P) & f_\theta(\mathbf{pref}(\mathbf{tau}, P)) &= \tau.f_\theta(P) \\
f_\theta(\mathbf{pref}(\mathbf{in}(X_1, X_2), P)) &= \theta(X_1)\theta(X_2).f_\theta P & f_\theta(\mathbf{nu}(X, P)) &= (\nu \theta(X))f_\theta(P) \\
f_\theta(\mathbf{pref}(\mathbf{out}(X_1, X_2), P)) &= \overline{\theta(X_1)}\theta(X_2).f_\theta P & f_\theta(\mathbf{choice}(P, Q)) &= f_\theta(P) + f_\theta(Q) \\
f_\theta(\mathbf{proc}(p(X_1, \dots, X_n))) &= p(\theta(X_1), \dots, \theta(X_n)) & f_\theta(\mathbf{par}(P, Q)) &= f_\theta(P) \mid f_\theta(Q) \\
f_\theta(\mathbf{def}(p(X_1, \dots, X_n), P)) &= p(\theta(X_1), \dots, \theta(X_n)) \stackrel{\text{def}}{=} f_\theta(P)
\end{aligned}$$

Definition 1 allows us to directly import the notions of bound and free names from π -calculus to our encoding. In actions of the form `in(X, Y)`, `out(X, Y)`, and `outbound(X, Y)`, the name X is said to be *free*. The name Y in the `out` action is also *free* while the name Y in `in` and `outbound` actions is said to be *bound*. Table 1 lists the free and bound names of process expressions (second and third column, respectively).

Note that the same name may occur both bound and free in a process expression. For instance, consider process expression `pref(out(Y, X), nu(X, pref(out(Y, X), nu(X, pref(out(Y, X), zero))))))`. The name X occurs both free (in the first `out`) and bound, and there are two distinct bound occurrences of X . Our encoding of the model checker becomes considerably simpler if we ensure that bound names are all distinct from each other and from free names. We call process expressions having this distinct-name property as *valid*. The formal definition of validity is achieved by associating with each process expression P a set of *uniquely bound names* (denoted by $ubn(P)$), as defined in the fourth column of Table 1.

Definition 2 (Validity) *A process expression P is valid if and only if $fn(P) \cap bn(P) = \emptyset$ and $ubn(P) = bn(P)$. A process definition of the form `def(p(\vec{X}), P)` is valid if and only if P is valid and $bn(P) \cap \vec{X} = \emptyset$, i.e. formal parameters do not appear bound in P .*

| Process Expression P | Free Names $fn(P)$ | Bound Names $bn(P)$ | Uniquely Bound Names $ubn(P)$ |
|---|-----------------------------|------------------------|--|
| $\text{pref}(\tau, P_1)$ | $fn(P_1)$ | $bn(P_1)$ | $ubn(P_1)$ |
| $\text{pref}(\text{in}(X_1, X_2), P_1)$ | $(fn(P_1) \cup \{X_1\})$ | $bn(P_1) \cup \{X_2\}$ | $(ubn(P_1) \cup \{X_2\})$ |
| $\text{pref}(\text{outbound}(X_1, X_2), P_1)$ | $-\{X_2\}$ | $-\{X_2\}$ | $-(bn(P_1) \cap \{X_2\})$ |
| $\text{pref}(\text{out}(X_1, X_2), P_1)$ | $fn(P_1) \cup \{X_1, X_2\}$ | $bn(P_1)$ | $ubn(P_1)$ |
| $\text{match}((X_1 = X_2), P_1)$ | $fn(P_1) \cup \{X_1, X_2\}$ | $bn(P_1)$ | $ubn(P_1)$ |
| $\text{par}(P_1, P_2)$ | $fn(P_1) \cup fn(P_2)$ | $bn(P_1) \cup bn(P_2)$ | $(ubn(P_1) \cup ubn(P_2))$ |
| $\text{choice}(P_1, P_2)$ | | | $-(bn(P_1) \cap bn(P_2))$ |
| $\text{nu}(X, P_1)$ | $fn(P_1) - \{X\}$ | $bn(P_1) \cup \{X\}$ | $(ubn(P_1) \cup \{X\}) - (bn(P_1) \cap \{X\})$ |

Table 1. Free, bound, and uniquely bound names of processes

The following property can be established based on the definition of validity:

Proposition 1 *Every subexpression of a valid process expression is also valid.*

We say that a process expression P is *closed* if and only if $fn(P) = \emptyset$. We say that a process definition of the form $\text{def}(p(\vec{X}), P)$ is *closed* if and only if all free names in P occur in \vec{X} , i.e. $fn(P) \subseteq \vec{X}$. The encoding of the model checker described in this paper requires that all process definitions are valid and closed. Note that restricting our attention to valid definitions does not reduce expressiveness since any process expression can be converted to an equivalent valid expression by suitably renaming the bound names.

