Int J Softw Tools Technol Transfer (2004) 4: 1–29 / Digital Object Identifier (DOI) 10.1007/s10009-003-0136-3

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, SUNY at Stony Brook, Stony Brook, NY 11794-4400, USA
e-mail: {pyang, cram, sas}@cs.sunysb.edu

Published online: ■■ 2004 – © Springer-Verlag 2004

Abstract. We present MMC, a model checker for mobile systems specified in the style of the π-calculus. MMC's development builds on that of XMC, a model checker for an expressive extension of Milner's value-passing calculus implemented using the XSB tabled logic-programming engine. MMC 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 logic programming provides 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 of Abadi and Gordon. Our experimental data show that MMC outperforms other known tools for model checking the π-calculus.

Keywords: pi-calculus – Mobile processes – Model checking – Logic programming – Tabled resolution

1 Introduction

In previous work [34], we showed that logic programming with tabulation can be used to construct an efficient and versatile model checker for concurrent systems. In particular, we presented XMC, a model checker supporting XL (an extension of Milner's value-passing CCS [27]) 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 [44] 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),[1] avoids redundant subcomputations, and computes the well-founded model of normal logic programs.

In general, tabled resolution enables XSB to terminate when evaluating fixed points over finite domains. The classic example illustrating the benefits derived from tabling is in computing the transitive closure of a (finite) graph. Such a computation will always terminate in XSB while a standard Prolog engine (i.e., one without tabling) may not! Tabling enables us to encode problems involving fixed-point computations, such as model checking, at a high level and provides us with the machinery needed to solve such problems efficiently.

XMC is written in a highly declarative fashion. The model checker is encoded in less than 100 lines of XSB Prolog using a binary predicate `models/2` that defines when an XL term satisfies a modal μ-calculus formula. The `models` predicate in turn utilizes the ternary predicate `trans/3`, which represents the transition relation of the labeled transition system corresponding to the given 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* (the Mobility Model Checker), a practical model checker for mobile systems specified in the style of the π-calculus [29].

A preliminary version of this paper appeared as "A Logical Encoding of the π-Calculus: Model Checking Mobile Processes Using Tabled Resolution," P. Yang, C.R. Ramakrishnan, and S.A. Smolka. In *Proceedings of the Fourth International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2003)*, New York. Lecture Notes in Computer Science, vol. 2575, Springer-Verlag, pp. 116–131 (Jan. 2003).

[1] A datalog program is a logic program for which all function symbols are constants.

The main technical difficulties that we encountered are due, not surprisingly, to the ability to express *channel passing* in the π-calculus, which in turn raises 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. The key to a direct encoding of the operational semantics of the π-calculus as a logic program 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*. In particular, all top-down resolution techniques (including SLD, OLDT, and SLG resolution [11, 22, 38]) rename the variables in a clause that is selected at each resolution step in order to avoid the capture of free variables (a procedure known as *standardization apart* [4]). Our encoding directly exploits this mechanism to rename bound names in a π-calculus process definition whenever the definition is needed to identify a transition.

Our encoding enables us to treat scope extrusion and intrusion, renaming, name restriction, etc. in a direct and efficient manner. Moreover, resolution uses unification for passing parameters and return values. In addition to using unification to pass parameter values to process definitions in the π-calculus, we use unification to pass values during communication. The result is that the MMC version of the `trans` relation, when applied to a π-calculus expression p, generates exactly the labeled transition system for p as prescribed by the π-calculus's transitional semantics [29].

The proof of this result, given in Sect. 4, is nontrivial, involving simultaneous induction on the depth of the derivation tree for the given `trans` query (in the case of soundness) or π-calculus transition (in the case of completeness). Additionally, the soundness proof of the encoding requires special care to guard against name clashes in deriving an answer to a `trans` query, while the completeness proof demands a careful accounting of the alpha-conversions that occur during the derivation of transitions.

To carry out the completeness proof, we introduce a new symbolic semantics for the π-calculus, called the *constructive* semantics, which carefully controls the manner in which alpha-conversions are performed during a derivation. In particular, alpha-conversion is limited to the application of the inference rule for process definitions, and when applying this rule, bound names are always renamed to *fresh* names not previously encountered in the derivation.

We show that the constructive semantics is complete with respect to the symbolic semantics of [21] (which has been shown to be equivalent to the standard semantics of [29]) and then prove that MMC's `trans` relation is complete with respect to the constructive semantics. The process of building derivations is more deterministic in the constructive semantics compared to the symbolic semantics. Indeed, MMC's `trans` relation can be seen as an encoding of the constructive semantics.

When evaluated using a *tabled* resolution procedure such as OLDT or SLG resolution [11, 38], the `trans` relation can be used to finitely compute the set of all symbolic transitions (modulo names of bound variables) for any finite-control π-calculus expression.[2] Furthermore, by taking advantage of a logic-programming engine's ability to manipulate terms and to perform unification, we can encode the monadic and polyadic versions of the π-calculus in a single framework. We can also treat, as syntactic sugar, the encryption/decryption constructs of the spi-calculus [2], an extension of the π-calculus for cryptographic protocols. 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.

In summary, MMC's development and implementation exploits the close similarity between the treatment of names in the symbolic semantics of the π-calculus and the treatment of variables in the resolution procedures for logic programs. Referring back to our original question, the new insights required to derive a tabled-resolution-based model checker for the π-calculus can be seen as the following:

- Process definitions should be constructed such that (i) all bound names are unique and distinct from free names and (ii) all free names appear as process parameters. Process definitions constructed in this manner are said to be *valid* and *closed*, and any set of π-calculus definitions can be converted to valid and closed form in linear time. In the presence of valid and closed process definitions, the constructive semantics for the π-calculus is easily derived. The constructive semantics, which only requires alpha-conversion in the application of the **Ide** rule for (parameterized) process identifiers and their definitions, is provably equivalent to the standard semantics for the π-calculus. Its main benefit is that it is amenable to direct implementation in a tabled logic-progamming system.
- All operations for handling names in the constructive semantics (binding of names, alpha-conversion, etc.) can be performed using the operations for handling variables in logic-program resolution (unification, standardization apart, etc.). Also, checking whether two process expressions are alpha-equivalent in the constructive semantics can be done naturally using operations in tabled resolution that identify vari-

[2] Finite-control π-calculus expressions are those that do not contain a `!` operator (infinite replication) or a `|` operator (parallel composition) in the scope of a recursion.

ant terms (terms that differ only in the names of variables).

The rest of the paper is organized as follows. Section 2 outlines the notational conventions followed in this paper and provides an overview of query evaluation in logic programs. Section 3 describes the computational basis of MMC: the encoding of the operational semantics of the π-calculus as a tabled logic program. The soundness and completeness proofs for this encoding are presented in Sect. 4. Section 5 defines the subset of the π-μ-calculus handled by MMC and the implementation of a model checker for this logic in MMC. Section 6 addresses the extension of MMC to support the spi-calculus. Section 7 gives experimental results documenting MMC's performance. Section 8 discusses related work, and our concluding remarks appear in Sect. 9. The source code of MMC and the example programs appearing in this paper are publicly available for download from [45].

2 Preliminaries

Syntax and semantics of the π-calculus. The π-calculus [29] is a process algebra for systems whose interconnections may change dynamically. Let α denote the set of action prefixes, P the set of process expressions, N the set of process identifiers, and D the set of process definitions. Further, let $u, v, w, x, y, z, \ldots$ range over names and $p, q, r, \ldots$ range over process identifiers. The syntax of the π-calculus is as follows:

$$\begin{aligned}
\alpha &::= x(y) \mid \overline{x}y \mid \tau \\
P &::= 0 \mid \alpha.P \mid (\nu x)P \mid P \,|\, P \mid P + P \\
&\quad\mid [x = y]P \mid p(y_1, ..., y_n) \\
D &::= p(x_1, ..., x_n) \overset{\text{def}}{=} P \text{ (where } i \neq j \Rightarrow x_i \neq x_j) .
\end{aligned}$$

Prefixes $x(y)$, $\overline{x}y$, and τ represent input, output, and internal actions, respectively. 0 is the process with no transitions, while $\alpha.P$ is the process that can perform an α action and then behave as process P. $(\nu x)P$ behaves as P with x local to P, meaning that x cannot be used as a channel over which to communicate with the environment. Process $[x = y]P$ behaves as P if the names x and y match, and as 0 otherwise. The operators $+$ and $|$ represent nondeterministic choice and parallel composition, respectively. The expression $p(y_1, \ldots, y_n)$ 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. Process invocation may be used to define recursive processes. Each process definition of the form $p(x_1, \ldots, x_n) \overset{\text{def}}{=} P$ associates a process name p and a list of formal parameters $x_1, \ldots, x_n$ with process expression P.

Of the several approaches to defining the operational semantics of the π-calculus, the symbolic semantics of Lin [21] is closest to the encoding described in this paper. This semantics has been shown to be equivalent to the standard semantics given in [29]. According to the semantics of [21], transitions are inferred by keeping track of equalities between names and deriving transitions in the context of such constraints.

To make the paper self-contained, we recall the semantics of [21] in Fig. 1, which is given as a set of inference rules. A transition inferred by the semantics is denoted by $P \xrightarrow{M,\alpha} P'$, where P and P' are π-calculus process expressions, α is an action, and M is a set of symbolic constraints that describe the equalities between names under which the transition is enabled. Action α can be either an internal τ action, an input action $x(y)$ that inputs a datum into name y via name x, an output action $\overline{x}y$ that outputs name y via name x, or a bound output action $\overline{x}\nu y$ that outputs local name y via name x.

The syntax of constraints M is as follows:

$$M ::= \emptyset \mid \{x = y\} \mid M \cup M .$$

M can be an empty set, an equality constraint over a pair of names, or a set of equality constraints. $\emptyset$ is sometimes denoted as *true*, and the union of two sets of constraints M_1 and M_2 is also denoted as $M_1 M_2$.

Note that the semantics in Fig. 1 are defined up to alpha-equivalence, i.e., alpha-equivalent terms have exactly the same transitions. Let $\equiv$ denote alpha-equivalence. We can add the following rule to Lin's semantics:

$$\textbf{Alpha:} \quad \frac{P' \equiv P,\ P \xrightarrow{\alpha} Q,\ Q \equiv Q'}{P' \xrightarrow{\alpha} Q'} .$$

Notation. Following the conventions used in logic-programming languages such as Prolog, we denote variables by identifiers beginning with uppercase letters (possibly subscripted) or underscore ('`_`'). Specific variables are written in the teletype font (e.g., `X`); entities that range over variables are written in italic font (e.g., X). Function symbols are denoted either by special symbols (such as '+' and '[]') or by identifiers beginning with lowercase letters. Again, specific function symbols are written in the teletype font (e.g., `f`, `pref`, etc.), while entities that range over function symbols are denoted by f with/without subscripts and primes. A function symbol f with arity n is denoted by f/n; the arity is dropped whenever it is clear from the context. Function symbols with zero arity are called constants.

We assume standard notions of terms, substitutions, unification, and the most general unifier (*mgu*) of terms. Terms constructed from function symbols and variables are denoted by t with/without subscripts and primes. Sets of terms are denoted by T. Two terms are *variants* of each other if they are identical modulo names of variables. Two special function symbols '`.`'/2 and '`[]`'/0 are used to construct lists (representing "cons" and "nil", respectively). For convenience, the following notation for lists may also be used: $[t_1]$ for '`.`'$(t_1,$'`[]`'$)$, $[t_1|t_2]$ for '`.`'(t_1, t_2),

$$\textbf{Prefix:}\ \frac{}{\alpha.P \xrightarrow{true,\alpha} P}$$

$$\textbf{Sum:}\ \frac{P_1 \xrightarrow{M,\alpha} Q_1}{P_1 + P_2 \xrightarrow{M,\alpha} Q_1} \quad \frac{P_2 \xrightarrow{M,\alpha} Q_2}{P_1 + P_2 \xrightarrow{M,\alpha} Q_2}$$

$$\textbf{Ide:}\ \frac{P\{y_1,\dots,y_n/x_1,\dots,x_n\} \xrightarrow{M,\alpha} Q}{A(y_1,\dots,y_n) \xrightarrow{M,\alpha} Q} \quad A(x_1,\dots,x_n) \overset{\text{def}}{=} P$$

$$\textbf{Match:}\ \frac{P \xrightarrow{M,\alpha} Q}{[x=y]P \xrightarrow{ML,\alpha} Q} \quad L = \begin{cases} \emptyset & \text{if } x=y; \\ x=y & \text{otherwise.} \end{cases}$$

$$\textbf{Par:}\ (1)\frac{P_1 \xrightarrow{M,\alpha} Q_1}{P_1 \mid P_2 \xrightarrow{M,\alpha} Q_1 \mid P_2} \quad bn(\alpha) \cap fn(P_2) = \emptyset$$

$$(2)\frac{P_2 \xrightarrow{M,\alpha} Q_2}{P_1 \mid P_2 \xrightarrow{M,\alpha} P_1 \mid Q_2} \quad bn(\alpha) \cap fn(P_1) = \emptyset$$

