A Provably Correct Compiler for Efficient Model Checking of Mobile Processes*

Ping Yang[1], Yifei Dong[2], C.R. Ramakrishnan[1], and Scott A. Smolka[1]

[1] Dept. of Computer Science, Stony Brook Univ., Stony Brook, NY, 11794, USA
[2] School of Computer Science, Univ. of Oklahoma, Norman, OK, 73019, USA
E-mail: {pyang,cram,sas}@cs.sunysb.edu, dong@cs.ou.edu

Abstract. We present an optimizing compiler for the π-calculus that significantly improves the time and space performance of the MMC model checker. MMC exploits the similarity between the manner in which resolution techniques handle variables in a logic program and the manner in which the operational semantics of the π-calculus handles names by representing π-calculus names in MMC as Prolog *variables*, with distinct names represented by distinct variables. Given a π-calculus process P, our compiler for MMC produces an extremely compact representation of P's symbolic state space as a set of *transition rules.* It also uses AC unification to recognize states that are equivalent due to symmetry.

1 Introduction

The recent literature describes a number of efforts aimed at building practical tools for the verification of concurrent systems using Logic Programming (LP) technology; see e.g. [23, 20, 11, 7, 6]. The basic idea underlying these approaches is to pose the verification problem as one of query evaluation over (constraint) logic programs; once this has been accomplished, the minimal-model computation in LP can be used to compute fixed points over different domains.

Taking this idea one step further, in [26], we developed the MMC model checker for the π-calculus [17], a process algebra in which new channel names can be created dynamically, passed as values along other channels, and then used themselves for communication. This gives rise to great expressive power: many computational formalisms such as the λ-calculus can be smoothly translated into the π-calculus, and the π-calculus provides the semantic foundation for a number of concurrent and distributed programming languages (e.g. [19]). MMC also supports the spi-calculus [1], an extension of the π-calculus for the specification and verification of cryptographic protocols. The treatment of channel names in the π- and spi-calculi poses fundamental problems in the construction of a model checker, which are solved in MMC using techniques based on LP query-evaluation mechanisms as explained below.

* Research supported in part by NSF grants CCR-9876242, CCR-9988155, IIS-0072927, CCR-0205376, CCR-0311512, ARO grants DAAD190110003 and DAAD190110019, and ONR grant N000140110967.

MMC, which stands for the Mobility Model Checker, targets the finite-control subset of the π- and spi-calculi[1] so that model checking, the problem of determining whether a system specification S entails a temporal-logic formula φ, may proceed in a fully automatic, push-button fashion. It uses the alternation-free fragment of the π-μ-calculus, a highly expressive temporal logic, for the property specification language.

MMC is built upon the XSB logic programming system with tabulation [25] and, indeed, tabled resolution played a pivotal role in MMC's development. State-space generation in MMC is performed by querying a logic program, called `trans`, that directly and faithfully encodes π's symbolic operational semantics [15]. The key to this encoding is the similarity between the manner in which resolution techniques (which underlie the query-evaluation mechanism of XSB and other logic-programming systems) handle variables in a logic program and the manner in which the operational semantics of the π-calculus handles names. We exploit this similarity by representing π-calculus names in MMC as Prolog *variables*, with distinct names represented by distinct variables.

Query evaluation in MMC resorts to renaming (alpha conversion) whenever necessary to prevent name capture, and parameter passing is realized via unification. Variables are checked for identity (i.e. whether there is a substitution that can distinguish two variables) whenever names need to be equal, for instance, when processes synchronize. The management of names using resolution's variable-handling mechanisms makes MMC's performance acceptable.

Other than MMC, there have been few attempts to build a model checker for the π-calculus, despite its acceptance as a versatile and expressive modeling formalism. The Mobility Workbench (MWB) [24] is an early model checker and bisimulation checker for π; the implementation of its model checker, however, does not address performance issues. The PIPER system [3] generates CCS processes as "types" for π-calculus processes, and formulates the verification problem in terms of these process types; traditional model checkers can then be applied. This approach requires, however, user input in the form of type signatures and does not appear to be fully automated. In [13], a general type system for the π-calculus is proposed that can be used to verify properties such as deadlock-freedom and race-freedom. A procedure for translating a subset of the π-calculus into Promela, the system modeling language of the SPIN model checker [12], is given in [21]. Spin allows channel passing and new name generation, but may not terminate in some applications that require new name generation where MMC does; e.g. a handover protocol involving two mobile stations.

Problem Addressed and our Solution: Although MMC's performance is considerably better than that of the MWB, it is still an order of magnitude worse than that of traditional model checkers for non-mobile systems, such as SPIN and the XMC model checker for value-passing CCS [20]. XMC, like MMC, is built on top of the XSB logic-programming engine; despite its high-level implementation in Prolog, benchmarks show that it still exhibits competitive performance [8].

[1] The class of *finite-control* processes are those that do not admit parallel composition within the scope of recursion.

One reason for this is the development of a compiler for XMC that produces compact transition-system representations from CCS specifications [9].