2.2 Operational Semantics of MMC

The operational semantics of π -calculus is traditionally given in terms of a symbolic transition system [?,?]. The transition relation of such a system can be derived from process definitions in MMC using the relation **trans** defined by the rules in Figure 1. The relation **trans** can be seen as a direct encoding of the symbolic semantics of [?]. At a high level, a tuple in the **trans** relation of the form $\text{trans}(P_1, A, M, P_2, Nin, Nout)$ means that process expression P_1 can evolve into process expression P_2 after an A action provided equality constraints over the names in M hold. (Nin and $Nout$ are integers used to generate new names, an implementation detail explained later.) A conjunction of equality constraints is encoded as a list in Prolog, each element in the list encoding an equality constraint over a pair of names. Each tuple in the **trans** relation corresponds to a transition in the symbolic semantics of [?] of the form $f_\theta(P_1) \xrightarrow{fc_\theta(M), f_\theta(A)} f_\theta(P_2)$, where fc_θ is defined as follows:

Definition 3 *The following function fc_θ maps the Prolog representation of equality constraints over names to equivalent constraints over π -calculus names:*

$$\begin{aligned}
fc_\theta([\] &= true \\
fc_\theta([X_1=X_2]) &= \theta(X_1) = \theta(X_2) \\
fc_\theta(\text{append}(M_1, M_2)) &= fc_\theta(M_1) \wedge fc_\theta(M_2)
\end{aligned}$$

When the transitions are generated, we avoid name capture by binding each distinct instance of a restricted name to a freshly generated name drawn from

```

% Pref
trans(pref(A, P), A, [], P, Nin, Nout).

% Sum
trans(choice(P, Q), A, M, P1, Nin, Nout) :- trans(P, A, M, P1, Nin, Nout).
trans(choice(P, Q), A, M, Q1, Nin, Nout) :- trans(Q, A, M, Q1, Nin, Nout).

% Id
trans(proc(PN), A, M, Q, Nin, Nout) :-
    def(PN, P), trans(P, A, M, Q, Nin, Nout).

% Match
trans(match((X=Y), P), A, ML, P1, Nin, Nout) :-
    X==Y -> trans(P, A, ML, P1, Nin, Nout)
    ; trans(P, A, M, P1, Nin, Nout),
      append([X=Y], M, ML).

% Par
trans(par(P, Q), A, M, par(P1, Q), Nin, Nout) :- trans(P, A, M, P1, Nin, Nout).
trans(par(P, Q), A, M, par(P, Q1), Nin, Nout) :- trans(Q, A, M, Q1, Nin, Nout).

% Com
trans(par(P, Q), tau, MNL, par(P1, Q1), Nin, Nout) :-
    trans(P, A, M, P1, Nin, Nout1),
    trans(Q, B, N, Q1, Nout1, Nout),
    complement(A, B, L),
    append(M, N, MN),
    append(MN, L, MNL).

% Res
trans(nu(Y, P), A, M, nu(Y, P1), Nin, Nout) :-
    gen_new_name(Y, Nin, Nout1),
    trans(P, A, M, P1, Nout1, Nout),
    not_in_action(Y, A),           % Y does not appear in action A
    not_in_constraint(Y, M).      % Y does not appear in constraint M

% Open
trans(nu(Y, P), outbound(X, Z), M, P1, Nin, Nout) :-
    gen_new_name(Y, Nin, Nout1),
    trans(P, out(X, Z), M, P1, Nout1, Nout),
    Y == Z, Y\==X,
    not_in_constraint(Y, M).

% Close
trans(par(P, Q), tau, MNL, nu(W, par(P1, Q1)), Nin, Nout) :-
    trans(P, A, M, P1, Nin, Nout1),
    trans(Q, B, M, Q1, Nout1, Nout),
    comp_bound(A, B, W, L),
    append(M, N, MN), append(MN, L, MNL).

gen_new_name(Y, Nin, Nout):-
    (var(Y) -> Nout is Nin + 1, Y = name(Nout)
    ; Nout = Nin).

complement(in(X, V), out(Y, V), L) :- X==Y -> L=[] ; L=[X=Y].
complement(out(X, V), in(Y, V), L) :- X==Y -> L=[] ; L=[X=Y].

comp_bound(outbound(X, W), in(Y, W), W, L):- X==Y -> L=[] ; L=[X=Y].
comp_bound(in(X, W), outbound(Y, W), W, L):- X==Y -> L=[] ; L=[X=Y].

```

Fig. 1. Encoding of π -calculus transition semantics.

an enumerable set indexed by an integer. These fresh names are represented by terms of the form $\mathbf{name}(N)$. Fields N_{in} and N_{out} in the \mathbf{trans} relation are used to maintain the index of the set of fresh names: N_{in} records the name with the largest index generated before a transition is generated; and N_{out} records the same after the transition is generated.