$$\textbf{Res:}\ \frac{P \xrightarrow{M,\alpha} Q}{(\nu y)P \xrightarrow{M,\alpha} (\nu y)Q} \quad y \notin n(M,\alpha)$$

$$\textbf{Com:}\ \frac{P_1 \xrightarrow{M,y(z)} Q_1,\ P_2 \xrightarrow{N,\overline{x}v} Q_2}{P_1 \mid P_2 \xrightarrow{MNL,\tau} Q_1\{v/z\} \mid Q_2} \quad L = \begin{cases} \emptyset & \text{if } x=y; \\ x=y & \text{otherwise.} \end{cases}$$

$$\frac{P_1 \xrightarrow{M,\overline{x}v} Q_1,\ P_2 \xrightarrow{N,y(z)} Q_2}{P_1 \mid P_2 \xrightarrow{MNL,\tau} Q_1 \mid Q_2\{v/z\}} \quad L = \begin{cases} \emptyset & \text{if } x=y; \\ x=y & \text{otherwise.} \end{cases}$$

$$\textbf{Open:}\ \frac{P \xrightarrow{M,\overline{x}y} Q}{(\nu y)P \xrightarrow{M,\overline{x}\nu y} Q} \quad y \notin n(M,x)$$

$$\textbf{Close:}\ \frac{P_1 \xrightarrow{M,y(w)} Q_1,\ P_2 \xrightarrow{N,\overline{x}\nu w} Q_2}{P_1 \mid P_2 \xrightarrow{MNL,\tau} (\nu w)(Q_1 \mid Q_2)} \quad L = \begin{cases} \emptyset & \text{if } x=y; \\ x=y & \text{otherwise.} \end{cases}$$

$$\frac{P_1 \xrightarrow{M,\overline{x}\nu w} Q_1,\ P_2 \xrightarrow{N,y(w)} Q_2}{P_1 \mid P_2 \xrightarrow{MNL,\tau} (\nu w)(Q_1 \mid Q_2)} \quad L = \begin{cases} \emptyset & \text{if } x=y; \\ x=y & \text{otherwise.} \end{cases}$$

Fig. 1. π-calculus symbolic transition semantics of [21]

and $[t_1, t_2|t_3]$ for $[t_1|[t_2|t_3]]$. We use *vars*(t) to denote the set of variables in a term t, $\overline{t}$ to denote a sequence of terms, and overload *vars*$(\overline{t})$ to denote the set of variables in $\overline{t}$.

We use θ, σ to denote substitutions, which are functions from variables to terms. When manipulating expressions in the π-calculus, we often use *renaming functions*, which are a special case of substitutions, mapping names to names in the π-calculus, or variables to variables in MMC. We often write substitutions as sets of assignments for variables with t/X denoting the assignment of term t to X. We overload *vars*(θ) to denote the set of variables mentioned in θ, i.e., for each $t/X \in \theta$, $vars(t) \subseteq vars(\theta)$ and $X \in vars(\theta)$. Further, $vars(\theta\sigma)$ is defined as $vars(\theta) \cup vars(\sigma)$. Finally, we use $t\theta$ to represent the application of the substitution θ to term t and $T\theta$ to represent the set $\{t\theta \mid t \in T\}$.

We use identifiers beginning with lowercase letters to denote predicate symbols and p (with/without subscripts and primes) to range over predicate symbols. A predicate symbol p with arity n is written as p/n; the arity is dropped when it is clear from the context. Terms with predicate symbols at the root (and only at the root) are called *atoms*. We use *true* to denote the special atom that is true in all models. A *goal* G is a conjunction of atoms. A logic program is a set of Horn clauses, where each clause is of the form $p(\overline{t})$:– G, where $p(\overline{t})$ is known as the head of the clause and G as its body. A *definite* logic program is one that does not contain negative literals in the clause bodies. Note that facts are represented by clauses of the form $p(\overline{t})$:– *true*. $\mathcal{P}$ is used to denote programs.

Query evaluation in logic programs. Top-down evaluation of logic programs is traditionally based on one of several resolution mechanisms such as SLD, OLD, and SLG [11, 22]. We give here an overview of these mechanisms, focusing on those aspects of resolution that are most relevant to this paper, viz. the manner in which variables and substitutions are handled in top-down, goal-directed query

1. $$\frac{}{\texttt{true}:\ \theta \rightarrow \theta}$$
2. $$\frac{G_1:\ \theta \rightarrow \theta'',\ G_2:\ \theta'' \rightarrow \theta'}{G_1, G_2:\ \theta \rightarrow \theta'}$$
3. $$\frac{G':\ \theta\sigma \rightarrow \theta'}{H:\ \theta \rightarrow \theta'}$$ where (a) $H'\ :-\ G'$ is a variant of some clause in $\mathcal{P}$
(b) $(vars(H') \cup vars(G')) \cap (vars(\theta) \cup vars(H)) = \emptyset$; and
(c) $\sigma = mgu(H\theta, H')$

Fig. 2. Query evaluation using OLD resolution

evaluation. Completeness and other aspects of resolution are addressed in, e.g., [22].

Figure 2 presents the tableau rules for query evaluation over a definite logic program. Each rule is of the form

$$\frac{premises}{consequent},$$

where *premises* is a list of sequents and *consequent* is a single sequent. A *sequent* is of the form $G : \theta_c \rightarrow \theta_a$, where G is a goal, θ_c is the call substitution (the substitution before calling G), and θ_a is the answer substitution (the substitution after calling G).

Given an atom G, the atom $G\theta_a$ is an answer for query $G\theta_c$ if and only if we can construct a finite tableau for the sequent $G : \theta_c \rightarrow \theta_a$. The call substitution θ_c can be seen as the environment under which a goal is evaluated. Rule 1 in Fig. 2 states that the atom *true* leaves its environment unchanged. Rule 2 captures the effect of evaluating a conjunction of goals: the second goal in the conjunction is evaluated in an environment resulting from the evaluation of the first goal.

Rule 3 specifies how a goal is evaluated by looking up its definition(s) in the program. Given a goal H, a clause whose head can be unified with H is first selected from the program [Rule 3(a)]. Then the goal and the clause are standardized apart, i.e., the variables in the selected clause are renamed so that there are no name clashes between the clause and the variables in the goal or its environment [Rule 3(b)]. In the third step [Rule 3(c)], the goal is unified with the head of the selected clause. Then the goals in the body of the selected clause are evaluated in the environment θ with the substitution $mgu(H\theta, H')$, and the resultant environment θ' is computed.

Of these steps, the one that is most relevant to this paper is Rule 3(b): variables in the selected clause are renamed so as not to clash with variables in the goal or its call substitution. We exploit this renaming capability by encoding π-calculus names as Prolog variables; this gives us a direct and efficient means of computing the operational semantics of π-calculus processes.

Relationship to resolution. In general, resolution mechanisms construct a *derivation* for a goal by applying one or more resolution rules at each step. These mechanisms define both the derivability of answers to goals and a strategy for constructing such a derivation. The tableau rules specify only the derivability of answers and leave the strategy unspecified.

The primary rule of resolution is called *program clause resolution* and is captured by Rule 3 of Fig. 2. Mechanisms such as SLD resolution (Selection rule-driven Linear resolution for Definite clauses) not only specify all the possible ways to extend a derivation at any step but also specify a rule to select among the different possibilities. For instance, SLD resolution specifies that the atom selected for resolution at each step in the derivation is picked based on a *selection function* that picks the selected atom based on the entire derivation. Note that even after an atom is selected, there may be multiple ways to generate the next step in the derivation, corresponding to different program clauses that can be used to resolve against the selected atom. Although SLD resolution is complete (i.e., every atom in the model of a program is derivable), the selection function is not fixed a priori and hence is not implemented in logic-programming systems.

In contrast, OLD resolution (Ordered Linear resolution for Definite clauses) specifies that the first atom in a goal is selected for resolution at each step. Note that tableau Rule 2 also expands the first atom in a goal before expanding the rest of the goal. Prolog systems typically implement OLD resolution and, in addition, select the program clause to resolve within the order in which the clauses appear in the program. However, OLD resolution is not complete in the sense that there may be infinite derivations even when the logic program has a finite model. The inability of Prolog systems to handle logic programs with left recursion is a manifestation of this incompleteness.

It should be noted that resolution mechanisms generally are concerned with the notion of *strong completeness*: that every derivation for an atom in the model be finite to ensure that query evaluation terminates whenever there is an answer to the query. Tableau systems such as the one shown in Fig. 2 use a weaker notion of completeness that demands only the existence of a finite tableau for every atom in the model. Fixing the selection function a priori as done in OLD resolution (as well as in Fig. 2) affects only the claim of strong completeness. In fact, it is easy to construct a finite tableau using the rules in Fig. 2 for any finite derivation computed by SLD resolution.

OLDT resolution [38] augments OLD resolution with memo tables (hence its name: OLD resolution with Tabulation). In OLDT, the tables record certain aspects of the derivation (e.g., goals that have been selected and answers that have been computed for them). OLDT also adds another resolution rule called *answer clause resolution* to resolve a selected atom with a previously generated answer instead of with program clauses. The choice of resolution rule to apply at each step is driven by the contents of the memo table. OLDT resolution is complete for Datalog programs, i.e., logic programs where all function symbols are constants.

SLG [11] is a tabled resolution mechanism for general logic programs, i.e., programs that may contain negative literals on the right-hand side of clauses. The query-evaluation mechanism in the XSB system is an implementation of SLG resolution, but with an ordered selection strategy (analogous to OLDT resolution).

We use the tableau rules of Fig. 2 to show the soundness and completeness of our logic-programming-based encoding of the transitional semantics of the π-calculus. Thus the correctness of our encoding is based not on the operational details of a particular resolution procedure but on the abstract notion of derivations in logic programs.

Note that tabled resolution procedures such as OLDT and SLG resolution have been shown to be complete with respect to the tableau rules of Fig. 2. Thus we can establish that our *implementation* itself is complete when our encoding is evaluated with an engine such as the XSB system that faithfully implements a tabled resolution procedure.

3 Encoding the operational semantics of the π-calculus

In this section, we describe our encoding of the operational semantics of the π-calculus in MMC. We begin by showing how π-calculus process expressions are encoded in MMC (Sect. 3.1). The operational semantics of the π-calculus is then given as the Prolog relation `trans` that, given a process definition, generates the corresponding symbolic transition system (Sect. 3.2). For simplicity, we first describe the encoding for the *monadic* π-calculus. This encoding is later modified to include optimizations for reducing the size of the generated symbolic transition system (Sect. 3.3) and then extended to the polyadic π-calculus (Sect. 3.4).

3.1 Syntax of MMC processes

We use $\mathcal{P}$ to denote the set of all process expressions and P and Q, possibly subscripted, to range over individual process expressions. We use $\mathcal{V}$ to denote an enumerable set of names and $X, X_1, X_2, \ldots$ to range over elements of $\mathcal{V}$. $\vec{\mathcal{V}}$ denotes a comma-separated list of names. In MMC, names are represented by Prolog variables. We use $\mathcal{N}$ to denote the enumerable set of process names and $p, p_1, p_2, \ldots$ 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:

$$\begin{aligned}
\mathcal{A} &::= \texttt{in}(\mathcal{V},\mathcal{V}) \mid \texttt{out}(\mathcal{V},\mathcal{V}) \mid \texttt{tau}\\
\mathcal{P} &::= \texttt{zero} \mid \texttt{pref}(\mathcal{A},\mathcal{P}) \mid \texttt{nu}(\mathcal{V},\mathcal{P}) \mid \texttt{par}(\mathcal{P},\mathcal{P})\\
&\qquad \mid \texttt{choice}(\mathcal{P},\mathcal{P}) \mid \texttt{match}(\mathcal{V}\texttt{=}\mathcal{V},\mathcal{P}) \mid \texttt{proc}(\mathcal{N}(\vec{\mathcal{V}}))\\
\mathcal{D} &::= \texttt{def}(\mathcal{N}(\vec{\mathcal{V}}),\mathcal{P})\,.
\end{aligned}$$

It is easy to see that MMC's syntax encodes the standard syntax for the π-calculus given in [29] and recounted in Sect. 2. The correspondence between MMC and π-calculus syntax is formalized in the following definition.

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

$$\begin{aligned}
f_\psi(\textit{zero}) &= 0\\
f_\psi(\textit{tau}) &= \tau\\
f_\psi(\textit{in}(X_1,X_2)) &= \psi(X_1)(\psi(X_2))\\
f_\psi(\textit{out}(X_1,X_2)) &= \overline{\psi(X_1)}\psi(X_2)\\
f_\psi(\textit{outbound}(X_1,X_2)) &= \overline{\psi(X_1)}\nu\psi(X_2)\\
f_\psi(\textit{pref}(A,P)) &= f_\psi(A).f_\psi(P)\\
f_\psi(\textit{nu}(X,P)) &= (\nu\psi(X))f_\psi(P)\\
f_\psi(\textit{par}(P,Q)) &= f_\psi(P) \mid f_\psi(Q)\\
f_\psi(\textit{choice}(P,Q)) &= f_\psi(P) + f_\psi(Q)\\
f_\psi(\textit{match}((X_1=X_2),P)) &= [\psi(X_1)=\psi(X_2)]f_\psi(P))\\
f_\psi(\textit{proc}(p(X_1,\ldots,X_n))) &= p(\psi(X_1),\ldots,\psi(X_n))\,.
\end{aligned}$$

Bound output actions ($\texttt{outbound}(X_1,X_2)$ in MMC and $\overline{\psi(X_1)}\nu\psi(X_2)$ in the π-calculus) are not actually syntactic constructs but rather arise in the context of the π-calculus's operational semantics and our encoding of the same (Sect. 3.2).