In this paper, we present an optimizing compiler, developed along the lines of [9], for the π- and spi-calculi. Our compiler (Section 3), which uses LP technology and other algorithms developed in the declarative-languages community (such as AC unification), seeks to improve MMC's performance by compiling process expressions into a set of *transition rules*. These rules, which form a logic program, can then be queried by a model checker for generating actual transitions. In contrast to the compiler for value-passing CCS, a number of fundamental compilation issues arise when moving to a formalism where channels can be passed as messages, and communication links can be dynamically created and altered via a technique known as scope extrusion and intrusion. Our approach to dealing with these issues is as follows:

– *Representation of states:* The compiler uses a very compact representation for transition-system states, requiring only a symbol for the control location (i.e. a program counter value) and the valuations of variables that are free and live at that state. It also identifies certain semantically equivalent states that may have different syntactic representations. For instance, process expressions $(\nu x)((\nu y)p(x, y) \mid q(x))$ and $(\nu x)(\nu y)(p(x, y) \mid q(x))$ are considered distinct in [26]. According to the structural congruence rule of the π-calculus, however, these expressions are behaviorally identical and are given the same state representation by the compiler.
– *Determining the scope of names:* Since names can be dynamically created and communicated in the π-calculus, the scope of a name cannot in general be determined at compile time. The compiler therefore generates transition rules containing *tagged* names that allows the scope of a name to be determined at model-checking time, when transitions are generated from the rules.
– *State-space reduction via symmetry:* The compiler exploits the associativity and commutativity (AC) of the parallel composition operator when generating transition rules for the model checker (Section 4). In particular, transition rules may contain AC symbols and the compiler uses AC unification and indexing techniques to realize a form of symmetry reduction, sometimes leading to an exponential reduction in the size of the state space.

Another important aspect of MMC's compiler is that it is *provably correct*: The compiler is described using a syntax-directed notation, which when encoded as a logic program and evaluated using tabled resolution becomes its implementation. Thus the compilation scheme's correctness implies the correctness of the implementation itself. Given the complex nature of the compiler, the ability to obtain a direct, high-level, provably correct implementation is of significant importance, and is a practical illustration of the power of declarative programming.

Our benchmarking results (Section 5) reveal that the compiler significantly improves MMC's performance and scalability. For example, on a handover protocol involving two mobile stations, the original version of MMC runs out of memory while attempting to check for deadlock-freedom, even though 2GB of memory is available in the system. In contrast, MMC with compilation verifies this property in 47.01sec while consuming 276.29MB of memory. In another example, a webserver application, the AC operations supported by the compiler allow MMC to verify a system having more than 20 servers; MMC without compilation could handle only 6 servers. The MMC system with the compiler is available in full source-code form from `http://lmc.cs.sunysb.edu/~mmc`.

2 Preliminaries

MMC: A Model Checker for the π-Calculus. In MMC [26], π-calculus processes are encoded as Prolog terms. Let $\mathcal{A}$ denote the set of prefixes, $\mathcal{P}$ the set of process expressions, and $\mathcal{D}$ the set of process identifiers. Further, let $X, Y, Z \ldots$ range over Prolog variables and $p, q, r, \ldots$ range over process identifiers. The syntax of the MMC encoding of π-calculus processes is as follows:

$$\begin{aligned}
\mathcal{A} &::= \texttt{in}(X,Y) \ \mid \ \texttt{out}(X,Y) \ \mid \ \texttt{tau} \\
\mathcal{P} &::= \texttt{zero} \ \mid \ \texttt{pref}(\mathcal{A},\mathcal{P}) \ \mid \ \texttt{nu}(X,\mathcal{P}) \ \mid \ \texttt{par}(\mathcal{P},\mathcal{P}) \mid \ \texttt{choice}(\mathcal{P},\mathcal{P}) \\
&\quad\ \mid \ \texttt{match}(X\texttt{=}Y,\mathcal{P}) \ \mid \ \texttt{proc}(p(Y_1,Y_2,\ldots,Y_n)) \\
\mathcal{D} &::= \texttt{def}(p(X_1,X_2,\ldots,X_n),\mathcal{P}) \quad \text{where } X_i\text{'s are pairwise distinct}
\end{aligned}$$

Prefixes `in(`X`,`Y`)`, `out(`X`,`Y`)` and `tau` represent input, output and internal actions, respectively. `zero` is the process with no transitions while `pref(`α`,`P`)` is the process that can perform an α action and then behave as process P. `nu(`X`,`P`)` behaves as P and X cannot be used as a channel over which to communicate with the environment. Process `match(`X`=`Y`,`P`)` behaves as P if the names X and Y match, and as `zero` otherwise. The constructors `choice` and `par` represent non-deterministic choice and parallel composition, respectively. The expression `proc(`$p(Y_1, \ldots, Y_n)$`)` denotes a *process invocation* where p is a process name (having a corresponding definition) and $Y_1, \ldots, Y_n$ is a comma-separated list of names that are the actual parameters of the invocation. Each process definition of the form `def(`$p(X_1, \ldots, X_n)$`,` P`)` associates a process name p and a list of formal parameters $X_1, \ldots, X_n$ with process expression P. A formal definition of the correspondence between MMC's input language and the syntax of the π-calculus can be found in [26].

The standard notions of bound and free names (denoted by bn() and fn() respectively) in the π-calculus carry over to the MMC syntax. We use $n(e)$ to denote the set of *all* names in a process expression e. Names bound by a restriction operator in e are called *local* names of e.

In order to simplify the use of resolution procedures to handle names represented by variables, we use the following naming conventions. We say that a process expression is *valid* if all of its bound names are unique and are distinct from its free names. We say that a process definition of the form `def(`N`,`P`)` is valid if P, the process expression on the right-hand side of the definition, is valid. A process definition of the form `def(`N`,`P`)` is said to be *closed* if all free names of P appear in N (i.e. are parameters of the process). In MMC, we require all process definitions to be valid and closed. Note that this does not reduce expressiveness since any process expression can be converted to an equivalent valid expression by suitably renaming the bound names.