It can be readily seen that the \mathbf{trans} relation preserves validity, formalized by the following proposition:

Proposition 2 *Let \mathcal{C} denote the set of Prolog constants, P_1 be a valid process expression, and N_1 be larger than any N in $\mathbf{name}(N)$ occurring in P_1 . For any one-to-one function $\sigma : \mathbf{fn}(P_1) \rightarrow \mathcal{C}$, if $\mathbf{trans}(P_1\sigma, A, M, P_2, N_1, N_2)$ is an answer to the query $\mathbf{trans}(P_1\sigma, A, M, ?, N_1, ?)$ then P_2 is also a valid process expression.*

Finally, by induction on the lengths of derivations, we can show that the transition relation computed using our encoding of Figure 1 is correct with respect to the symbolic transition semantics of [?].

Theorem 3 *Let S be the logic program encoding the symbolic semantics (given in Figure 1), and D be a set of process definitions. Let P_1 be a valid MMC process expression, N_1 be the largest N in $\mathbf{name}(N)$ occurring in P_1 , and $\sigma : \mathbf{fn}(P) \rightarrow \mathcal{C}$ be a one-to-one function. Then $\mathbf{trans}(P_1\sigma, A, M, P_2, N_1, N_2)$ is an answer derivable from the logic program $D \cup S$ if and only if $f_\theta(P_1) \xrightarrow{fc_\theta(M\sigma^{-1}), f_\theta(A\sigma^{-1})} f_\theta(P_2\sigma^{-1})$ is a derivation in the symbolic semantics for the π -calculus.*

The use of structural congruence: While the encoding of the operational semantics is complete, it is not yet sufficient to build a model checker: the semantics distinguishes between process expressions based on their syntax, even if their behaviors are identical. For example, consider the following process definitions:

```
def(ser(Pc), nu(X, pref(out(Pc, X), proc(ser(Pc)))))
def(cli(Pc), pref(in(Pc, X), proc(cli(Pc)))).
def(system, nu(Pc, par(proc(ser(Pc)), proc(cli(Pc)))).
```

Process **system** consists of processes **ser(Pc)** and **cli(Pc)**. Process **ser(Pc)** repeatedly generates a new private name X and sends X to process **cli(Pc)**. Since each time this happens the name X is different from any other names previously generated during the computation, the state space of **system** is infinite as shown below:

```
system = nu(name(0), par(proc(ser(name(0))), proc(cli(name(0)))))
  \xrightarrow{\tau} nu(name(0), nu(name(1), par(proc(ser(name(0))), proc(cli(name(0)))))
  \xrightarrow{\tau} nu(name(0), nu(name(1), nu(name(2), par(proc(ser(name(0))), proc(cli(name(0)))))
  \xrightarrow{\tau} \dots
```

However, in $\mathbf{nu}(\mathbf{name}(1), \mathbf{par}(\mathbf{proc}(\mathbf{ser}(\mathbf{name}(0))), \mathbf{proc}(\mathbf{cli}(\mathbf{name}(0)))))$, $\mathbf{name}(1)$ does not occur in $\mathbf{par}(\mathbf{proc}(\mathbf{ser}(\mathbf{name}(0))), \mathbf{proc}(\mathbf{cli}(\mathbf{name}(0)))))$, and hence the behavior of these two process expressions is identical. This can be formalized as the following structural-congruence rule:

$$\mathbf{nu}(X, P) \equiv P \quad \text{if } X \text{ does not occur in } P.$$

After applying this rule to modify the clauses for `Res` and `Close`, which handle restricted names, the process `system` exhibits finite behavior.

Using resolution mechanism to generate new names: Note that we have used a global counter (implemented using `Nin` and `Nout`) to generate new names when applying a restriction operator. However, using constants to generate new names in Prolog results in redundant states and transitions. For example, the two process terms `nu(name(0), pref(out(X, name(0)), zero))` and `nu(name(1), pref(out(X, name(1)), zero))` appear different, although they differ only in the bound names: `name(0)` and `name(1)`. We can exploit the fact that *variant checks*— i.e. checking if two terms are identical modulo names of variables— can be inexpensively performed in the XSB tabled logic programming system on which MMC is implemented. Instead of generating integers to index new names, we use existential logical variables in terms, and let the resolution mechanism generate a new variable every time that clause is used. This is done by discarding the arguments implementing the counter, i.e., `Nin` and `Nout`, and defining `gen_new_name` as the fact: `gen_new_name(name(_))`.

Using this mechanism, we generate `name(V0)` in the place of `name(0)`, and `name(V1)` in the place of `name(1)`. The two terms, `nu(name(V0), pref(out(X, name(V1)), zero))` and `nu(name(V1), pref(out(X, name(V1)), zero))`, now become variants of each other and MMC will treat them as the same state.