We can also map MMC process definitions to π-calculus process definitions using the syntax transformer η.

Definition 2. *Function η mapping process definitions in MMC's syntax to the standard π-calculus syntax is defined as follows:*

$$\eta(\textit{def}(p(X_1,\ldots,X_n),P)) = p(\rho(X_1),\ldots,\rho(X_n)), \\ \stackrel{\text{def}}{=} f_\rho(P)$$

where ρ is a one-to-one function mapping Prolog variables to π-calculus names.

MMC syntax	π-calculus syntax
`def(s(`X_4`),par(proc(p(`X_4`)),proc(q(`X_4`)))).`	$s(x_4) \overset{\text{def}}{=} p(x_4) \mid q(x_4)$
`def(p(`X_1`),pref(in(`X_1`,`X_2`),proc(p(`X_1`)))).`	$p(x_1) \overset{\text{def}}{=} x_1(x_2).p(x_1)$
`def(q(`X_3`),nu(`X_2`,pref(out(`X_3`,`X_2`),proc(q(`X_3`))))).`	$q(x_3) \overset{\text{def}}{=} (\nu x_2)\overline{x_3}x_2.q(x_3)$

Fig. 3. Example process definitions given in both MMC and π-calculus syntax

Example 1. In Fig. 3, three example process definitions are given in both the MMC and π-calculus syntax. □

Definition 1 allows us to directly import the notions of bound and free names from the 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 the `in` and `outbound` actions is said to be *bound*. Among the bound names of a process P, it is sometimes useful to distinguish between names bound by input actions and the other bound names. The latter are called the *local* names of P, denoted by $ln(P)$). Table 1 inductively defines the free, bound, and local names of MMC process expressions (columns 2–4, respectively).

Note that the same name may occur both bound and free in a process expression. For example, in `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`.

The encoding of our model checker becomes considerably simpler if we ensure that bound names are all distinct from each other and from the free names. We say that process expressions having this distinct-name property are *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 inductively in the fifth column of Table 1.

Definition 3 (Validity). *A process expression P is valid if $bn(P)$ are Prolog variables, $fn(P) \cap bn(P) = \emptyset$ and $ubn(P) = bn(P)$. A process definition of the form* **def**$(p(\vec{X}), P)$ *is valid if P is valid and $bn(P) \cap \vec{X} = \emptyset$, i.e., formal parameters do not appear bound in P.*

The following property can be easily 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 $fn(P) = \emptyset$. A process definition of the form `def`$(p(\vec{X}), P)$ is *closed* if all free names in P occur in $\vec{X}$, i.e., $fn(P) \subseteq \vec{X}$. The encoding of the MMC model checker requires that all process definitions be 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. Considering only valid and closed process definitions ensures that the simple syntax transformer η (Definition 2) preserves the distinctness of free names and also prevents the capture of free names by bound names.

We can also recast π-calculus process expressions and definitions in MMC's syntax. This is achieved by defining a function g_φ (analogous to f_ψ) over π-calculus process expressions, given a one-to-one mapping φ of π-calculus names to Prolog variables, and a function ζ (analogous to η) over π-calculus process definitions. Note that the notions of validity of process expressions and closed process definitions can be lifted to the π-calculus as well.

We will exploit the analogy between the syntactically distinct domains of the π-calculus and MMC to help

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

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

establish the equivalence between the semantics of the π-calculus and MMC. We begin by defining the operational semantics of MMC in terms of a logic program.

3.2 Operational semantics of MMC

The operational semantics of the π-calculus is traditionally given in terms of a symbolic transition system [21, 32] where transitions are of the form $P \xrightarrow{M,\alpha} Q$, signifying that process P can perform an α action and then behave as process Q if constraint M holds. As we will subsequently show (Theorems 2 and 3), the `trans` relation defined by the rules of Fig. 4 is a direct encoding of the symbolic semantics of [21] and Fig. 1.

Intuitively, a tuple $\texttt{trans(}P_1\texttt{,}A\texttt{,}M\texttt{,}P_2\texttt{)}$ in the `trans` relation means that process expression P_1 can evolve into process expression P_2 via the execution of an A action provided that the set M of equality constraints over names holds. A set of equality constraints is encoded as a conjunction of constraints in Prolog, each conjunct representing an equality constraint over a pair of names. A tuple in the `trans` relation such as the one above corresponds to a transition in the symbolic semantics of Fig. 1 of the form $f_\psi(P_1) \xrightarrow{fc_\psi(M), f_\psi(A)} f_\psi(P_2)$, where fc_ψ is defined as follows.

Definition 4. *Function fc_ψ maps the MMC representation of equality constraints over names to equivalent constraints over π-calculus names as follows:*

$$\begin{aligned} fc_\psi(true) &= \emptyset \\ fc_\psi(X_1 = X_2) &= \{\psi(X_1) = \psi(X_2)\} \\ fc_\psi((M_1, M_2)) &= fc_\psi(M_1) fc_\psi(M_2)\,. \end{aligned}$$

Similarly, given a one-to-one function φ mapping π-calculus names to MMC variables, we can define the function gc_φ that maps equality constraints over π-calculus names to an equivalent MMC representation of equality constraints.

The following example illustrates how the operational semantics of the π-calculus is computed in MMC.

Example 2. Recall the MMC process definitions of Example 1.

(1) $\texttt{def(s(}X_4\texttt{), par(proc(p(}X_4\texttt{)), proc(q(}X_4\texttt{)))).}$
(2) $\texttt{def(p(}X_1\texttt{), pref(in(}X_1\texttt{,}X_2\texttt{), proc(p(}X_1\texttt{)))).}$
(3) $\texttt{def(q(}X_3\texttt{), nu(}X_2\texttt{, pref(out(}X_3\texttt{,}X_2\texttt{), proc(q(}X_3\texttt{))))).}$

We carefully explain how the transitions of processes $\texttt{proc(p(}Y_1\texttt{))}$, $\texttt{proc(q(}Y_2\texttt{))}$, and $\texttt{proc(s(}Y_3\texttt{))}$ are computed. We use the notation `_$n` to denote the different fresh variables generated during the computation of these transitions and use variables U_i as query variables.

1. Consider the query $\texttt{trans(}Pr\texttt{, }U_1\texttt{, }U_2\texttt{, }U_3\texttt{)} : \theta_1 \to \theta_2$, where $Pr\theta_1 = \texttt{proc(p(}Y_1\texttt{))}$ and $\{U_1, U_2, U_3\} \cap vars(\theta_1) = \emptyset$. The **Ide** clause binds PN to $\texttt{p(}Y_1\texttt{)}$.

```
% Pref
trans(pref(A, P), A, true, P).

%Sum
trans(choice(P1, P2), A, M, Q1) :- trans(P1, A, M, Q1).
trans(choice(P1, P2), A, M, Q2) :- trans(P2, A, M, Q2).

% Ide
trans(proc(PN), A, M, Q) :- def(PN, P), trans(P, A, M, Q).

% Match
trans(match((X=Y), P), A, ML, Q) :- X==Y, trans(P, A, ML, Q).
trans(match((X=Y), P), A, (X=Y,M), Q) :- X\==Y, trans(P, A, M, Q).

% Par
trans(par(P1, P2), A, M, par(Q1, P2)) :- trans(P1, A, M, Q1).
trans(par(P1, P2), A, M, par(P1, Q2)) :- trans(P2, A, M, Q2).

% Com
trans(par(P1, P2), tau, (M, N, L), par(Q1, Q2)) :-
      trans(P1, A, M, Q1),
      trans(P2, B, N, Q2),
      complement(A, B, L).

% Res
trans(nu(Y, P), A, M, nu(Y, Q)) :-
      trans(P, A, M, Q),
      /* Y does not appear in action A*/
      not_in_action(Y, A),
      /* Y does not appear in constraint M*/
      not_in_constraint(Y, M).

% Open
trans(nu(Y, P), outbound(X, Z), M, Q) :-
      trans(P, out(X, Z), M, Q),
      Y==Z, Y\==X,
      not_in_constraint(Y, M).

% Close
trans(par(P1, P2), tau, (M,N,L), nu(W, par(Q1, Q2))) :-
      trans(P1, A, M, Q1),
      trans(P2, B, N, Q2),
      comp_bound(A, B, W, L).

not_in_action(Y, in(X, Z)) :- Y \== X, Y \== Z.
not_in_action(Y, out(X, Z)) :- Y \== X, Y \== Z.
not_in_action(Y, outbound(X, Z)) :- Y \== X, Y \== Z.
not_in_action(Y, tau).

not_in_constraint(X, true).
not_in_constraint(X, (Y=Z)) :- X\==Y, X\==Z.
not_in_constraint(X, (M,N)) :-
    not_in_constraint(X,M), not_in_constraint(X,N).

complement(in(X, V), out(Y, V), true) :- X==Y.
complement(in(X, V), out(Y, V), (X=Y)) :- X\==Y.
complement(out(X, V), in(Y, V), true) :- X==Y.
complement(out(X, V), in(Y, V), (X=Y)) :- X\==Y.

comp_bound(outbound(X, W), in(Y, W), W, true) :- X==Y.
comp_bound(outbound(X, W), in(Y, W), W, (X=Y)) :- X\==Y.
comp_bound(in(X, W), outbound(Y, W), W, true) :- X==Y.
comp_bound(in(X, W), outbound(Y, W), W, (X=Y)) :- X\==Y.
```

Fig. 4. Logic program `MMCtrans` encoding π-calculus transitional semantics

When predicate $\texttt{def(p(}Y_1\texttt{),}P\texttt{)}$ is invoked, all occurrences of X_1 and X_2 in process definition (2) are renamed to fresh variables (Rule 3 in Fig. 2), say, `_$1` and `_$2`, respectively. Y_1 and P are then unified with `_$1` and $\texttt{pref(in(}Y_1\texttt{,\_\$2), proc(p(}Y_1\texttt{)))}$, respectively. Thus process $\texttt{p(}Y_1\texttt{)}$ has the same behavior as process $\texttt{pref(in(}Y_1\texttt{,\_\$2), proc(p(}Y_1\texttt{)))}$. By applying the **Prefix** clause, we obtain the answer ($U_1\theta_2 =$ `in`(Y_1,`_$2`), $U_2\theta_2 =$ `true`, $U_3\theta_2 =$ `proc(p(`Y_1`))`).

2. Consider the query `trans(`Qr, U_4, U_5, U_6`)` : $\theta_3 \to \theta_4$, where $Qr\theta_3 =$ `proc(q(`Y_2`))` and $\{U_4, U_5, U_6\} \cap vars(\theta_3) = \emptyset$. From the **Ide** clause, all occurrences of X_3 and X_2 in process definition (3) are renamed to fresh variables, say, `_$3` and `_$4`, respectively. Process `q(`Y_2`)` then has the same behavior as process `nu(_$4, pref(out(`Y_2`,_$4), proc(q(`Y_2`))))`. Both the **Open** and **Res** clauses can be applied to compute the transitions of process `nu(_$4, pref(out(`Y_2`,_$4), proc(q(`Y_2`))))` when we consider only the left-hand side of these clauses. Although the `pref(out(`Y_2`, _$4), proc(q(`Y_2`)))` process can perform the output action `out(`Y_2`, _$4)`, the **Res** clause fails since local name `_$4` is in the action. The **Open** clause, on the other hand, succeeds with the answer ($U_4\theta_4 =$ `outbound(`Y_2`,_$4)`, $U_5\theta_4 =$ `true`, $U_6\theta_4 =$ `proc(q(`Y_2`))`).
3. Consider the query `trans(`Sr, U_7, U_8, U_9`)` : $\theta_c \to \theta_a$, where $Sr\theta_c =$ `proc(s(`Y_3`))` and $\{U_7, U_8, U_9\} \cap vars(\theta_c) = \emptyset$. From the **Ide** clause, all occurrences of X_4 in process definition (1) are renamed to a fresh variable, say, `_$5`. Process `s(`Y_3`)` then has the same behavior as process `par(proc(p(`Y_3`)), proc(q(`Y_3`)))`. From 1. and 2. above, and by applying the **Par** clause, process `par(proc(p(`Y_3`)), proc(q(`Y_3`)))` can perform input action `in(`Y_3`,_$2)` and bound output action `outbound(`Y_3`, _$4)`. To compute the synchronous transitions, we cannot apply the **Com** clause since neither process can perform an output action. We can, however, apply the **Close** clause since the channel names used in actions `in(`Y_3`,_$2)` and `outbound(`Y_3`, _$4)` are the same (identity check `==`).
Note that the evaluation of the query for `proc(p(`Y_3`))` occurs earlier than that of the query for `proc(q(`Y_3`))`. Therefore, although two different names are represented by the same variable X_2 in process definitions (2) and (3), the resolution mechanism renames these two instances of X_2 to different variables [Rule 3(b) in Fig. 2], i.e., `_$2` in step 1 above and `_$4` in step 2 are different. The **Close** clause then binds `_$2` to `_$4` (unification `=`), indicating that a name originally private to process `q` is now in the scope of `q`. This corresponds to *scope extrusion* in the π-calculus. Thus process `par(proc(p(`Y_3`)), proc(q(`Y_3`)))` can perform a `tau` action. The answers to this query are (a) ($U_7\theta_a =$ `in(`Y_3`, _$2)`, $U_8\theta_a =$ `true`, $U_9\theta_a =$ `proc(p(`Y_3`))`), (b) ($U_7\theta_a =$ `outbound(`Y_3`, _$4)`, $U_8\theta_a =$ `true`, $U_9\theta_a =$ `proc(q(`Y_3`))`), and (c) ($U_7\theta_a =$ `tau`, $U_8\theta_a =$ `true`, $U_9\theta_a =$ `nu(_$4, par(proc(p(`Y_3`)), proc(q(`Y_3`))))`). □

3.3 Optimizing the encoding of the operational semantics

The encoding of Fig. 4 can be optimized for performance in several ways.