The operational semantics of the π-calculus is typically given in terms of a symbolic transition system [17, 15]. The MMC model checker computes symbolic transitions using the relation `trans(`s`,`a`,`c`,`d`)` where s and d represent the source and destination states of a transition, a the action and c a constraint on the names of s under which the transition is enabled. In the original model checker [26], this relation was computed by *interpreting* MMC process expressions: the `trans` relation was a direct encoding of the *constructive* semantics of π-calculus given in [26] which is equivalent to the symbolic semantics of [15].

In MMC, properties are specified using the alternation-free fragment of the π-μ-calculus [5], and the MMC model checker is encoded as the binary predicate `models(`P`,`F`)` which succeeds if and only if a process P satisfies a formula F. The encoding of the model checker is given in [26].

Logic Programs: We assume standard notions of predicate symbols, function symbols, and variables. Terms constructed from these such that predicate symbols appear at (and only at) the root are called *atoms*. The set of variables occurring in a term t is denoted by $vars(t)$; we sometimes use $vars(t_1, t_2, \ldots, t_n)$ to denote the set of all variables in terms t_1, t_2, …, t_n. A *substitution* is a map from variables to terms constructed from function symbols and variables. We use (possibly subscripted) θ, θ' to denote substitutions; the composition of two substitutions θ_1 and θ_2 is denoted by $\theta_1\theta_2$. A term t under substitution θ is denoted by $t\theta$. A *renaming* is a special case of substitution that defines a one-to-one mapping between variables.

Horn clauses are of the form $a_0 :- a_1, \ldots, a_n$ where the a_i are atoms. A *goal* (also called a query) is an atom. *Definite* logic programs are a set of Horn clauses. In this paper, we consider only definite logic programs, and henceforth drop the qualifier "definite". Top-down evaluation of logic programs based on one of several resolution mechanisms such as SLD, OLD, and SLG [16, 4], determines the substitution(s) under which the goal can be derived from the program clauses. We use $G \overset{\theta}{\Longrightarrow}_P \Box$ to denote the derivation of a goal G over a program P, where θ represents the substitution collected in that derivation.

3 A Compiler for the π-calculus

In this section, we present our compiler for the MMC model checker. Given a process expression E, it generates a set of *transition rules* from which E's transitions can be easily computed. The number of transition rules generated for an expression E is typically much smaller than the number of transitions in E's state space. More precisely, the number of transition rules generated for an expression E is polynomial in the size of E even in the worst case, while the number of transitions in E's state space may be exponential in the size of E.

The rules generated by the compiler for a given MMC process expression E define E's *symbolic transition system*, and are represented using a Prolog predicate of the form `trans(`s`,`a`,`c`,`d`)` where s and d are the transition's source and destination *states*, a is the *action* taken, and c is a *constraint* on the names of s under which the transition is enabled. Although the clauses of the definition of `trans` resemble facts, the constraints c that appear in them can be evaluated only at run-time, and hence encode rules.

The representation used for states is as follows. If E is a *sequential* process expression (i.e. does not contain a parallel composition operator) then it is represented by a Prolog term of the form $\mathtt{s}_i(\overline{V})$ where $\overline{V}$ are the free variables in E and $\mathtt{s}_i$ represents the control location (analogous to a program counter) of E. For instance, let E_1 be the MMC process expression `pref(in(X,Z),pref(out(Z,Y), zero))`. Names `X` and `Y` are free in E_1 and `Z` is bound in E_1. The symbolic state corresponding to E_1 is then given by a Prolog term of the form `s1(X,Y)`, where `s1` denotes the control state. Observe that E_1 can make an `in(X,Z)` action and become E_1', where E_1' =`pref(out(Z,Y), zero)`. The state corresponding to E_1' is a term of the form `s2(Z,Y)`. The symbolic transition from E_1 to E_1' becomes the clause `trans(s1(X,Y),in(X,Z),true,s2(Z,Y))`.

If E is a *parallel* expression of the form `par(`E_1`,`E_2`)` then it is represented by a term of the form `prod(`s_i`,`s_j`)` where s_i and s_j are the states corresponding to E_1 and E_2, respectively. For example, let E_2 =`pref(out(U,V),pref(in(V,W), zero))`, and let `s3(U,V)` and `s4(V)` be the states corresponding to E_2 and `pref(in(V,W), zero))`, respectively. Letting E_1 be defined as above, then the state corresponding to E is `prod(s1(X,Y),s3(U,V))`.

Observe that E can perform a `tau` action and become process E' = `par(pref(out(V,Y), zero),pref(in(V,W),zero))` whenever the free names `X` of E_1 and `U` of E_2 are the same. Such a transition can then be represented by a Horn clause or *rule* of the form:

```
trans(prod(s1(X,Y),s3(U,V)),tau,X=U,prod(s2(V,Y),s4(V)))
```

where the constraint `X=U` is the condition under which the transition is enabled.

Further observe that in E, subprocess E_1 is capable of an autonomous (non-synchronous) transition, taking E from `par(`E_1`,`E_2`)` to `par(`E_1'`,`E_2`)`. Such a transition can be represented by a rule of the form:

```
trans(prod(s1(X,Y),P),in(X,Z),true,prod(s2(Z,Y),P)).
```

where `P` is a variable that ranges over states; thus transition rules may specify a set of states using *state patterns* rather than simply individual states.