A consequence of this representation is that the equality of two names can no longer be checked by unification (`=`), but by the identity operator (`==`). For instance while the unification `name(V0)=name(V1)` will succeed after unifying `V0` and `V1`, the identity check `name(V0)==name(V1)` will fail unless `V0` and `V1` are already unified.

Another consequence of this representation is that the goal-reordering optimization, which was employed in early versions of XMC, can no longer be directly applied to our encoding. Consider the `Com` rule. In general, the number of solutions of `complement(A, B, L)` is much smaller than that of `trans(Q, B, N, Q1)` (note that the above optimization has discarded fields `Nout1` and `Nout`). Thus, by reordering `trans(Q, B, N, Q1)` and `complement(A, B, L)`, we will compute fewer intermediate answers. This optimization can be applied using the earlier (integer) representation of names and can result in significant performance gains. However, the representation of restricted names using variables means that the program is dependent on the order in which variables are bound, and hence this optimization is not directly applicable. Hence we specialize the `trans` rules to two versions, the first of which is used when the action (the second argument) is known; and the second of which is used when the action is unknown. The specialization lets us change the join order appropriately without affecting the correctness.

From Monadic to Polyadic π -Calculus: The polyadic version of the π -calculus is supported in MMC by extending the above construction as follows. The syntax is extended by introducing a set \mathcal{F} of tuple constructors (n -ary function symbols

for $n \geq 0$) and considering the set of terms \mathcal{T} built from \mathcal{F} and \mathcal{V} . The grammar given in Section 2.1 becomes (only the changed rules are shown):

$$\begin{aligned} \mathcal{A} &::= \text{in}(\mathcal{V}, \mathcal{T}) \mid \text{out}(\mathcal{V}, \mathcal{T}) \mid \text{outbound}(\mathcal{V}, \vec{\mathcal{V}}, \mathcal{T}) \\ \mathcal{P} &::= \text{unify}((\mathcal{V} = \mathcal{T}), P) \mid \text{proc}(\mathcal{PN}(\vec{\mathcal{T}})) \end{aligned}$$

In essence, the communication actions can now be used to place names in (or extract names from) tuples and other data structures, and process invocations may contain such data structures. Note that when multiple names can be sent in a single message (e.g. when sending a tuple of names), the bound output action needs to keep track of the set of bound names (the second parameter) in the message (the third parameter). The **Open** and **Close** rules in the transition semantics for the polyadic version change correspondingly. Furthermore, we introduce an operator **unify** to decompose a term into subterms by pattern matching. The names in T are bound names in an expression of the form $\text{unify}((X = T), P)$. An expression $\text{unify}((X = T), P)$ behaves as P when the names in T are bound to terms over \mathcal{F} and \mathcal{V} such that X and T unify, and as **zero** if such a unifier does not exist. The modified rules can be directly encoded in Prolog as before; see [?] for details.

2.3 Model Checking in the π - μ -Calculus

A modal logic for the monadic π -calculus, the π - μ -calculus, was originally proposed in [?] and extended to the polyadic π -calculus in [?] and [?]. The π - μ -calculus has variants of the traditional box and diamond modal operators to reflect the early and late semantics of the π -calculus.

Below, we present an encoding of a model checker for an expressive subset of the π - μ -calculus which does not have explicit quantifiers (\exists and \forall). We use the following syntax to represent formulas in our subset of the π - μ -calculus. We use \mathcal{F} to denote the set of (non fixed-point) formulas; \mathcal{A} and \mathcal{V} (from Section 2.1) to denote sets of actions and names; \mathcal{Z} to denote formula variables in the π - μ -calculus; and \mathcal{E} to denote fixed-point equations defining the formula variables.

$$\begin{aligned} \mathcal{F} &::= \text{tt} \mid \text{ff} \mid \text{pred}((\mathcal{V}, \mathcal{V}), \mathcal{F}) \mid \text{and}(\mathcal{F}, \mathcal{F}) \mid \text{or}(\mathcal{F}, \mathcal{F}) \mid \text{diam}(\mathcal{A}, \mathcal{F}) \\ &\quad \mid \text{box}(\mathcal{A}, \mathcal{F}) \mid \text{form}(\mathcal{Z}(\vec{\mathcal{V}})) \\ \mathcal{E} &::= \text{fdef}(\mathcal{Z}(\vec{\mathcal{V}}), \text{lfp}(\mathcal{F})) \mid \text{fdef}(\mathcal{Z}(\vec{\mathcal{V}}), \text{gfp}(\mathcal{F})) \end{aligned}$$