Eliminating unused names. While the encoding of the operational semantics is sound and complete, it is not yet sufficient to build a model checker: the semantics distinguishes process expressions based on their syntax, even if their behavior is 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` is the parallel composition of processes `ser(Pc)` and `cli(Pc)`. Process `ser(Pc)` repeatedly generates a new local name `X` and sends `X` to process `cli(Pc)`. Since each time this happens the name `X` is different from any other name previously generated during the computation, the state space of `system` is infinite, as shown below:

$$\begin{aligned}
&\texttt{system} = \texttt{nu}(Pc,\texttt{par(proc(ser(}Pc\texttt{)),proc(cli(}Pc\texttt{))))}\\
&\xrightarrow{\tau} \texttt{nu}(Pc,\texttt{nu}(X,\texttt{par(proc(ser(}Pc\texttt{)),proc(cli(}Pc\texttt{)))))}\\
&\xrightarrow{\tau} \texttt{nu}(Pc,\texttt{nu}(X,\texttt{nu}(X',\\
&\qquad\qquad \texttt{par(proc(ser(}Pc\texttt{)),proc(cli(}Pc\texttt{))))))}\\
&\xrightarrow{\tau} \;\ldots
\end{aligned}$$

The state space of process `system` can be made finite by noticing that local names that are not actually used can be eliminated. For example, since X does not occur in `par(proc(ser(`Pc`)), proc(cli(`Pc`)))`, we can remove this name from process `nu(`X`, par(proc(ser(`Pc`)), proc(cli(`Pc`))))` without affecting its behavior. That is, the behaviors of `nu(`X`, par(proc(ser(`Pc`)), proc(cli(`Pc`))))` and `par(proc(ser(`Pc`)), proc(cli(`Pc`)))` are identical. This observation can be formalized as the following structural-congruence rule:

$$\texttt{nu(X,P)} \equiv \texttt{P} \quad \text{if } \texttt{X} \notin fn(\texttt{P}).$$

We can use this rule to modify clauses **Res** and **Close**, which handle local names; after doing so, process `system` exhibits finite behavior. For example, the **Res** clause becomes

```
trans(nu(Y, P), A, M, Q) :-
    trans(P, A, M, P1),
    not_in_action(Y, A),
    not_in_constraint(Y, M),
    (occurs(Y, P1)
       -> Q = nu(Y, P1)
       ;  Q = P1 ).
```

Goal reordering. Consider the **Com** rule. In general, the number of solutions of `complement(A, B, L)` is much smaller than the number of tuples in `trans(P2, B, N, Q2)`. Thus, by reordering `trans(P2, B, N, Q2)` and `complement(A, B, L)`, we will compute fewer intermediate answers. This optimization has been applied to early versions of the XMC model checker [34] and resulted in significant performance gains. However, representing local names using variables means that the program is dependent on the order in which variables are bound, and hence goal reordering is not directly applicable. In particular, in clauses **Com** and `Close`, the computation of complementary actions (`complement` and `comp_bound`) cannot be moved ahead of the computation of the second `trans` goal: the computation of complementary actions checks for identity of channel names, and one of the actions becomes unknown when the goals are reordered.

We overcome this difficulty and enable the reordering of goals by moving the check for identity of channel names from the definition of `complement` and `comp_bound` into the definition of `trans` itself. For example, consider the prefix clause of `trans`. When the action (the second argument) is known, the rule will be applied only when the channel names in the prefix action and the given action are identical; when the action is unknown, the logical variable denoting the unknown action is simply bound to the action in the prefix. This change requires us to identify and separate two cases throughout the computation of `trans`: when the action (the second argument) is known and when it is unknown. We do so by specializing the `trans` clauses for these two cases.

3.4 From monadic to polyadic π-calculus

The polyadic version of the π-calculus is supported in MMC by extending the above encoding 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 Sect. 3.1 becomes (only the changed rules are shown)

$$\mathcal{A} ::= \texttt{in}(\mathcal{V}, \mathcal{T}) \mid \texttt{out}(\mathcal{V}, [\,], \mathcal{T})$$

$$\mathcal{P} ::= \texttt{unify}((\mathcal{V} = \mathcal{T}), P) \mid \texttt{proc}(\mathcal{PN}(\overrightarrow{\mathcal{T}}))\,.$$

In essence, 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. We use terms of the form $\texttt{out}(\mathcal{V}, \overrightarrow{\mathcal{V}}, \mathcal{T})$ to represent output actions (when $\overrightarrow{\mathcal{V}} = [\,]$) and bound output actions ($\overrightarrow{\mathcal{V}} \neq [\,]$). That is, $\overrightarrow{\mathcal{V}}$ is used to keep track of the set of local names in a message to support the scope extrusion of multiple names. Correspondingly, we need to change the **Open** clause and combine the **Close** and **Com** clauses in the transitional semantics of the polyadic version. The changes are given in Fig. 5.

According to the **Open** clause, if process P can perform the output action `out(`X`, [ ], `Z`)` where Y occurs in Z, then process `nu(`Y`, `P`)` can make the bound output action `out(`X`, `$[Y]$`, `Z`)`. Similarly, if process P can make the bound output action `out(`X`, `$[V_1, \dots, V_n]$`, `Z`)` with Y occurring in Z, then process `nu(`Y`,`P`)` can make the bound output action `out(`X`, `$[Y, V_1, \dots, V_n]$`, `Z`)`. Clauses **Close** and **Com** are combined into one rule. In the **Close/Com** rule, predicate `comp_bound(`A, B, W, L`)` checks if A and B are complementary actions, i.e., actions of the form `in(`X`,`Y`)` and `out(`X`,`W`,`Z`)`. Process `par(`P`,`Q`)` can perform a synchronizing `tau` transition if processes P and Q can perform complementary actions. After the `tau` transition, `par(`P`, `Q`)` evolves to `par(`$P1$`, `$Q1$`)` if W is empty and to `nu(`V_1`, ..., nu(`V_n`,par(`$P1$`,`$Q1$`))...)` if W is $[V_1, \dots, V_n]$. This construction is handled in predicate `makestate(`W`, par(`$P1$`,`$Q1$`), `Ns`)`.

We also introduce the operator `unify` to decompose a term into subterms by pattern matching. The semantics of `unify` is as follows:

```
trans(unify((X=T),P), A, C, T) :-
      X = T, trans(P, A, C, T).
```

The names in T are bound names in an expression of the form $\texttt{unify}((X = T), P)$. An expression $\texttt{unify}((X = T), P)$ 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.

4 Soundness and completeness of the encoding

Our encoding of the symbolic semantics of the π-calculus (Fig. 1) is given in Fig. 4 as a logic program. In this section, we prove the soundness and completeness of the encoding. By soundness we mean that every transition derivable by query evaluation over our encoding is also derivable from the symbolic semantics of Fig. 1. By completeness we mean that every derivation in the symbolic semantics of the π-calculus has an equivalent derivation in MMC.

When talking about deriving transitions via query evaluation, we are specifically referring to the rules of Fig. 2, which are the basic rules of resolution. Our encoding checks the identity of Prolog variables using built-in

```
Open:
trans(nu(Y,P), out(X,[Y|R1],Z), M, Q):-
      trans(P, out(X,R1,Z), M, Q),
      contains(Z, Y), Y\==X,
      not_in_constraint(Y, M).
```

```
Close/Com:
trans(par(P,Q), tau, (M,N,L), Ns):-
      trans(P, A, M, P1),
      trans(Q, B, N, Q1),
      comp_bound(A, B, W, L),
      makestate(W, par(P1,Q1), Ns).
```

Fig. 5. Open, **Close**, and **Com** rules for polyadic π-calculus

1. $$\frac{}{Y == Z:\ \theta \rightarrow \theta}$$
   $\forall\sigma.\theta\sigma$ is consistent $\Rightarrow Y\theta\sigma = Z\theta\sigma$.
2. $$\frac{}{Y\backslash == Z:\ \theta \rightarrow \theta}$$
   $\exists\sigma.\theta\sigma$ is consistent and $Y\theta\sigma \neq Z\theta\sigma$.

Fig. 6. Resolution rules for built-in Prolog operators: $Y == Z$ and $Y\backslash == Z$

operators '`==`' and '`\==`'. The rules of resolution for these operators are given in Fig. 6.

In the following theorem, Part 6 states that our encoding is sound; the other parts are propositions that are used to establish this result. In particular, Parts 1–4 are used to prove Part 5, which in turn is used to establish Part 6. The proof of the theorem is by induction on the length of the derivation using the resolution rules of Figs. 2 and 6. The induction is *simultaneous* in the sense that all six parts of the theorem are proved simultaneously. `MMCtrans` in the proof refers to the logic program of Fig. 4.

Theorem 2 (Soundness). *Let D be a set of process definitions and P_r be a Prolog variable. Assume that $\mathtt{trans}(P_r, U_1, U_2, U_3) : \theta_c \rightarrow \theta_a$ is an answer derivable from the logic program $D \cup \mathtt{MMCtrans}$, where $var(\theta_c) \cap \{U_1, U_2, U_3\} = \emptyset$ and $P_r\theta_c$ is a valid process expression. Let $\mathtt{trans(S1, A, M, T1)} = \mathtt{trans}(P_r, U_1, U_2, U_3)\theta_a$. Then the following hold:*

1. ***(Preservation of free and local names).*** *For every free name $F_v \in fn(P_r\theta_c)$, $F_v = F_v\theta_a$; for every local name $L_v \in ln(P_r\theta_c)$, $L_v = L_v\theta_a$.*
2. ***(Origin of free names).*** *For every name $V \in (fn(\mathtt{A}) \cup n(\mathtt{M}))$, $V \in fn(P_r\theta_c)$; for every name $V_1 \in fn(\mathtt{T1})$, $V_1 \in fn(P_r\theta_c)$ or $V_1 \in bn(\mathtt{A})$.*
3. ***(Origin of bound names).*** *For every name $Y \in (bn(\mathtt{A}) \cup bn(\mathtt{T1}))$, Y is a variable and ($Y \in bn(P_r\theta_c)$ or $Y \notin vars(\theta_c)$).*
4. ***(Distinctness of bound names).*** *$vars(bn(\mathtt{A})) \cap vars(bn(\mathtt{T1})) = \emptyset$.*
5. ***(Validity of destination).*** *$\mathtt{T1}$ is a valid process expression.*
6. ***(Soundness of transition).*** *Given a one-to-one function ψ mapping MMC variables to π-calculus names such that $vars(P_r\theta_c) \subseteq domain(\psi) \subseteq vars(\theta_c)$, there exists an extension ψ' of ψ such that $f_\psi(P_r\theta_c) \overset{fc_{\psi'}(\mathtt{M}), f_{\psi'}(\mathtt{A})}{\longrightarrow} f_{\psi'}(\mathtt{T1})$ is a derivation in the symbolic semantics of the π-calculus in Fig. 1 with respect to process definition $\eta(D)$, where $vars(\mathtt{T1}) \subseteq domain(\psi') \subseteq vars(\theta_a)$.*

Proof: See Appendix A.

The completeness proof is more complex compared to the soundness proof. Recall that the soundness proof establishes that for every transition derivable in MMC, there is an "equivalent" transition derivable from the π-calculus semantics. The notion of equivalence was formalized by showing that the corresponding process expressions in MMC and the π-calculus were related by a one-to-one mapping between MMC variables and π-calculus names.