One of the challenges we encountered in developing a compiler for MMC concerned the handling of scope extrusion in the π-calculus. In MMC without compilation [26], local names can be determined when computing transitions and hence scope extrusion was implemented seamlessly using Prolog's unification mechanism. However, at compilation time, it may be impossible to determine whether a name is local. For instance, it is not clear if y is a local name in process $x(y).\overline{x}y$ before the process receives an input name. Intuitively, we can solve this problem by carrying local names explicitly in the states of the `trans` rule. This approach, however, introduces a significant amount of overhead when invoking the `trans` to compute the synchronization between two processes. Further, if we do not know for certain whether a name is local, we must also carry a constraint within the rule.

In order to handle scope extrusion efficiently, we propose the following solution. We present names in MMC in one of two forms: either as plain Prolog variables (as in the above examples), or as terms of the form `name(`Z`)` where

Z is a Prolog variable. Names of the latter kind are used to represent *local* names, generated by the restriction operator, whereas names of the former kind represent all others. This distinction is essential since we expand the scope of restricted names using the structural congruence rule $(\nu x)P|Q \equiv (\nu x)(P|Q)$ whenever $x \notin \text{fn}(Q)$; this expansion process lets us consolidate the pairs of rules Open and Prefix and Close and Com in the semantics of the π-calculus into single rules. This distinction also enables us to quickly check whether a name is restricted without explicitly keeping track of the environment.

3.1 Compilation Rules

Definition 1 (State Assignment) A *state-assignment function* σ maps process expressions to *positive* integers such that for any two valid expressions E and E', $\sigma(E) = \sigma(E')$ if and only if E and E' are variants of each other (i.e. identical modulo names of variables).

Definition 2 (State Generation) Given a state-assignment function σ, the *state generation function* $\Psi_\sigma(\ )$ maps a process expression E to a state as follows:

$$\Psi_\sigma(E) = \begin{cases} \texttt{state}_0 & \text{if } E = \texttt{zero} \\ \texttt{prod}(\Psi_\sigma(E_1), \Psi_\sigma(E_2)) & \text{if } E = \texttt{par}(E_1, E_2) \\ \Psi_\sigma(E_1[\texttt{name(}V\texttt{)}/X]) & \text{if } E = \texttt{nu}(X, E_1) \\ \ \text{where } V \notin \text{n}(E_1) & \\ \texttt{state}_{\sigma(E)}(\text{fn}(E)) & \text{otherwise} \end{cases}$$

For each process expression E, MMC's compiler recursively generates a set of transition rules; i.e., E's transition rules are produced based on the transition rules of its subexpressions. The following operation over sets of transition rules is used in defining the compilation procedure:

Definition 3 (Source state substitution) Given a set of transition rules R, a pair of states s and s', and a constraint C, the *source-state substitution* of R, denoted by $R_{[s \leftarrow s';C]}$, is the set of transition rules

$$\{\texttt{trans}(s', a, (c, C), d) | \texttt{trans}(s, a, c, d) \in R\},$$

i.e. the set of rules obtained by first selecting rules whose source states unify with s, replacing s by s' in the source state, and adding constraint C to the condition part of the selected rules. If C is empty (i.e. `true`) then we denote the substitution simply by $R_{[s \leftarrow s']}$.

The transition rules generated for a process can be viewed as an automaton, and source-state substitution can be viewed as an operation that replaces the start state of a given automaton with a new state.

Definition 4 (Compilation Function) Given a state-assignment function σ, the compilation function $[\![\cdot]\!]$ maps MMC process expressions to sets of transition rules such that for any process expression E, $[\![E]\!]$ is the smallest set that satisfies the equations of Figure 1.

Expression E	**Transition Rules** $[\![E]\!]$
`zero`	$\emptyset$
`pref`(α, E_1)	$[\![E_1]\!] \cup \{\texttt{trans}(\Psi_\sigma(E), \alpha, \texttt{true}, \Psi_\sigma(E_1))\}$
`choice`(E_1, E_2)	$[\![E_1]\!] \cup [\![E_2]\!] \cup [\![E_1]\!]_{[\Psi_\sigma(E_1) \leftarrow \Psi_\sigma(E)]} \cup [\![E_2]\!]_{[\Psi_\sigma(E_2) \leftarrow \Psi_\sigma(E)]}$
`nu`(X, E_1)	$[\![E_1[\texttt{name}(V)/X]]\!] \quad V \notin \mathrm{n}(E_1)$
`match`(C, E_1)	$[\![E_1]\!] \cup ([\![E_1]\!]_{[\Psi_\sigma(E_1) \leftarrow \Psi_\sigma(E);C]})$
`par`(E_1, E_2)	$\{\texttt{trans}(\texttt{prod}(s_1, V_2), a, c, \texttt{prod}(d_1, V_2)) \mid \texttt{trans}(s_1, a, c, d_1) \in [\![E_1]\!]\}$ $\cup \{\texttt{trans}(\texttt{prod}(V_1, s_2), a, c, \texttt{prod}(V_1, d_2)) \mid \texttt{trans}(s_2, a, c, d_2) \in [\![E_2]\!]\}$ $\cup \{\texttt{trans}(\texttt{prod}(s_1, s_2), \texttt{tau}, (c_1, c_2, c), \texttt{prod}(d_1, d_2)\theta) \mid \texttt{trans}(s_1, a_1, c_1, d_1) \in [\![E_1]\!] \wedge \texttt{trans}(s_2, a_2, c_2, d_2) \in [\![E_2]\!] \wedge \mathit{vars}(s_1, a_1, c_1, d_1) \cap \mathit{vars}(s_2, a_2, c_2, d_2) = \emptyset \wedge c = (u_1 == u_2) \wedge \theta = \mathit{mgu}(v_1, v_2)$ where $\{a_1, a_2\} = \{\texttt{in}(u_1, v_1), \texttt{out}(u_2, v_2)\} \wedge (c_1, c_2, c)$ is satisfiable$\}$
`proc`$(p(\vec{v}))$	$[\![E_1[\vec{v}/\vec{X}]]\!] \cup [\![E_1[\vec{v}/\vec{X}]]\!]_{[\Psi_\sigma(E_1[\vec{v}/\vec{X}]) \leftarrow \Psi_\sigma(E)]}$
	where `def`$(p(\vec{X}), E_1)$ is a variant of a definition s.t. $\mathrm{bn}(E_1) \cap \vec{v} = \emptyset$.