And and or are boolean connectives; **diam** and **box** are model operators; **lfp** and **gfp** represent least and greatest fixed point operators respectively; and **pred** is used to encode a match operation. Names in a formula definition are implicitly quantified; the quantifiers are determined as follows. Names appearing on the left hand side of a definition are called formal parameters, and the remaining names in a definition are called *local* names. For a local name X , let φ be a largest subformula of the right hand side such that $\varphi = \text{diam}(A, F)$ ($\varphi = \text{box}(A, F)$) and X occurs in A . Then X is existentially (universally) quantified, with its scope covering φ . We require that every local name in a formula be quantified in the above manner. Model checking π - μ -calculus where quanti-

$$\begin{array}{l}
\text{Id} \quad \frac{}{P \vdash_{\theta} \mathbf{tt}} \quad \theta \text{ is consistent} \\
\text{Pred} \quad \frac{P \vdash_{\theta'} F}{P \vdash_{\theta} \mathbf{pred}((X=Y), F)} \quad \theta' = \theta \cup \mathit{mgu}(X, Y) \\
\text{And} \quad \frac{P \vdash_{\theta} F_1 \quad P \vdash_{\theta} F_2}{P \vdash_{\theta} \mathbf{and}(F_1, F_2)} \\
\text{Not} \quad \frac{P \not\vdash_{\theta} F}{P \vdash_{\theta} \mathbf{not}(F)} \\
\text{Or} \quad \frac{P \vdash_{\theta} F_1}{P \vdash_{\theta} \mathbf{or}(F_1, F_2)} \quad \frac{P \vdash_{\theta} F_2}{P \vdash_{\theta} \mathbf{or}(F_1, F_2)} \\
\text{Diam} \quad \frac{P_1 \vdash_{\theta'} F}{P \vdash_{\theta} \mathbf{diam}(A, F)} \quad \mathbf{trans}(P, A', -, P_1), \quad \theta' = \theta \cup \mathit{mgu}(A, A') \\
\text{Box} \quad \frac{P_1 \vdash_{\theta_1} F, \dots, P_n \vdash_{\theta_n} F}{P \vdash_{\theta} \mathbf{box}(A, F)} \quad \{(P_1, \theta_1), \dots, (P_n, \theta_n)\} = \{(P', \mathit{mgu}(A, A')) \mid \mathbf{trans}(P, A', -, P')\} \\
\text{Lfp} \quad \frac{P \vdash_{\theta} F[\vec{V}'/\vec{V}]}{P \vdash_{\theta} \mathbf{form}(Z(\vec{V}'))} \quad \mathbf{fdef}(Z(\vec{V}), \mathbf{lfp}(F))
\end{array}$$

Fig. 2. The tableau rules of the subset of π - μ -calculus

fiers are restricted as described above requires the ability to handle inequality constraints (e.g. $X \neq Y$). Inequality constraints arise even when the logic and the process specification uses only equalities, for instance, to record the substitutions under which a transition is not enabled. Equality constraints are handled by a logic programming system. In contrast, inequality constraints have to be explicitly treated: either representing them symbolically, or enumerating their consequences (i.e., $X \neq Y$ interpreted over a domain $\{a, b, c\}$ for X and Y can be enumerated as $X = a, Y = b, X = a, Y = c, \dots$). While enumeration leads to poor performance, symbolic representation adds an additional layer of implementation (i.e. a constraint solver) with its attendant overheads. We avoid this overhead by imposing a condition that the set of constraints (the constraint store) needed while model checking is either empty or consist only of constraints over restricted names.

The semantics of this subset of π - μ -calculus can be readily derived from the semantics of the full logic given in [?]. From this semantics, we can also derive a tableau proof system for our subset given in Figure 2.3. The tableau can be shown to be sound and complete with respect to the semantics of π - μ -calculus provided all free names in the process expression and formula in original model-checking goal $P \vdash F$ are distinct. The tableau treats only least fixed point formulas but handles negation; greatest fixed point formulas are handled using their dual least fixed point forms (i.e. using the identity $\nu Z.F \equiv \neg \mu Z. \neg F[\neg Z/Z]$). The parameter θ in the tableau keeps track of the current substitution of names in the formula and names in the process expressions. A substitution θ is said to be consistent if for any name X , $X = t_1 \in \theta$ and $X = t_2 \in \theta$ then $t_1 = t_2$. In the figure, $\mathit{mgu}(t_1, t_2)$ denotes the most general unifier of the terms t_1 and t_2 , where both terms denote actions. The logic programming encoding of the tableau system is given in Figure 2.3 which can be directly executed on the XSB

```

Id      models(_P, tt).
Match  models(P, pred((X=Y), F)) :- X=Y, models(P, F).
And    models(P, and(F_1, F_2)) :- models(P, F_1), models(P, F_2).
Or     models(P, or(F_1, F_2))  :- models(P, F_1) ; models(P, F_2).
<A>   models(P, diam(A, F))     :- trans(P, A, _M, P1), models(P1, F).
[A]   models(P, box(A, F))      :- forall(P1, trans(P, A, _M, P1), models(P1, F)).
Neg    models(P, not(F))        :- sk_not(models(P, F)).
Lfp    models(P, form(Z))       :- fdef(Z, lfp(F)), models(P, F).