However, when constructing a derivation using the symbolic semantics of Fig. 1 due to [21], the bound names of processes can be renamed arbitrarily due to alpha conversion; worse still, the same names can be used repeatedly, with different binding occurrences. This precludes us from relating process expressions in the π-calculus and MMC in the completeness proof by mapping π-calculus names to MMC variables. Hence we establish the completeness of MMC in two steps:

1. We first present what we call the *constructive symbolic semantics* (or simply the *constructive semantics*) for the π-calculus where (i) alpha conversion is limited to the application of the **Ide** inference rule of Fig. 1 and (ii) when applying the **Ide** rule, bound names are always renamed to *fresh* names not previously encountered in the derivation.[3] We show that when all process definitions are valid and closed, every derivation derivable in the original symbolic semantics – and indeed every sequence of transitions derivable in the original semantics – has an equivalent derivation in the constructive semantics.
2. We then show that every transition derivable in the constructive semantics has a corresponding transition derivable in MMC such that the names of π-calculus process expressions have a one-to-one mapping to the names of the corresponding MMC process expressions.

While the fundamental strategy used in the two-step proof could be combined into a single-step proof, the decomposition into two steps makes the proof clearer. The problems of alpha-conversion and renaming are tackled in the first step without reference to logic programming. The implicit renaming of variables carried out by resolution is brought to bear in the second step.

Constructive symbolic semantics. To define the constructive semantics of the π-calculus, we use sequents of the form $V_1, V_2 :\ P \overset{M,\alpha}{\longmapsto} Q$ to denote that a *valid* process P can make an α action under constraint M and then behave as process Q, where V_1 and V_2 are sets of names and $n(P) \in V_1$. The constructive semantics uses V_1 to record the set of names that have been encountered in a derivation and V_2 to supply fresh names whenever needed. The constructive semantics is given in Fig. 7. It assumes that all process definitions are valid and closed.

The constructive semantics makes the search for derivations more deterministic in the following sense. The

[3] We use the modifier "constructive" since the new semantics provides some guidance on *how* to derive a transition as opposed to just defining what a transition *is*.

$$\textbf{Prefix:}\ \frac{}{V_1, V_1 :\ \alpha.P \overset{true,\alpha}{\longmapsto} P}$$

$$\textbf{Sum:}\ \frac{V_1, V_2 :\ P_1 \overset{M,\alpha}{\longmapsto} Q_1}{V_1, V_2 :\ P_1 + P_2 \overset{M,\alpha}{\longmapsto} Q_1} \quad \frac{V_1, V_2 :\ P_2 \overset{M,\alpha}{\longmapsto} Q_2}{V_1, V_2 :\ P_1 + P_2 \overset{M,\alpha}{\longmapsto} Q_2}$$

$$\textbf{Ide:}\ \frac{V_2, V_3 :\ P' \overset{M,\alpha}{\longmapsto} Q}{V_1, V_3 :\ A(y_1, \dots, y_n) \overset{M,\alpha}{\longmapsto} Q}$$

$A(x_1, \dots, x_n) \overset{\text{def}}{=} P$ and $\{z_1, \dots, z_k\} = bn(P)$, $V_2 = V_1 \cup \{z'_1, \dots, z'_k\}$ such that $z'_i \notin V_1$ and z'_i are pairwise distinct, $\vartheta = \{z'_i/z_i | 1 \le i \le k\}$, and $P' = P\{y_1, \dots, y_n/x_1, \dots, x_n\}\vartheta$

$$\textbf{Match:}\ \frac{V_1, V_2 :\ P \overset{M,\alpha}{\longmapsto} Q}{V_1, V_2 :\ [x = y]P \overset{ML,\alpha}{\longmapsto} Q} \quad L = \begin{cases} \emptyset & \text{if } x = y; \\ x = y & \text{otherwise.} \end{cases}$$

$$\textbf{Par:}\ (1)\frac{V_1, V_2 :\ P_1 \overset{M,\alpha}{\longmapsto} Q_1}{V_1, V_2 :\ P_1 \mid P_2 \overset{M,\alpha}{\longmapsto} Q_1 \mid P_2} \quad bn(\alpha) \cap fn(P_2) = \emptyset$$

$$(2)\frac{V_1, V_2 :\ P_2 \overset{M,\alpha}{\longmapsto} Q_2}{V_1, V_2 :\ P_1 \mid P_2 \overset{M,\alpha}{\longmapsto} P_1 \mid Q_2} \quad bn(\alpha) \cap fn(P_1) = \emptyset$$

$$\textbf{Res:}\ \frac{V_1, V_2 :\ P \overset{M,\alpha}{\longmapsto} Q}{V_1, V_2 :\ (\nu y)P \overset{M,\alpha}{\longmapsto} (\nu y)Q} \quad y \notin n(M, \alpha)$$

$$\textbf{Com:}\ \frac{V_1, V_2 :\ P_1 \overset{M,y(z)}{\longmapsto} Q_1,\ V_2, V_3 :\ P_2 \overset{N,\overline{x}v}{\longmapsto} Q_2}{V_1, V_3 :\ P_1 \mid P_2 \overset{MNL,\tau}{\longmapsto} Q_1\{v/z\} \mid Q_2} \quad L = \begin{cases} \emptyset & \text{if } x = y; \\ x = y & \text{otherwise.} \end{cases}$$

$$\frac{V_1, V_2 :\ P_1 \overset{M,\overline{x}v}{\longmapsto} Q_1,\ V_2, V_3 :\ P_2 \overset{N,y(z)}{\longmapsto} Q_2}{V_1, V_3 :\ P_1 \mid P_2 \overset{MNL,\tau}{\longmapsto} Q_1 \mid Q_2\{v/z\}} \quad L = \begin{cases} \emptyset & \text{if } x = y; \\ x = y & \text{otherwise} \end{cases}$$

$$\textbf{Open:}\ \frac{V_1, V_2 :\ P \overset{M,\overline{x}y}{\longmapsto} Q}{V_1, V_2 :\ (\nu y)P \overset{M,\overline{x}\nu y}{\longmapsto} Q} \quad y \notin n(M, x)$$

$$\textbf{Close:}\ \frac{V_1, V_2 :\ P_1 \overset{M,y(w)}{\longmapsto} Q_1,\ V_2, V_3 :\ P_2 \overset{N,\overline{x}\nu w}{\longmapsto} Q_2}{V_1, V_3 :\ P_1 \mid P_2 \overset{MNL,\tau}{\longmapsto} (\nu w)(Q_1 \mid Q_2)} \quad L = \begin{cases} \emptyset & \text{if } x = y; \\ x = y & \text{otherwise.} \end{cases}$$

$$\frac{V_1, V_2 :\ P_1 \overset{M,\overline{x}\nu w}{\longmapsto} Q_1,\ V_2, V_3 :\ P_2 \overset{N,y(w)}{\longmapsto} Q_2}{V_1, V_3 :\ P_1 \mid P_2 \overset{MNL,\tau}{\longmapsto} (\nu w)(Q_1 \mid Q_2)} \quad L = \begin{cases} \emptyset & \text{if } x = y; \\ x = y & \text{otherwise.} \end{cases}$$

Fig. 7. Constructive semantics for π-calculus

symbolic semantics in [21] relies on the use of alpha-conversion at *any* point in the proof. The alternative semantics in [32] uses alpha-conversion whenever a bound name is "exposed" in the action of a transition, i.e., in the **Pref** and **Open** rules. Neither semantics specifies how the new names will be chosen when alpha-conversion is applied. Note that the choice of names may affect the success of the derivation search. Specifically, a poor choice of names may prevent us from applying the **Par** rule, which checks for a disjointness among bound names and free names of two process expressions. Thus, when attempting to construct a derivation for a given transition, the failure to expand the current sequent may be because

1. The transition cannot be derived at all, or
2. We chose a wrong sequent at an earlier step (note the nondeterminism due to the **Choice** and **Par** rules), or
3. We made a poor choice of names when applying alpha-conversion at an earlier step.

The problem of poor choice of names is relatively minor in theory since the choice of names does not affect the *derivability* of a transition but only the success of the currently chosen derivation path. However, it complicates the procedure for constructing derivations.

Our constructive semantics removes the nondeterminism due to choice of names during alpha-conversion. In particular, in our semantics,

1. Alpha-conversion is performed in the **Ide** rule, and
2. Since all process definitions are valid and closed, the choice of names for alpha-conversion does not affect the successful completion of a derivation.

The second feature above ensures that the check for distinctness of names in the **Par** rule will always succeed, thereby removing the need for the check.

The advantages of our semantics come at the cost of extra bookkeeping: each sequent carries with it the set of names seen so far in the derivation. When using this semantics in MMC, this bookkeeping, as well as the alpha-conversion in the **Ide** rule, are automatically performed by the logic-program evaluation engine.

Note that every derivation in the constructive semantics can be readily mapped to a derivation in the original symbolic semantics (Fig. 1). The proof that the constructive semantics is complete, i.e., that every derivation in the original symbolic semantics has an equivalent derivation in the constructive semantics, is more complicated and can be found in Appendix B. We can now establish the completeness of MMC with respect to the original semantics by giving a simpler proof that MMC is complete with respect to the constructive semantics.

Theorem 3 (Completeness of MMC). *Let D_π be a set of π-calculus process definitions and S be a process expression. Let $S \xrightarrow{M,\alpha} S'$ be a transition derivable in the symbolic semantics in Fig. 1. Let θ_c be a call substitution, P_r a Prolog variable, and ψ a function mapping variables in Prolog to π-calculus names such that $P_r\theta_c$ is a valid process expression and $f_\psi(P_r\theta_c) \equiv S$. Let U_1, U_2, U_3 be three distinct variables not in vars(θ_c). Then there is an answer* `trans(`P_r`,` U_1`,` U_2`,` U_3`)` $: \theta_c \to \theta_a$ *and a function ψ' that agrees with ψ on all free names of $P_r\theta_c$ such that $f_{\psi'}(U_1\theta_a) = \alpha$, $fc_{\psi'}(U_2\theta_a) = M$, and $f_{\psi'}(U_3\theta_a) \equiv S'$.*

Proof: See Appendix B.

Theorems 2 and 3, along with the requirement in MMC that all process definitions be valid and closed, allow us to inductively conclude that all derivable transitions for a given MMC process are correct with respect to the symbolic semantics of Fig. 1.

5 Model checking in the π-μ-calculus

In this section, we describe the MMC model checker for systems specified in the polyadic π-calculus and properties written in an expressive subset of the π-μ-calculus. The π-μ-calculus [14] extends the modal logic for the monadic π-calculus proposed in [30] in two ways: (i) it caters to the polyadic version of the π-calculus rather than the monadic version and (ii) it introduces least and greatest fixed-point operators. Like the logic of [30], 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.

The main difference between the logic we use, which we call the π-μ'-calculus, and the π-μ-calculus is that our logic does not have explicit quantifiers ($\exists$ and $\forall$). Rather, as described below, names are implicitly quantified in our logic. Opting for implicit as opposed to explicit quantification of names leads to a considerably simpler implementation of the MMC model checker, without a noticeable sacrifice of expressiveness.

The syntax of the π-μ'-calculus is now given. Let $\mathcal{F}$ denote the set of (non-fixed-point) formulas; $\mathcal{A}$ and $\mathcal{V}$ the sets of actions and names, respectively; $\mathcal{Z}$ the set of formula variables; and $\mathcal{E}$ the set of fixed-point equations defining the formula variables.

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

The operators `and` and `or` are boolean connectives; `diam` and `box` are modal operators; `lfp` and `gfp` are least and greatest fixed-point operators, respectively; and `pred` is used to encode the match operator.

Often, properties can be written more succinctly using additional *derived* modalities such as `diamSet`, `diamMinus`, `diamSetMinus`, etc; formulas using the derived modalities can always be rewritten using only the basic `diam` and `box` modalities. For instance, $\texttt{diamSet}(\{A_1, A_2, \ldots A_n\}, \varphi)$ represents $\texttt{or}(\texttt{diam}(A_1,\varphi),$ $\texttt{or}(\texttt{diam}(A_2,\varphi), \ldots \texttt{diam}(A_n,\varphi)\cdots))$; $\texttt{diamMinus}(A,\varphi)$ represents $\texttt{diamSet}(\{B|B \neq A\},\varphi)$; and $\texttt{diamSetMinus}(A,\varphi)$ represents $\texttt{diamSet}(\{B|B \notin A\},\varphi)$.

Names in a formula definition are implicitly quantified as follows. Names appearing in the left-hand side of a definition are called "formal parameters", and the remaining names are called "local names". For a local name X, let φ be a largest subformula of the right-hand side such that $\varphi = \texttt{diam}(A, F)$ and X occurs in A. Then X is existentially quantified, with its scope covering φ. Similarly, if $\varphi = \texttt{box}(A, F)$ is the largest formula in the right-hand side such that the local name X occurs in A, then X is universally quantified, with its scope covering φ. We require that every local name in a formula be quantified in this manner.

For example, the property `f(X)` states that there exists an input action on name X in all executions of the system:

```
fdef(f(X), lfp(or(diam(in(X,Y), tt),
           boxSetMinus({}, form(f(X)))))).
```

Note that name `X` is a parameter in `f(X)` and local name `Y` is existentially quantified since it occurs within a diamond modality.