Fig. 1. Compilation rules for MMC.

The salient points of the compilation rules are as follows:

– In contrast to the CCS compiler [9], control states `entry` and `exit` are not included in the π-calculus compilation rules. Instead, these states are uniquely determined by the process expressions. This also avoids the generation of internal i-transitions in the compilation rules.
– The rules for `pref`, `choice`, `match`, and `proc` can be seen as direct encodings of the corresponding inference rules in Lin's symbolic semantics [15].
– The compilation rule for `nu` specifies that the transition rules of `nu`(X, E) are identical to the transition rules of E where free occurrences of X have been replaced with a fresh local name `name`(V). Note that transitions of `nu`(X, E) can be computed by discarding all transitions of E whose action is over channel X. This effect is achieved by considering at model-checking time only those transitions that are not over channels with local names. Additionally, a local name becomes global if it is output along a global channel using the `Open` rule. Thus the scope of names can only be completely determined at model-checking time, when transitions are generated, and not at compile time when transition *rules* are generated. Hence transition rules assume that every local name can eventually become global, and we check for locality of names when transitions are generated.
– The compilation rule for `par` precomputes, conservatively, all possible synchronizations between the parallel components. In general, we can determine whether two actions are complementary only when the binding of names is

known; hence we generate rules for `tau` transitions guarded by constraints that are evaluated at model-checking time.

3.2 Proof of the Compiler's Correctness

We show that MMC's interpreted transition relation (henceforth called `INT`) given in [26] is sound and complete with respect to the transition relation produced by the compiler.

Definition 5 A transition from state s_1 to s_2 with action a under constraint c is said to be *derivable* from a logic program $\mathcal{P}$ (denoted by $s_1 \xrightarrow{a,c}_{\mathcal{P}} s_2$) if $\texttt{trans(}s_1, X, Y, Z\texttt{)} \overset{\theta}{\Longrightarrow}_{\mathcal{P}} \Box$ (i.e. the query succeeds with answer θ) and there is a renaming ρ such that $X\theta\rho = a$, $Y\theta\rho \equiv c$ and $Z\theta\rho = s_2$.

A transition from state s to state s' where the action does not contain local names is denoted by $s \longmapsto_{\mathcal{P}} s'$; a sequence of zero or more such transitions is denoted by $s \overset{*}{\longmapsto}_{\mathcal{P}} s'$.

The soundness and completeness proofs make use of the following fact.

Lemma 1 (Extensionality) Let p and q be valid process expressions such that $p \longrightarrow_{\texttt{INT}} q$. Then any transition from q derived using the rules compiled from q can also be derived using the rules compiled from p and vice versa. That is, $\Psi_\sigma(q) \xrightarrow{a,c}_{[\![q]\!]} \Psi_\sigma(q')$ iff $\Psi_\sigma(q) \xrightarrow{a,c}_{[\![p]\!]} \Psi_\sigma(q')$.

The proof is by induction on the number of steps needed to derive a transition in $\Psi_\sigma(q)$.

The following lemma asserts that the transitions from an initial state derivable from the compiled transition relation can also be derived using `INT`.

Lemma 2 Let p be a valid process expression. Then $p \xrightarrow{a,c}_{\texttt{INT}} q$ (i.e. expression p becomes q after action a according to `INT`) whenever $\Psi_\sigma(p) \xrightarrow{a,c}_{[\![p]\!]} \Psi_\sigma(q)$ and a does not contain local names of p.

The proof is by induction on the number of steps needed to derive the transition from p using $[\![p]\!]$.

Theorem 3 (Soundness) Let e be a valid process expression and $e \overset{*}{\longmapsto}_{\texttt{INT}} p$. Then $p \xrightarrow{a,c}_{\texttt{INT}} q$ whenever $\Psi_\sigma(p) \xrightarrow{a,c}_{[\![e]\!]} \Psi_\sigma(q)$ and a does not contain local names of p.

The soundness theorem follows from Lemmas 1 and 2.

The following lemma asserts that the transitions from an initial state derivable from `INT` can be derived using the compiled transition relation.

Lemma 4 Let p be a valid process expression. If $p \xrightarrow{a,c}_{\texttt{INT}} q$ (i.e. expression p become q after action a according to `INT`) then $\Psi_\sigma(p) \xrightarrow{a,c}_{[\![p]\!]} \Psi_\sigma(q)$.

The proof is by induction on the number of steps needed to derive the transition from p using `INT`.

Theorem 5 (Completeness) Let e be a valid process expression and $e \overset{*}{\longmapsto}_{\mathtt{INT}} p$. If $p \xrightarrow{a,c}_{\mathtt{INT}} q$ then $\Psi_\sigma(p) \xrightarrow{a,c}_{[\![e]\!]} \Psi_\sigma(q)$.

The completeness theorem follows from Lemmas 1 and 4.