```

Fig. 3. Encoding of MMC's π - μ -calculus model checker

system. In the program, `sk_not(Goal)` refers to the negation of `Goal` which treats all variables in the term `Goal` as existentially quantified.

The soundness and completeness of the tableau system can be proved following [?]. The following theorem states the correctness of the model checker.

Theorem 4 *Let D be a set of process and formula definitions, S be the program consisting of the clauses in Figures 1 and 2.3, P be a valid process expression, and F be a formula containing only processes and formula variables defined in D . Let σ be a one-to-one function mapping free variables in P and F to Prolog constants. Then `models($P\sigma$, $F\sigma$)` is an answer derivable from the logic program $D \cup S$ if and only if $P \vdash F$ is a derivation in the tableau shown in Figure 2.3.*

The `models` predicate in MMC is an optimized version of the one shown in Figure 2.3 aimed at reducing the number of goals that will be tabled in XSB. The optimization is routine and is not shown.

3 Encoding the spi-calculus

The *spi-calculus* is an extension of the π -calculus with primitives for encryption and decryption to facilitate specification of cryptographic protocols [?]. Below we show that spi-calculus process expressions can be encoded in MMC using its support for the polyadic π -calculus. To express message encryption and decryption, and to represent structured messages composed of multiple segments, we use terms built from names and the two binary function symbols `encrypt` and `msg`. The encryption and decryption primitives of the spi-calculus are encoded in MMC as follows.

Encryption of a message M with a symmetric key K , denoted in the spi-calculus as $\{M\}_K$, is encoded in MMC by the term `encrypt(M , K)`. This term can be passed as a parameter or can appear as data on an output action. Decryption is specified in the spi-calculus using a *case* expression. For example, *case* L of $\{x\}_K$ in P behaves as $P[M/x]$ if L is of the form $\{M\}_K$, and as a deadlocked process otherwise. In MMC, we use `unify` to look into message components and `match` to verify whether the encryption and decryption keys match. For instance, the above *case* expression is encoded in MMC by the expression `unify($L = \text{encrypt}(X, E)$, $\text{match}(E = K, P)$)`. In MMC's encoding, `unify` extracts the key portion of the message and `match` checks if the given key K matches the encryption key E .

For handling specifications that use asymmetric public/private keys, we introduce two unary function symbols `priv` and `pub`, and use `priv(K)` and `pub(K)`

to denote the private and public keys of a key pair K . We also introduce a new process expression `code(Oper, P)` where *Oper* represents the operations written as Prolog predicate; `code(Oper, P)` performs *Oper* and then behaves as *P*. We use operation `complement(K1, K2)` to map the public key of a key pair to the corresponding private key and vice versa. For instance, `complement(pub(KA), K')` binds K' to `priv(KA)`. Using the above representation, a message M encrypted by principal A with public key K_A^{pu} is encoded as the term `t = encrypt(M, pub(KA))`. For instance, a principal B attempting to decrypt the term t with a key K will use the expression `unify(t = encrypt(X, K), code(complement(K, K1), match(K1=priv(KB), ...)))` which will deadlock unless K is same as `priv(KB)`. Similarly, a message M encrypted with a private key K_A^{pr} is encoded in MMC as `encrypt(M, priv(KA))`.

Note that, as in the spi-calculus, the restriction operator `nu` can be used to generate fresh nonces and shared keys.

Systems with an intruder are modeled so that all communication between principals go through the intruder. We assume that there is only one intruder, and that the behavior of the intruder can be specified by a recursive process definition. When an intruder receives a message from a principal, it chooses to either transmit, intercept or fake the message transmission. The capabilities of the intruder to store and retrieve messages are encoded using a set data structure, and operations `store(S, t, S')` and `retrieve(S, t)`, where S and S' are sets and t is a term. An intruder's ability to decompose or compose messages can be encoded using `unify` and requires no extensions. Details of the encoding of example protocols and the extensions made to MMC appear in [?].

Security properties such as authenticity can be expressed in our subset of the π - μ -calculus. For verifying authenticity properties, we use two `out` actions on distinguished, global channels (called `send` and `commit` below) for each pair of principals in the protocol. For instance, when principal A initiates communication with B , it does an `out` action on channel `send_AB`; similarly, when A thinks it is communicating with B , it does an `out` action on channel `commit_AB`. Authenticity is violated if a principal commits to a communication without a corresponding (preceding) initiation.

Using MMC, we can detect the violation of authenticity in the Needham-Schroeder protocol originally found by [?]. We have also verified authenticity properties of the Yahalom protocol, and the modified Needham-Schroeder protocol (see [?]).