Model checking the π-μ'-calculus requires the ability to handle inequality constraints of the form $X \neq Y$. Inequality constraints arise even when the logic and the process specification use only equalities, for instance, to record substitutions under which a transition is not enabled. Equality constraints are handled by the logic-programming system in MMC. In contrast, inequality constraints must be explicitly treated, either by representing them symbolically or by enumerating their consequences; i.e., $X \neq Y$ interpreted over a domain $\{a, b, c\}$ for X and Y can be enumerated as $X = a, Y = b, X = a, Y = c, \ldots$. While enumeration leads to poor performance, symbolic representation adds an additional layer of implementation (i.e., a constraint solver) with its attendant overhead. We avoid this overhead by imposing the condition that during model checking, the set of constraints (the constraint store) associated with transitions be empty. In practice, we have found that most π-calculus applications satisfy this condition.

The semantics of π-μ' can be readily derived from the semantics of the full logic given in [30]. From this semantics, we can also derive the tableau proof system for π-μ', which is given in Fig. 8. The tableau can be shown to be sound and complete with respect to the semantics of the π-μ'-calculus provided all free names in the process expression and formula in the model checking goal $P \vdash_\theta F$ are distinct.

The tableau treats only least fixed-point formulas but also handles negation. Therefore, greatest fixed-point formulas are handled using their dual least fixed-point forms, that is, 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, if $X = t_1 \in \theta$ and $X = t_2 \in \theta$, then $t_1 = t_2$. In Fig. 8, $mgu(t_1, t_2)$ 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 Fig. 9, which can be directly executed in the XSB 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 approach of [36]. The following theorem states the correctness of the model checker.

Theorem 4. *Let D be a set of process and formula definitions, S the logic program consisting of the clauses in Figs. 4 and 9, P a valid process expression, and F a formula. Also, let Pr and Fr be two distinct Prolog variables. Then there exists a δ mapping free names of P to free names of F such that $\texttt{models}(Pr, Fr) : \theta_c \to \theta_a$ is an answer derivable from the logic program $D \cup S$ if and only if $P \vdash_\delta F$ is a derivation in the tableau of Fig. 8, where $Pr\theta_c = P\delta$ and $Fr\theta_c = F\delta$.*

Proof: See Appendix C. $\square$

The `models` predicate implemented in MMC is an optimized version of the one shown in Fig. 9, aimed at reduc-

True
$$\frac{}{P \vdash_\theta \texttt{tt}} \qquad \theta \text{ is consistent}$$

Match
$$\frac{P \vdash_\theta F}{P \vdash_\theta \texttt{pred}((X = Y), F)} \qquad X\theta = Y\theta$$

And
$$\frac{P \vdash_\theta F_1 \quad P \vdash_\theta F_2}{P \vdash_\theta \texttt{and}(F1, F2)}$$

Not
$$\frac{P \nvdash_\theta F}{P \vdash_\theta \texttt{not}(F)}$$

Or
$$\frac{P \vdash_\theta F_1}{P \vdash_\theta \texttt{or}(F_1, F_2)} \qquad \frac{P \vdash_\theta F_2}{P \vdash_\theta \texttt{or}(F_1, F_2)}$$

Diam
$$\frac{P_1 \vdash_{\theta_1} F}{P \vdash_\theta \texttt{diam}(A, F)} \qquad \texttt{trans}(P, A_1, \_, P_1),\ \ \theta_1 = \theta \cup mgu(A, A_1)$$

Box
$$\frac{P_1 \vdash_{\theta_1} F, \ldots, P_n \vdash_{\theta_n} F}{P \vdash_\theta \texttt{box}(A, F)} \qquad \{(P_1, \theta_1), \ldots, (P_n, \theta_n)\} = \{(P', \theta \cup mgu(A, A')) \mid \texttt{trans}(P, A', \_, P')\}$$

Lfp
$$\frac{P \vdash_\theta F[\vec{V_1}/\vec{V}]}{P \vdash_\theta \texttt{form}(Z(\vec{V_1}))} \qquad \texttt{fdef}(Z(\vec{V}), \texttt{lfp}(F)) \text{ and}$$
$$P \vdash_\theta \texttt{form}(Z(\vec{V_1})) \text{ does not occur in the derivation before}$$

Fig. 8. Tableau rules for π-μ'-calculus

```
True   models(_P,tt).
Match  models(P,pred((X=Y),F)) :- X==Y, models(P,F).
And    models(P,and(F1,F2))    :- models(P,F1), models(P,F2).
Or     models(P,or(F1,F2))     :- models(P,F1); models(P,F2).
<A>    models(P,diam(A,F))     :- trans(P,A1,_,P1), A1=A, models(P1,F).
[A]    models(P,box(A,F))      :- forall(P1,trans(P,A,_,P1), models(P1,F)).
Neg    models(P,not(F))        :- sk_not(models(P,F)).
Lfp    models(P,form(D))       :- fdef(D,lfp(F)), models(P,F).
```

Fig. 9. Encoding of MMC's π-μ'-calculus model checker

ing the number of goals that will be tabled in XSB. The optimization is routine and is not shown.

6 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 [2]. In this section, we show that spi-calculus process expressions can be encoded in MMC using its support for the polyadic π-calculus. To capture message encryption and decryption, and to represent structured messages composed of multiple segments, we use terms built from names and two binary function symbols `encrypt` and `mesg`. The encryption and decryption primitives of the spi-calculus are encoded in MMC as follows.

Encryption of a message M with symmetric key K, denoted in the spi-calculus as $\{M\}_K$, is encoded in MMC by the term $\texttt{encrypt}(M, K)$. Such a term can be passed as a parameter to a process invocation or cast as a data item in an output action. Decryption is specified in the spi-calculus using a *case* expression: *case* L *of* $\{x\}_K$ *in* P behaves as $P[M/x]$ if message L is of the form $\{M\}_K$, and as the deadlocked process otherwise. This expression is encoded in MMC as $\texttt{unify(}L = \texttt{encrypt(}X\texttt{, }E\texttt{),}$ $\texttt{match(}E = K, P\texttt{))}$, where `unify` extracts the key portion of the message and `match` checks if the given key K matches the encryption key E.

For protocols that use asymmetric public/private keys, we introduce two unary function symbols `priv` and `pub`, and use $\texttt{priv(}K\texttt{)}$ and $\texttt{pub(}K\texttt{)}$ to denote the private and public keys of a key pair K. We also introduce a new process expression, $\texttt{code(}\mathit{Oper}, P\texttt{)}$, that performs operation *Oper* (written as a Prolog predicate) and then behaves as P. Operator $\texttt{complement(}K, K_1\texttt{)}$ is added to obtain the asymmetric key K_1 of K. A message M encrypted by principal A with B's public key K_B^{pu} is encoded as the term $t = \texttt{encrypt(}M\texttt{, pub(}K_B\texttt{))}$. A principal B attempting to decrypt the term t with a key K will use the expression $\texttt{unify(}t = \texttt{encrypt(}X\texttt{, }K\texttt{), code(complement(}K\texttt{, }K_1\texttt{), match(}$ $K_1 = \texttt{priv(}K_B\texttt{)}, \dots\texttt{)))}$, which will deadlock unless K_1 is the same as $\texttt{priv(}K_B\texttt{)}$. Similarly, a message M encrypted with a private key K_A^{pr} is encoded in MMC as $\texttt{encrypt(}$ $M\texttt{,priv(}K_A\texttt{))}$.

We can specify shared-key as well as public-key cryptographic protocols in MMC using the above encoding. Note that, as in the spi-calculus, the restriction operator `nu` can be used to generate fresh nonces and shared keys. For example, consider the fragment of a shared-key protocol given in Fig. 10. `N_a` is a nonce generated by A and `Key_ab` is a key shared by the principals A and B. The code fragment models a message that principal A sends to principal B with A's identity (in clear text) and the nonce `N_a` encrypted with `Key_ab`. `BAin` and `ABout` represent, respectively, the channels from which A receives and sends the message.

```
def(sender(MyID,In,Out,Key_ab,Key_ai),
  nu(N_a,
    pref(out(Out,mesg(MyID,encrypt(N_a,Key_ab))),
      ... /* A's behavior after sending the message */

def(receiver(MyID,In,Out,Key_ab,Key_bi),
  pref(in(In, M),
    unify((M = mesg(A_id,encrypt(Nonce,Key))), % Decompose M
      match((Key = Key_ab),
        ... /* B's behavior after receiving the message */

def(intruder(MyID,In_a,Out_a,In_b,Out_b,Key_ai,Key_bi,Mset,Nset),
  nu(N_i, pref(in(In_a,M),
      choice(
        /* Communicate with sender, decrypt the mesg and store the nonce.*/
        unify((M = mesg(ID,encrypt(Nonce,Key))),
          match((Key = Key_ai),code(store(Nset,(Nonce,ID),Nsetnew),
            proc(intruder(MyID,In_a,Out_a,In_b,Out_b,Key_ai,Key_bi,Mset,Nsetnew))))),
        choice(
          /* Overhear, store the message.*/
          pref(out(Out_b,M),code(store(Mset,M,Msetnew),
            proc(intruder(MyID,In_a,Out_a,In_b,Out_b,Key_ai,Key_bi,Msetnew,Nset)))),
          /* Replay an old message*/
          code(retrieve(Mset,OldMesg), pref(out(Out_b,OldMesg),
            proc(intruder(MyID,In_a,Out_a,In_b,Out_b,Key_ai,Key_bi,Mset,Nset)))))))).

def(proto(A,B,I),
  nu(BAin,nu(ABout,nu(ABin,nu(BAout,nu(Key_ab,nu(Key_ai,nu(Key_bi,
    par(proc(sender(A,BAin,ABout,Key_ab,Key_ai)),
      par(proc(intruder(I,ABout,BAin,BAout,ABin,Key_ai,Key_bi,[],[])),
        proc(receiver(B,ABin,BAout,Key_ab,Key_bi)))))))))))).
```

Fig. 10. Encoding transmission of encrypted messages in MMC

Systems with an intruder I are modeled such that all communications between principals go through the intruder, the behavior of which is specified by a recursive process definition. Note that the intruder model is different from that of spi-calculus, where intruders are not explicitly specified. When an intruder receives a message from a principal, it chooses to transmit, intercept, or fake the message transmission. The capabilities of the intruder to store and retrieve messages are encoded using a set of data structures 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`. In Fig. 10, we present only the communications between principals A and B, and the intruder's behaviors of overhearing and replaying messages. Other behaviors of the intruder can be modeled in a similar way. Notice that MMC does not support the modeling of unbounded sessions, which requires an unbounded message store. However, by specifying the number of rounds, MMC supports the modeling of multiple sessions.

Security properties such as authenticity can be expressed in our subset of the π-μ-calculus; in contrast, the spi-calculus specifies properties in terms of process equivalence. 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. Thus, when principal A takes part in the protocol run with B, it does an `out` action on channel `send_AB`; similarly, when A commits to a session 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. This style of authenticity property is called a *correspondence assertion* in [43]. For instance, an attack on principal A can be stated by the following formula:

$$F = \mu F.\langle\overline{\texttt{commit\_AB}}x\rangle true \vee \langle -\overline{\texttt{send\_BA}}y\rangle F\,.$$

Using MMC, we detected an authenticity violation, originally reported in [23], in the Needham–Schroeder protocol and several attacks in the BAN–Yahalom protocol [10]. We also verified several security properties of the modified Needham–Schroeder protocol (the Needham–Schroeder–Lowe protocol). In the case of BAN–Yahalom, the protocol we specified is a variant of the one in [10], while the attacks that we found are described in [37].

7 Benchmarking results

In this section, we present a variety of experimental results aimed at assessing MMC's performance. By MMC we expressly mean the tabled logic program comprising the operational semantics of the π-calculus (Fig. 4) and the tableau system for our subset of the π-μ-calculus, π-μ' (Fig. 9), with the optimizations described in Sects. 3 and 5 suitably applied. All reported performance data were obtained on an Intel Xeon 1.7-GHz machine with 2 GB RAM running Debian GNU/Linux 2.4.21 and XSB version 2.5 (optimal mode, slg-wam with batched scheduling, garbage collector turned off).

Handover, Needham–Schroeder, Needham–Schroeder–Lowe, and BAN–Yahalom Protocols. Table 2 contains performance data for MMC (the polyadic version) applied to the verification of four benchmark protocols: a simplified handover procedure from [31] and the Needham–Schroeder, Needham–Schroeder–Lowe, and BAN–Yahalom authenticity protocols. The handover protocol is specified in the π-calculus, while the Needham–Schroeder, Needham–Schroeder–Lowe, and BAN–Yahalom protocols are written in the spi-calculus. The attacks MMC found in the BAN–Yahalom protocol are described in [37]. One is the interleaving attack specified as the property "responder B commits to a session with initiator A before A commits to the session with B". Another is the replay attack specified as property "initiator A commits to a session with responder B before B participates in the protocol run with A".