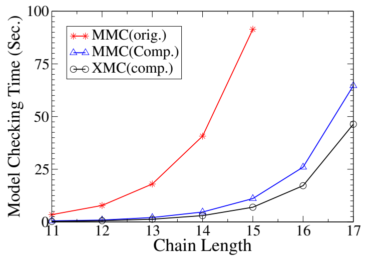
Implementation: The MMC compiler is implemented as a logic program that directly encodes the compilation rules of Figure 1. The use of *tabled resolution* makes such an implementation feasible, ensuring that each process expression in the program is compiled only once. Furthermore, tabling ensures that recursive process definitions can be compiled without extra control. More importantly, the direct implementation means that the correctness of the implementation follows from the correctness of the compilation rules. The implementation also uses partial evaluation to optimize the set of transition rules generated. The application of this optimization is straightforward, and is clearly reflected in the compiler's source code.

4 State-Space Reduction using AC Unification/Matching

One source of inefficiency in using the compiled transition relation (also in the interpreter `INT`) is the treatment of the product operator `prod`. In particular, it does not exploit the fact that `prod` is associative and commutative (AC); i.e., `prod(`s_1`, `s_2`)` is semantically identical to (has the same transitions as) `prod(`s_2`,`s_1`)`, and `prod(`s_1`,prod(`s_2`,`s_3`))` is semantically identical to `prod(prod(`s_1`,`s_2`),`s_3`)`. Thus, treating `prod` as an AC operator and using AC unification [22] during resolution will undoubtedly result in a reduction in the number of states that need to be examined during model checking.

AC matching and unification algorithms are traditionally viewed as prohibitively expensive in programming systems. Since, however, `prod` occurs only at the top-most level in terms representing states, a particularly efficient procedure for AC unification during clause selection can be attained. As is done in extant AC unification procedures (see, e.g. [14]), state terms are kept in a canonical form by defining an order among terms with non-AC symbols, and a term of the form `prod(`s_1`,prod(`s_2`,`s_3`))` is represented as the term `prod([`s_1`,`s_2`,`s_3`])` where [s_1,s_2,s_3] is a list whose component states occur in the order defined over non-AC terms.

State-space reduction is achieved by treating `prod` as an AC symbol and this can be seen as a form of *symmetry reduction* [10]. The state patterns generated at compile time are kept in canonical form. At model-checking time, the states derived from these patterns are rewritten (if necessary) to maintain canonical forms. Apart from generating fewer states at model-checking time, the compiler generates fewer transition rules when using AC unification. For example, consider the term E =`par(`E_1`,`E_2`)`. For E's autonomous transitions, the non-AC compiler generates rules of the form



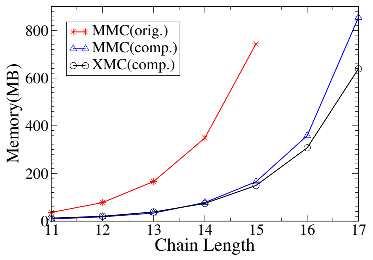


Fig. 2. Effect of compilation on chains of one-place buffers.

$$\begin{array}{ll} & \{\texttt{trans(prod(}s\texttt{,}V\texttt{),}a\texttt{,}c\texttt{,prod(}d\texttt{,}V\texttt{))} \mid \{\texttt{trans(}s\texttt{,}a\texttt{,}c\texttt{,}d\texttt{)} \in [\![E_1]\!]\} \\ \cup & \{\texttt{trans(prod(}V\texttt{,}s\texttt{),}a\texttt{,}c\texttt{,prod(}V\texttt{,}d\texttt{))} \mid \{\texttt{trans(}s\texttt{,}a\texttt{,}c\texttt{,}d\texttt{)} \in [\![E_2]\!]\} \end{array}$$

while the AC compiler generates $[\![E_1]\!] \cup [\![E_2]\!]$. Since rules common to $[\![E_1]\!]$ and $[\![E_2]\!]$ occur only once in $[\![E]\!]$, the number of transition rules for E is reduced.

The use of AC unification can lead to an exponential reduction in the size of the state space. Even in examples that do not display explicit symmetry, state-space reductions by factors of two or more can be seen (Section 5). The number of transition rules generated is also reduced, by more than a factor of two in most examples.

The AC unification operation itself is currently programmed in Prolog and is considerably more expensive than Prolog's in-built unification operation. As will be seen in Section 5, the overhead due to AC unification can be reduced (by a factor of five or more) through the use of AC matching and indexing techniques based on *discrimination nets* [2].

5 Performance Results

We used several model-checking benchmarks to evaluate the performance of the MMC compiler. All reported results were obtained on an Intel Xeon 1.7GHz machine with 2GB RAM running Debian GNU/Linux 2.4.21 and XSB version 2.5 (with slg-wam and local scheduling and without garbage collection).

Benchmark 1: Chains of one-place buffers. This example was chosen for three reasons. (1) We can use it to easily construct large state spaces: a chain of size i has a state space of size $O(2^i)$. (2) The example is structured such that any performance gains due to compilation are due strictly to the compact state representation, thereby allowing us to isolate this optimization's effect. (3) This example does not involve channel passing, thereby enabling us to see how MMC with compilation compares performance-wise to XMC.