4 Experimental Results

We implemented the MMC model checker starting from the encoding of the `trans` and `models` relations given in Section 2.3. We then applied a number of logic-programming optimizations to this encoding, including goal reordering, clause resolution factoring, and the use of resolution to generate new names. The experimental results presented in this section reflect the performance of this optimized version of MMC, and were obtained on a 1GHz Pentium III machine with 256MB memory running Linux 7.0 and XSB v2.4.

| Protocol | States | Trans | Formula | Time (sec) | Mem (MB) |
|-------------------|--------|--------|-----------|------------|----------|
| Handover | 137 | 220 | deadlock | 0.12 | 0.93 |
| | | | lost data | 0.40 | 2.68 |
| Needham-Schroeder | 59 | 101 | deadlock | 0.14 | 0.70 |
| | | | attack | 0.03 | 0.39 |
| Yahalom | 29133 | 107652 | attack | 0.31 | 1.07 |

Table 2. Performance results of MMC on Handover, Needham-Schroeder, and Yahalom protocols.

| Benchmark | States | Trans | Property | Time(sec) | |
|-----------|--------|-------|-----------|-----------|-------|
| | | | | XMC | MMC |
| rether | 593 | 697 | deadlock | 0.47 | 0.57 |
| sieve(3) | 615 | 1423 | ae_finish | 0.73 | 1.53 |
| sieve(5) | 4023 | 16091 | ae_finish | 10.12 | 16.67 |
| leader(3) | 67 | 88 | ae_leader | 0.07 | 0.08 |
| leader(5) | 864 | 2687 | ae_leader | 2.00 | 2.32 |
| leader(7) | 11939 | 25632 | ae_leader | 45.83 | 63.12 |

Table 3. Comparative performance of XMC and MMC.

Table 2 illustrates MMC’s performance on three standard benchmarks: a simplified handover procedure from [?], and the Needham-Schroeder and Yahalom cryptographic protocols. Our specifications of Needham-Schroeder and Yahalom utilize MMC’s spi-calculus extensions, and contain a number of process expressions to which the restriction operator is applied. These expressions are candidates for the structural congruence rule, which ensures that MMC terminates for finite-control agents. Eliminating the application of this rule to expressions for which the scope of a restricted name does not contain recursion, led to a 3-fold improvement in model-checking execution times.

We also conducted experiments aimed at assessing both how MMC scales to large transition systems and how it compares in performance to the Mobility Workbench. In particular, for verifying the absence of deadlocks in chains of buffers of size 4, 8, and 12, MMC’s model checker takes 0.02s, 0.83s, and 25.46s, respectively. On the same formula, the MWB’s model checker for the polyadic π -calculus takes 0.58s for a buffer of size 4, but does not terminate within 13 hours for a buffer of size 8. The MWB also provides a built-in “deadlock” function that uses depth-first search to detect deadlocks instead of model checking the corresponding π - μ -calculus formula. MMC’s model checker outperforms MWB’s deadlock function for large chain lengths (e.g. for a buffer of size 12, the MWB’s deadlock function takes 139.43s); both systems show comparable performance for smaller chain lengths. The MWB also has a prototype implementation of a model prover [?] based on sequent calculus. However, at the time of this writing, the implementation appears to be in an unstable state, either looping on certain least fixed point formulas or terminating incorrectly (too early) on certain greatest fixed point formulas. Hence we were unable to get meaningful performance measurements for the MWB’s prover.

Finally, Table 3 compares the performance of MMC and the initial release of XMC on several examples from the XMC benchmark suite. (The initial version of XMC did not utilize the compiler for process expressions described in [?]). MMC is slightly slower than the first version of XMC, and this is to be expected given the non-mobile nature of these benchmarks. In particular, MMC spends time checking for structural congruence, despite the optimization discussed above. Also, the `Open` and `Close` clauses in the `trans` relation are never used, but MMC tries (and eventually fails) to resolve using these rules. Implementing a process-expression compiler for MMC along the lines of XMC’s compiler [?] will eliminate these overheads.

5 Conclusion

We presented MMC, a practical model checker for the π and spi-calculi. We are currently extending the functionality of MMC to include a symbolic bisimulation checker [?], and to handle the full π - μ -calculus, taking advantage of recent developments to add light-weight constraint processing to tabled logic programming [?].

Our results indicate that MMC's performance is comparable to that of the first versions of XMC. However, the compilation techniques incorporated into later versions of XMC have vastly improved its performance, reducing execution times by factors of 2 or more and reducing space needs by an order of magnitude [?]. A central feature of XMC's compiler is that it statically generates rules that cover all possible synchronizations between processes composed in parallel. XMC does not permit channel passing, rendering this kind of analysis possible. For the π -calculus, such static techniques appear to be infeasible. Nevertheless, we need to find mechanisms to reduce the cost of finding synchronizing transitions in order to derive model checkers for mobile processes that compete in performance with the current version of XMC. Another avenue of research is to augment MMC with program transformations (developed in [?] for combining induction-based proofs with model checking) to verify infinite families of mobile processes.