Nonce generation in the specifications of the Needham–Schroeder, Needham–Schroeder–Lowe, and BAN–Yahalom protocols leads to a number of process expressions containing the restriction operator. These are candidates for the structural-congruence rule, which can be used to eliminate unused names. Although the structural-congruence rule is needed to ensure that MMC terminates

Table 2. Performance data for Handover, Needham–Schroeder, and BAN–Yahalom protocols

Benchmark	State	Trans	Formula	Time (s)	Mem (MB)
Handover	108	164	Deadlock freedom	0.05	1.28
			No data lost	0.09	2.49
Needham–Schroeder	167	287	Attack	0.02	0.70
Needham–Schroeder–Lowe	108	181	No attack	0.22	1.93
BAN–Yahalom	29 133	107 652	Interleaving attack	0.13	2.91
			Replay attack	0.75	13.41

for finite-control agents, repeated checks for whether the rule can be applied may impose substantial performance overhead. We can eliminate the use of the structural-congruence rule and still ensure termination whenever the scope of a restricted name does not fall within a recursion. This optimization can significantly reduce the model checking time. For example, to check if the BAN–Yahalom protocol has the property "responder B commits to a session with initiator A before A initiates the protocol run with B", the model checker takes 209.87 s to give the answer "No", requiring almost a complete traversal of the state space. Introducing the optimization leads to a threefold improvement in model checking time.

Monadic vs. polyadic versions of MMC. We compared the monadic and the polyadic versions of MMC by checking for deadlock freedom in chains of buffers of varying length, as specified in Fig. 11. Note that when a chain is deadlock free, the model checker traverses the entire state space of the system. Process $\texttt{lbuf}_i\texttt{(In,Out)}$ represents a chain of buffers of length i, each of which receives an input from channel `In` and outputs a value along channel `Out`. Process $\texttt{sbuf}_i$ wraps $\texttt{lbuf}_i$ with `gen`, a generator of values, and `sink`, a consumer of values.

Figure 12 contains the execution times taken by MMC (both the monadic and polyadic versions) for verifying the deadlock freedom property of process $\texttt{sbuf}_i(V)$ for different values of i. Note that the size of the transition system grows exponentially in the chain length, while MMC's time and space performance grows linearly with the number of transitions in the system. Furthermore, the polyadic version is slightly slower (but no worse than 7%) than the monadic version and consumes almost the same amount of memory as the monadic version.

MMC vs. MWB. We now compare the performance of MMC (the polyadic version) on process $\texttt{sbuf}_i(V)$ with that of the Mobility Workbench (MWB). The MWB has a model checker and a model prover [7] for model checking the polyadic π-calculus. It also provides a separate function, called "deadlocks", to check whether or not a process is deadlocked [42].

MMC is considerably faster than the MWB model checker. For example, MMC takes 0.02 s for checking deadlock freedom on a buffer of size 4, while MWB takes 0.54 s; on a buffer of size 8, MMC takes 0.83 s while MWB does not terminate in 13 h. For small chain lengths, MWB's deadlock-detection function is comparable in speed to the MMC model checker. However, for larger systems, MMC outperforms the "deadlock" function in MWB. For example, MMC takes 15.93 s on a buffer of size 12, while MWB's deadlock detection function takes 139.43 s. MWB's prover 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 this tool.

MMC vs. XMC. Table 3 shows the time performance of MMC and the first versions of XMC, i.e., those that did not employ the XMC compiler [16]. The examples, Rether (a real-time ethernet protocol), Sieve, and Leader, were selected from the XMC benchmark suite. Observe from the table that MMC is slightly slower than the first version of XMC. Two factors contribute to this. First, the

```
def(buf(In,Out),pref(in(In,X),pref(out(Out,X),proc(buf(In,Out))))).
def(lbuf1(In,Out),proc(buf(In,Out))).
def(lbuf_i(In,Out),nu(M, par(proc(buf(In,M)),proc(lbuf_{i-1}(M,Out))))).
def(sbuf_i(V),nu(M, nu(Out, par(proc(gen(M,V)),par(proc(lbuf_i(M,Out)),proc(sink(Out))))))).
def(gen(Out,V),pref(out(Out,V),proc(gen(Out,V)))).
def(sink(In),pref(in(In,X),proc(sink(In)))).
```

Fig. 11. Specification of a chain of buffers

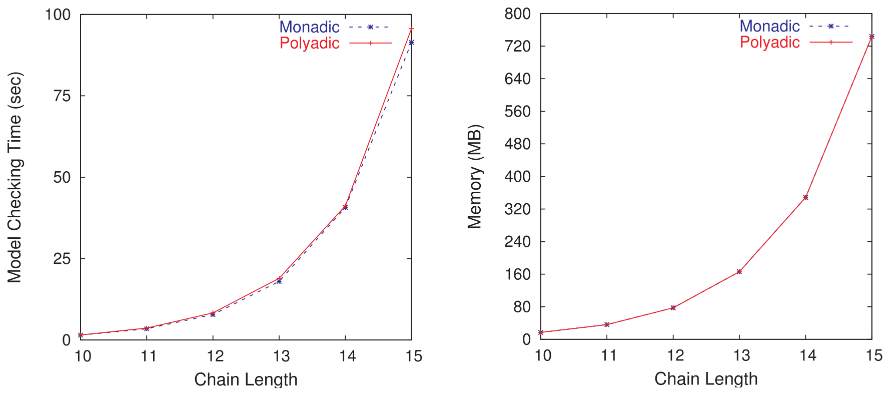

Fig. 12. Time and space performance of MMC for verifying deadlock freedom of chains of buffers

Table 3. Comparative performance of XMC and MMC

Benchmark	States	Trans	Property	Time (s)	
				XMC	MMC
rether	593	697	deadlock freedom	0.24	0.29
sieve(3)	615	1423	ae_finish	0.37	0.78
sieve(5)	4023	16091	ae_finish	5.01	8.25
leader(3)	67	88	ae_leader	0.03	0.04
leader(5)	864	2687	ae_leader	0.95	1.10
leader(7)	11 939	25 632	ae_leader	21.59	29.74

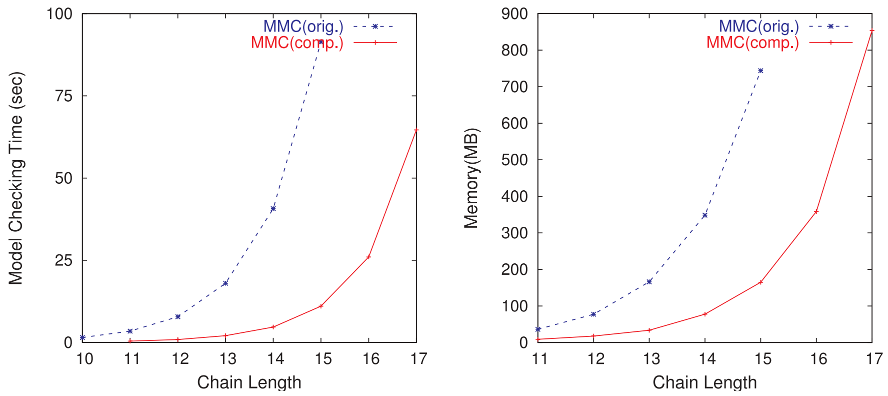

Fig. 13. Performance comparison between MMC with and without compilation on chains of buffers

check for structural congruence, despite the optimization discussed earlier, results in additional time. Second, the `Open` and `Close` clauses in relation `trans` are not needed in these examples, but MMC tries (and eventually fails) to resolve using these two rules.

In recent work (Yang et al. 2003, unpublished manuscript), we have implemented a compiler for MMC, along the lines of the XMC compiler [16], which, given a π-calculus process expression, produces a succinct representation of the process's symbolic transition system. The compiler is equipped with a number of optimizations that, as we have seen through extensive benchmarking, practically eliminate the overheads in MMC discussed above. These optimizations include implementation of a trie data structure for transition indexing, an AC unification-based notion of symmetry reduction, and a new state representation that eliminates futile attempts to apply the restriction operator. Figure 13 (taken from Yang et al., unpublished manuscript) illustrates the speedup enjoyed by the compiled version of MMC over the interpreted version on the example of Fig. 11 (checking for deadlock freedom in varying-length chains of buffers).

8 Related work

A number of analysis techniques have been developed for the π-calculus, and several of them have been incorporated in tools. The Mobility Workbench (MWB) [42] implemented the first model checker for the polyadic π-calculus [28] and the π-μ-calculus [14]. In addition to a model checker, the MWB consists of a bisimulation checker and a prover based on the sequent calculus [18]. Picasso [5] is a static analyzer for the π-calculus that focuses on checking secrecy of information such as process-level leaks and insecure communications. The Maude system is based on rewriting logic and supports cryptographic protocol analysis [15] and mobile computation [17]. Maude also handles may-testing equivalence of nonrecursive π-calculus processes [39]. MMC, in contrast, is a more traditional model checker for recursive mobile processes encoded in the π-calculus.

Regarding related work for the spi-calculus, Cryptyc [19] uses static type checking to find security violations such as secrecy or authenticity errors in cryptographic protocols specified in the spi-calculus. In [1, 8], techniques are proposed for verifying secrecy and authenticity of cryptographic protocols specified in an extension of the π-calculus; the intruder is modeled using Prolog rules. These techniques as well as Cryptyc 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, it can be used to verify other properties such as deadlock freedom as well as liveness properties such as lossless transmission. Interestingly, in this regard, Hüttel [20], along with Josva Kleist, Uwe Nestmann, and Björn Victor, has shown that the finite-control fragment of the spi-calculus is Turing-powerful. Hüttel has also shown that framed bisimilarity, an equivalence relation proposed by Abadi and Gordon [3], is decidable for finite (nonrecursive) spi-calculus processes.

Other related tools and techniques for analyzing security protocols besides those based on the spi-calculus include the FDR model checker for CSP [24]; the NRL Protocol Analyzer [25]; Paulson's inductive approach using the theorem prover Isabelle [33]; the Strand Space approach of [40], where a "strand" is a sequence of events representing either an execution by a legitimate party or a penetrator; the resolution-based verification method for cryptographic protocols of [9], which uses "tagging" to enforce termination, a syntactic transformation of messages that leaves attack-free executions invariant; Cohen's first-order invariant-based method for proving protocols secure against guessing attacks in an unbounded model, which has been implemented as an extension to the protocol verifier TAPS [12]; and the constraint-solving method of Miller and Shmatikov [26]. MMC also uses constraint-solving in its implementation of a model checker for the spi-calculus, where the constraints required are limited to equality and inequality constraints.

MMC is perhaps 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 that of the model checkers and equivalence checkers of the MWB reported in the literature [7, 41, 42]. Moreover, MMC's performance is even comparable to that of the first versions of XMC where, as in MMC, labeled transition systems were generated by interpreting process terms. Extensive benchmark results documenting MMC's performance are given in Sect. 7.

9 Conclusion

We have presented MMC, a practical model checker for the π-calculus. Through the use of tabled logic programming, MMC directly encodes the operational semantics of the π-calculus and (a subset of) the π-μ-calculus without unduly sacrificing performance. Key to this development is the similarity in the way resolution techniques handle variables in a logic program and the way the operational semantics of the π-calculus handles names. This similarity is exploited in MMC through the use of Prolog variables to represent π-calculus names.

We have also provided a detailed, formal proof of the correctness of our encoding, showing that MMC, given a π-calculus expression p, generates exactly the labeled transition for p as prescribed by the π-calculus's transitional semantics. Moreover, the high-level nature of the encoding makes it particularly versatile, allowing us to encode the monadic and polyadic versions of the π-calculus, as well as the spi-calculus for cryptographic protocols, in a single framework.

The results presented in this paper are a step toward routinely deploying model checkers for expressive process calculi with channel passing. The next step is to expand on the functionality of MMC. We are currently investigating extensions to the symbolic bisimulation checker developed using logic programming [6] to the transition systems derived from the π-calculus. We also plan to extend MMC to the full π-μ-calculus, taking advantage of recent developments to add lightweight constraint processing to tabled logic programming [13].

Another avenue of research is to extend MMC's capabilities beyond that of finite-control systems. A promising approach to this problem lies in the technique of [35], which uses logic-program transformations to seamlessly integrate induction-based proofs with model checking. This yields a model checker for parameterized systems (infinite families) of processes.

Acknowledgements. The authors are grateful to the anonymous reviewers for their valuable comments and suggestions. They would also like to thank Lenore Zuck for her much-appreciated editorial support. The research reported in this article was supported in part by NSF Grants EIA-9705998, CCR-9876242, CCR-9988155, and CCR-0205376; ONR Grant N000140110967; and ARO Grants DAAD190110003 and DAAD190110019.