The graphs of Figure 2 show the time and space requirements of the original MMC model checker, MMC with compiled transition rules, and the XMC system (with compiled transition rules) to verify deadlock freedom in chains of varying length. As expected, MMC with compilation outperforms the original MMC model checker both in terms of time and space. Moreover, MMC with

Instance	States			Transitions		
	Orig	Comp	AC	Orig	Comp	AC
1bsp	104	58	29	164	86	43
2bsp	607	408	76	1033	636	130
3bsp	3373	2304	224	5725	3600	416
2ms	N/A	73344	5026	N/A	227712	15461

(a)

Instance	Prop.	Time (Sec.)				Memory (MB)			
		Orig	Comp	AC	AC-net	Orig	Comp	AC	AC-net
1bsp	df	0.04	0.09	0.10	0.09	1.28	1.58	1.37	1.26
	ndl	0.04	0.07	0.15	0.09	1.42	1.60	1.39	1.27
2bsp	df	0.55	0.31	0.44	0.31	7.46	4.02	2.12	1.99
	ndl	0.86	0.31	0.42	0.30	9.69	4.44	2.21	2.06
3bsp	df	4.69	1.04	1.20	0.74	49.30	10.15	2.78	2.60
	ndl	6.90	1.13	1.21	0.76	64.77	16.51	3.06	2.82
2ms	df	N/A	47.01	75.64	33.72	N/A	276.29	24.70	24.47
	ndl	N/A	54.45	80.17	31.93	N/A	340.31	30.69	24.85

(b)

Table 1. (a)Number of states and transitions for variants of Handover protocol. (b) Performance of MMC for model checking variants of Handover protocol.

compilation approaches the time and space performance of XMC: the mechanisms needed to handle channel passing in MMC appear to impose an overhead of about 20% in time and 40% in space.

Benchmark 2: Handover procedure. Table 1(a) gives the number of states and transitions generated by three versions of MMC for four variants of the handover procedure of [18]; the four versions differ in the number of passive base stations (bsp) and mobile stations (ms). The column headings “Orig”, “Comp”, and “AC” refer to the original version of MMC, MMC with compilation, and MMC with compilation and AC reduction, respectively. The results show that MMC with compilation and AC reduction generates the fewest number of states and transitions whereas MMC without compilation generates the most. This is due to the fact that the performance of MMC with compilation is insensitive to the placement of the ν operator.

Table 1(b) presents the time and memory needed to verify the deadlock-freedom (df) and no-data-lost (ndl) properties of the handover protocol. Column heading “AC-net” refers to the version of MMC with AC discrimination nets; the other column headings are as in Table 1(a). Observe that MMC with compilation is more efficient and has superior scalability compared to MMC without compilation. Also observe that the use of AC unification reduces the number of states visited by up to a factor of 20 and space usage by a similar factor, although a concomitant increase in CPU time can be seen. The use of AC discrimination nets, however, mitigates this overhead by reducing the number of AC unification operations attempted, resulting in uniformly better time

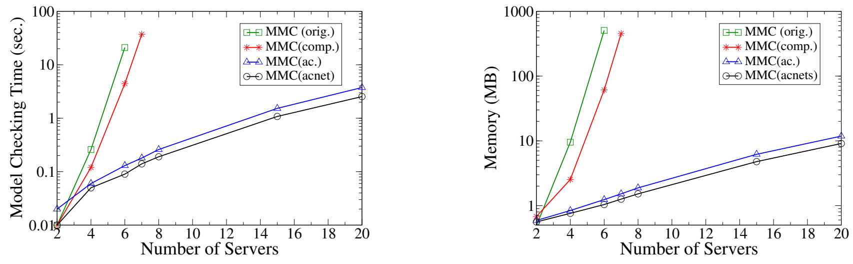


Fig. 3. Effect of AC-based symmetry reduction on chains of web-servers.

and space performance compared to all other schemes. Note that in the current implementation, both the AC unification and indexing operations are written as Prolog predicates while the non-AC unification and indexing operations use the primitives provided in the Prolog engine. Engine-level support for AC unification and indexing will result in further improvements in performance.

Benchmark 3: Variable-length chains of webservers (from [3]). This example models a file reader of a webserver. The file is divided into several blocks and each block is read and transmitted over the network by a separate process. Blocks can be read in parallel but are required to be transmitted in sequential order. Our AC-unification-based state-space reduction technique applied to this benchmark results in a state space that grows quadratically with the length of the chain, while non-AC techniques (compiled or original) result in a state space that grows exponentially in size. Figure 3 shows the time and memory requirements when the "order-preserved" property is verified on this example. MMC with compilation and AC unification performs best in terms of time, space, and scalability. Note that independent of the number of servers, the AC compiler generates the same number of transition rules (65). The discrimination net-based indexing improves the time and space performance even further.

Benchmark 4: Security protocols specified using the spi-calculus. Table 2(a) gives the number of states and transitions generated by three versions of MMC for three security protocols specified in the spi-calculus: Needham-Schroeder, Needham-Schroeder-Lowe and BAN-Yahalom. Observe that MMC with compilation and AC generates the fewest number (or the same as MMC with compilation) of states and transitions, whereas MMC without compilation generates the most. As mentioned above, this is because MMC without compilation is sensitive to the placement of ν operator. Table 2(b) gives the time (as $x + y$ where x is the compilation time and y is the model-checking time) and memory consumed when model checking these protocols. Compilation (with or without AC discrimination nets) yields an order of magnitude of improvement in time usage and a factor of 10-35% improvement in memory usage. The performance of MMC with AC is similar to that of MMC with AC discrimination nets and is not given in the table.