References

1. Mobility model checker for the π -calculus. Dept. of Computer Science, SUNY at Stony Brook, 2002. Available from <http://www.cs.sunysb.edu/~lmc/mmc>.
2. M. Abadi and B. Blanchet. Analyzing security protocols with secrecy types and logic programs. In *Proceedings of POPL'02*, pages 33–44, Jan. 2002.
3. M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. In *Fourth ACM Conference on CCS*, pages 36–47. ACM Press, 1997.
4. B. Aziz and G.W. Hamilton. A privacy analysis for the pi-calculus: The denotational approach. In *Proceedings of the 2nd Workshop on the Specification, Analysis and Validation for Emerging Technologies*, Copenhagen, Denmark, July 2002.
5. S. Basu, M. Mukund, C. R. Ramakrishnan, I. V. Ramakrishnan, and R. M. Verma. Local and symbolic bisimulation using tabled constraint logic programming. In *International Conference on Logic Programming*, pages 166–180, 2001.
6. F. B. Beste. The model prover - a sequent-calculus based modal μ -calculus model checker tool for finite control π -calculus agents. Technical report, Swedish Institute of Computer Science, 1998.
7. B. Blanchet. From secrecy to authenticity in security protocols. In *9th International Static Analysis Symposium*, pages 242–259, September 2002.
8. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 2001.
9. B. Cui and D. S. Warren. A system for tabled constraint logic programming. In *First International Conference on Computational Logic*, pages 478–492, 2000.
10. M. Dam. Proof systems for pi-calculus logics. *Logic for Concurrency and Synchronisation*, 2001.

11. G. Denker and J. Meseguer. Protocol specification and analysis in Maude. In *Proc. of Workshop on Formal Methods and Security Protocols*, June 1998.
12. Y. Dong and C.R. Ramakrishnan. An optimizing compiler for efficient model checking. In *Proceedings of FORTE/PSTV '99*, 1999.
13. F. Duran, S. Eker, P. Lincoln, and J. Meseguer. Principles of mobile maude. In *Proc. ASA/MA*, volume 1882, pages 73–85. Springer-Verlag, 2000.
14. T. Franzen. A theorem-proving approach to deciding properties of finite-control agents. Technical report, Swedish Institute of Computer Science, 1996.
15. A. Gordon and A.S.A. Jeffrey. Authenticity by typing for security protocols. In *IEEE Computer Security Foundations Workshop*, 2001.
16. H. Lin. Symbolic bisimulation and proof systems for the π -calculus. Technical report, School of Cognitive and Computer Science, U. of Sussex, UK, 1994.
17. G. Lowe. An attack on the Needham-Schroeder public-key authentication protocol. *Information Processing Letters*, pages 131–133, 1995.
18. G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. *Software Concepts and Tools*, 17:93–102, 1996.
19. C. Meadows. The NRL protocol analyzer: an overview. *Journal of Logic Programming*, 26(2):113–131, 1996.
20. R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989.
21. R. Milner. The polyadic π -calculus: a tutorial. *The Proceedings of the International Summer School on Logic and Algebra of Specification*, 1991.
22. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, Parts I and II. *Information and Computation*, 100(1):1–77, 1992.
23. R. Milner, J. Parrow, and D. Walker. Modal logics for mobile processes. *Theoretical Computer Science*, pages 149–171, 1993.
24. F. Orava and J. Parrow. An algebraic verification of a mobile network. *Formal Aspects of Computing*, 4:497–543, 1992.
25. J. Parrow. An introduction to the π -calculus. In Bergstra, Ponse, and Smolka, editors, *Handbook of Process Algebra*. Elsevier, 2001.
26. Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. W. Swift, and D. S. Warren. Efficient model checking using tabled resolution. In *Proceedings of CAV '97*, Haifa, Israel, July 1997.
27. A. Roychoudhury, K. Narayan Kumar, C.R. Ramakrishnan, I.V. Ramakrishnan, and S.A. Smolka. Verification of parameterized systems using logic-program transformations. In *Proceedings of TACAS 2000*, 2000.
28. C. Stirling and D. Walker. Local model checking in the modal mu-calculus. *Theoretical Computer Science*, pages 161–177, 1991.
29. P. Thati, K. Sen, and N. Marti-oliet. An executable specification of asynchronous pi-calculus semantics and may testing in Maude 2.0. In *Intl. Workshop on Rewriting Logic and its Applications*, 2002.
30. B. Victor. The mobility workbench user's guide. Technical report, Department of Computer Systems, Uppsala University, Sweden, 1995.
31. B. Victor and F. Moller. The mobility workbench — a tool for the π -calculus. In D. Dill, editor, *Proceedings of CAV'94*. Springer-Verlag, 1994.
32. XSB. The XSB logic programming system v2.4, 2001. Available from <http://xsb.sourceforge.net>.