Benchmark	States			Transitions		
	Orig	Comp	AC-net	Orig	Comp	AC-net
Needham-Schroeder	167	164	164	287	282	282
Needham-Schroeder-Lowe	108	105	105	181	176	176
BAN-Yahalom	29133	6674	2011	107652	18106	5322

(a)

Benchmark	Prop.	Time (Sec.)			Memory (MB)		
		Orig	Comp	AC-net	Orig	Comp	AC-net
Needham-Schroeder	attack	0.02	0.07+0.01	0.09+0.02	0.70	1.03	1.12
Needham-Schroeder -Lowe	no attack	0.22	0.08+0.01	0.11+0.02	1.93	1.16	1.23
BAN-Yahalom	interleaving attack	0.11	0.16+0.01	0.18+0.01	2.15	1.89	1.67
	replay attack	0.14	0.16+0.00	0.18+0.01	2.88	1.78	1.65

(b)

Table 2. (a) Number of states and transitions for the spi-calculus examples. (b) Performance of MMC for model checking the spi-calculus examples.

6 Conclusions

We have shown that an optimizing compiler for the π- and spi-calculi can be constructed using logic-programming technology and other algorithms developed in the declarative-languages community. Extensive benchmarking data demonstrate that the compiler significantly improves the performance and scalability of the MMC model checker. The compiler is equipped with a number of optimizations targeting the issues that arise in a modeling formalism where channels can be passed as messages, and communication links can be dynamically created and altered via scope extrusion and intrusion. Of particular interest is the new symmetry-reduction technique that we have seamlessly integrated into the compiler though the use of AC-unification in resolution.

We are currently investigating techniques that adaptively apply the expensive AC operations only when necessary. We are also in the process of identifying conditions under which channels can be statically named, reducing the overhead incurred when most channels are fixed and only a few are mobile. This would enable us to tightly integrate the XMC system with MMC, paying the price for mobility only when channel names are dynamically created and communicated.

References

1. M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. In *Proceedings of CCS*, pages 36–47. ACM Press, 1997.
2. L. Bachmair, T. Chen, and I.V. Ramakrishnan. Associative-commutative discrimination nets. In *TAPSOFT*, pages 61–74, 1993.
3. S. Chaki, S.K.Rajamani, and J. Rehof. Types as models: model checking message-passing programs. In *Proceedings of POPL*, pages 45 – 57, 2002.

4. W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *Journal of the ACM*, 43(1):20–74, January 1996.
5. M. Dam. Proof systems for pi-calculus logics. *Logic for Concurrency and Synchronisation*, 2001.
6. G. Delzanno and S. Etalle. Transforming a proof system into Prolog for verifying security protocols. In *LOPSTR*, 2001.
7. G. Delzanno and A. Podelski. Model checking in CLP. In *Proceedings of Tools and algorithms for Construction and Analysis of Systems*, 1999.
8. Y. Dong, X. Du, G. Holzmann, and S. A. Smolka. Fighting livelock in the i-Protocol: A case study in explicit-state model checking. *Software Tools for Technology Transfer*, 4(2), 2003.
9. Y. Dong and C. R. Ramakrishnan. An optimizing compiler for efficient model checking. In *Proceedings of FORTE*, pages 241–256, 1999.
10. E. A. Emerson and A. P. Sistla. Symmetry and model checking. *Formal Methods in System Design: An International Journal*, 9(1/2):105–131, August 1996.
11. G. Gupta and E. Pontelli. A constraint based approach for specification and verification of real-time systems. In *IEEE Real-Time Systems Symposium*, 1997.
12. G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
13. A. Igarashi and N. Kobayashi. A generic type system for the pi-calculus. *Theoretical Computer Science*, 311(1–3):121–163, 2004.
14. H. Kirchner and P.-E. Moreau. Promoting rewriting to a programming language: a compiler for non-deterministic rewrite programs in associative-commutative theories. *J. Functional Prog.*, 11(2):207–251, 2001.
15. H. Lin. Symbolic bisimulation and proof systems for the π-calculus. Technical report, School of Cognitive and Computer Science, U. of Sussex, UK, 1994.
16. J. W. Lloyd. *Foundations of Logic Programming*. Springer, 1984.
17. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, Parts I and II. *Information and Computation*, 100(1):1–77, 1992.
18. F. Orava and J. Parrow. An algebraic verification of a mobile network. *Journal of Formal Aspects of Computing*, 4:497–543, 1992.
19. B. C. Pierce and D. N. Turner. Pict: a programming language based on the pi-calculus. In *Proof, language, and interaction: essays in honour of Robin Milner*, pages 455–494. MIT Press, 2000.
20. Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. L. Swift, and D. S. Warren. Efficient model checking using tabled resolution. In *Proceedings of CAV*. Springer, 1997.
21. H. Song and K. J. Compton. Verifying pi-calculus processes by Promela translation. Technical Report CSE-TR-472-03, Univ. of Michigan, 2003.
22. M. E. Stickel. A unification algorithm for associative-commutative unification. *Journal of the ACM*, 28(3):423–434, 1981.
23. L. Urbina. Analysis of hybrid systems in CLP(R). In *Constraint Programming (CP)*, 1996.
24. B. Victor. The Mobility Workbench user's guide. Technical report, Department of Computer Systems, Uppsala University, Sweden, 1995.
25. XSB. The XSB logic programming system v2.5, 2002. Available under GPL from `http://xsb.sourceforge.net`.
26. P. Yang, C. R. Ramakrishnan, and S. A. Smolka. A logical encoding of the π-calculus: Model checking mobile processes using tabled resolution. In *Proceedings of VMCAI*, 2003. Extended version in *Software Tools for Technology Transfer*, 6(1):38-66,2004.