References

1. Abadi M, Blanchet B (2002) Analyzing security protocols with secrecy types and logic programs. In: Proceedings of the 29th annual ACM SIGPLAN – SIGACT symposium on principles of programming languages (POPL 2002), Portland, OR, January 2002. ACM Press, New York, pp 33–44
2. Abadi M, Gordon AD (1997) A calculus for cryptographic protocols: the spi calculus. In: Proceedings of the 4th ACM conference on computer and communications security, Z urich, Switzerland, April 1997. ACM Press, New York, pp 36–47
3. Abadi M, Gordon AD (1998) A bisimulation method for cryptographic protocols. Nordic J Comput 5(4):267–303
4. Apt K (1990) Logic programming. In: Van Leeuwen J (ed) Handbook of theoretical computer science, vol B: Formal models and semantics. Elsevier/MIT Press, Amsterdam/Cambridge, MA, pp 493–574
5. Aziz B, Hamilton GW (2002) 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, July 2002
6. Basu S, Mukund M, Ramakrishnan CR, Ramakrishnan IV, Verma RM (2001) Local and symbolic bisimulation using tabled constraint logic programming. In: Proceedings of the international conference on logic programming, Cyprus, November 2001, pp 166–180
7. Beste FB (1998) 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, Kista, Sweden
8. Blanchet B (2002) From secrecy to authenticity in security protocols. In: Hermenegildo M, Puebla G (eds) Proceedings of the 9th international static analysis symposium, Madrid, September 2002. Lecture notes in computer science, vol 2477. Springer, Berlin Heidelberg New York, pp 242–259

9. Blanchet B, Podelski A (2003) Verification of cryptographic protocols: tagging enforces termination. In: Proceedings of the conference on foundations of software science and computation structures (FoSSaCS'03), Warsaw, Poland, April 2003. Lecture notes in computer science, vol 2620. Springer, Berlin Heidelberg New York, pp 136–152
10. Burrows M, Abadi M, Needham R (1996) A logic of authentication, from Proceedings of the Royal Society 426(1871), 1989. In: Stallings W (ed) Practical cryptography for data Internetworks. IEEE Press, New York
11. Chen W, Warren DS (1996) Tabled evaluation with delaying for general logic programs. J ACM 43(1):20–74
12. Cohen E (2002) Proving protocols safe from guessing. In: Proceedings of the workshop on foundations of computer security (FCS'02), Copenhagen, July 2002, pp 85–92
13. Cui B, Warren DS (2000) A system for tabled constraint logic programming. In: Proceedings of the 1st international conference on computational logic, London, UK, July 2000. Lecture notes in computer science, vol 1861. Springer, Berlin Heidelberg New York, pp 478–492
14. Dam M (2001) Proof systems for pi-calculus logics. Logic for concurrency and synchronisation, Kluwer, Dordrecht, pp 145–212
15. Denker G, Meseguer J (1998) Protocol specification and analysis in Maude. In: Proceedings of the workshop on formal methods and security protocols, Indianapolis, IN, June 1998
16. Dong Y, Ramakrishnan CR (1999) An optimizing compiler for efficient model checking. In: Formal methods for protocol engineering and distributed systems (FORTE). Proceedings of IFIP, Beijing, October 1999. Kluwer, Dordrecht, 156:241–256
17. Duran F, Eker S, Lincoln P, Meseguer J (2000) Principles of mobile Maude. In: Kotz D, Mattern F (eds) Proceedings of ASA/MA, Switzerland, September 2000. Lecture notes in computer science, vol 1882. Springer, Berlin Heidelberg New York, pp 73–85
18. Franzen T (1996) A theorem-proving approach to deciding properties of finite-control agents. Technical report, Swedish Institute of Computer Science, Kista, Sweden
19. Gordon A, Jeffrey ASA (2001) Authenticity by typing for security protocols. In: Proceedings of the IEEE computer security foundations workshop, Cape Breton, Novia Scotia, Canada, June 2001, pp 145–159
20. Hüttel H (2002) Deciding framed bisimilarity. In: Proceedings of the 4th international workshop on verification of infinite-state systems (INFINITY 2002), Brno, Czech Republic, August 2002
21. Lin H (1994) Symbolic bisimulation and proof systems for the π-calculus. Technical report, School of Cognitive and Computer Science, University of Sussex, Sussex, UK
22. Lloyd JW (1984) Foundations of logic programming. Springer, Berlin Heidelberg New York
23. Lowe G (1995) An attack on the Needham–Schroeder public-key authentication protocol. Inf Process Lett 56(3):131–133
24. Lowe G (1996) Breaking and fixing the Needham-Schroeder public-key protocol using FDR. Softw Concepts Tools 17:93–102
25. Meadows C (1996) The NRL protocol analyzer: an overview. J Logic Programm 26(2):113–131
26. Millen J, Shmatikov V (2001) Constraint solving for bounded process cryptographic protocol analysis. In: Proceedings of the 8th ACM conference on computer and communications security, Philadelphia, November 2001. ACM Press, New York
27. Milner R (1989) Communication and concurrency. In: International Series in Computer Science. Prentice-Hall, Upper Saddle River, NJ
28. Milner R (1993) The polyadic π-calculus: a tutorial. In: Bauer FL, Brauer W, Schwichtenberg H (eds) Logic and algebra of specification. Springer, Berlin Heidelberg New York, pp 203–246
29. Milner R, Parrow J, Walker D (1992) A calculus of mobile processes, parts I and II. Inf Comput 100(1):1–77
30. Milner R, Parrow J, Walker D (1993) Modal logics for mobile processes. Theor Comput Sci pp 149–171
31. Orava F, Parrow J (1992) An algebraic verification of a mobile network. Formal Aspects Comput 4:497–543
32. Parrow J (2001) An introduction to the π-calculus. In: Bergstra JA, Ponse A, Smolka SA (eds) Handbook of process algebra. Elsevier, Amsterdam
33. Paulson L (1998) The inductive approach to verifying cryptographic protocols. J Comput Secur 6:85–128
34. Ramakrishna YS, Ramakrishnan CR, Ramakrishnan IV, Smolka SA, Swift TW, Warren DS (1997) Efficient model checking using tabled resolution. In: Proceedings of the 9th international conference on computer-aided verification (CAV), Haifa, Israel, June 1997. Lecture notes in computer science, vol 1254. Springer, Berlin Heidelberg New York, pp 143–154
35. Roychoudhury A, Narayan Kumar K, Ramakrishnan CR, Ramakrishnan IV, Smolka SA (2000) Verification of parameterized systems using logic-program transformations. In: Proceedings of the 6th international conference on tools and algorithms for the construction and analysis of systems (TACAS), Berlin, March 2000. Lecture notes in computer science, vol 1785. Springer, Berlin Heidelberg New York, pp 172–187
36. Stirling C, Walker D (1991) Local model checking in the modal mu-calculus. Theor Comput Sci 89(1):161–177
37. Syverson P (1994) A taxonomy of replay attacks. In: Proceedings of the computer security foundations workshop VII, Franconia, NH, June 1994. IEEE Press, New York
38. Tamaki H, Sato T (1986) OLDT resolution with tabulation. In: Proceedings of the international conference on logic programming, London, UK, July 1986. MIT Press, Cambridge, MA, pp 84–98
39. Thati P, Sen K, Marti-oliet N (2002) An executable specification of asynchronous pi-calculus semantics and may testing in Maude 2.0. In: Proceedings of the 4th international workshop on rewriting logic and its applications, Italy, September 2002
40. Thayer Fabrega FJ, Herzog JC, Guttman JD (1999) Strand spaces: proving security protocol correct. J Comput Secur 7:191–230
41. Victor B (1995) The Mobility Workbench user's guide. Technical report, Department of Computer Systems, Uppsala University, Sweden
42. Victor B, Moller F (1994) The Mobility Workbench – a tool for the π-calculus. In: Dill D (ed) Proceedings of the 6th international conference on computer-aided verification (CAV '94), Stanford, CA, June 1994. Springer, Berlin Heidelberg New York
43. Woo TYC, Lam SS (1993) A semantic model for authentication protocols. In: Proceedings of the IEEE symposium on research in security and privacy, Oakland, CA, May 1993, pp 178–194
44. XSB (2001) The XSB logic programming system v2.4. `http://xsb.sourceforge.net`
45. Yang P, Ramakrishnan CR, Smolka SA (2002) Mobility Model Checker for the π-calculus. `http://www.cs.sunysb.edu/~lmc/mmc`

Appendices

A Proof of Theorem 2

Theorem 2 (Soundness). *Let D be a set of process definitions and P_r be a Prolog variable. Assume that $\mathtt{trans}(P_r, U_1, U_2, U_3) : \theta_c \to \theta_a$ is an answer derivable from the logic program $D \cup \mathtt{MMCtrans}$, where $var(\theta_c) \cap \{U_1, U_2, U_3\} = \emptyset$ and $P_r\theta_c$ is a valid process expression. Let $\mathtt{trans}(S1, A, M, T1) = \mathtt{trans}(P_r, U_1, U_2, U_3)\theta_a$. Then the following hold:*

1. **(Preservation of free and local names).** For every free name $F_v \in fn(P_r\theta_c)$, $F_v = F_v\theta_a$; For every local name $L_v \in ln(P_r\theta_c)$, $L_v = L_v\theta_a$.
2. **(Origin of free names).** For every name $V \in (fn(\mathtt{A}) \cup n(\mathtt{M}))$, $V \in fn(P_r\theta_c)$; For every name $V_1 \in fn(\mathtt{T1})$, $V_1 \in fn(P_r\theta_c)$ or $V_1 \in bn(\mathtt{A})$.

3. **(Origin of bound names).** For every name $Y \in (bn(\mathtt{A}) \cup bn(\mathtt{T1}))$, Y is a variable and ($Y \in bn(P_r\theta_c)$ or $Y \notin vars(\theta_c)$).
4. **(Distinctness of bound names).** $vars(bn(\mathtt{A})) \cap vars(bn(\mathtt{T1})) = \emptyset$.
5. **(Validity of destination).** `T1` is a valid process expression.
6. **(Soundness of transition).** Given a one-to-one function ψ mapping MMC variables to π-calculus names such that $vars(P_r\theta_c) \subseteq domain(\psi) \subseteq vars(\theta_c)$, there exists an extension ψ' of ψ such that $f_\psi(P_r\theta_c) \overset{fc_{\psi'}(\mathtt{M}),f_{\psi'}(\mathtt{A})}{\longrightarrow} f_{\psi'}(\mathtt{T1})$ is a derivation in the symbolic semantics of the π-calculus in Fig. 1 with respect to process definition $\eta(D)$, where $vars(\mathtt{T1}) \subseteq domain(\psi') \subseteq vars(\theta_a)$.

Proof. The proof is by induction on the depth of the derivation tree, derived using the resolution procedure of Fig. 2, for the given `trans` query. Parts 1 and 2 are easy to prove. We present the proofs for the remaining parts. For Part 4, we give the proof for the case where $Y \in bn(\mathtt{T1})$; the case where $Y \in bn(\mathtt{A})$ can be similarly proved. For Part 5, inductively assuming that the bound names of `T1` are variables (Part 3), we need only prove that $bn(\mathtt{T1}) \cap fn(\mathtt{T1}) = \emptyset$ and $bn(\mathtt{T1}) = ubn(\mathtt{T1})$.
The proof is split into different cases based on the clause of Fig. 4 used in the last step of the derivation. Clause names are written in bold font for easy identification, i.e., **Prefix** clause, **Sum** clause, etc. Inference rules from the symbolic semantics of Fig. 1 are referred to in the proof as the **Prefix** rule, the **Sum** rule, etc. Whenever we say "symbolic semantics" (or simply "semantics") in the proof, we are indeed referring to the inference rules of Fig. 1.

Pref: If $P_r\theta_c = \mathtt{pref(A,P)}$, then the derivation tree is of the following form:

$$\frac{\mathtt{true}: \ \theta_c\sigma \ \rightarrow \ \theta_c\sigma}{\mathtt{trans}(P_r, U_1, U_2, U_3): \ \theta_c \ \rightarrow \ \theta_c\sigma}$$

where

i. $\mathtt{trans(pref(}A'\mathtt{,}P'\mathtt{),}A'\mathtt{,true,}P'\mathtt{).}$ is a variant of the **Prefix** clause;
ii. $\{A', P'\} \cap (\{P_r, U_1, U_2, U_3\} \cup vars(\theta_c)) = \emptyset$;
iii. $\sigma = \{A' = \mathtt{A}, P' = \mathtt{P}, U_1 = A', U_2 = \mathtt{true}, U_3 = P'\}$.

Part 3: From iii, $bn(U_3\theta_c\sigma) = bn(\mathtt{P})$. Since `pref(A,P)` is a valid process expression, the bound names of `pref(A,P)` are variables and hence the bound names of `P` are variables. Further, since $bn(\mathtt{P}) \subseteq bn(\mathtt{pref(A,P)})$, Part 3 holds.
Part 4 can be directly proved by the fact that `pref(A,P)` is a valid process expression.
Part 5: Let $\mathtt{T1} = U_3\theta_c\sigma$. Then by iii, $\mathtt{T1} = \mathtt{P}$. We need to prove that `P` is a valid process expression, which follows from the fact that `pref(A,P)` is a valid process expression and from Proposition 1.
Part 6: Assume that $\mathtt{trans(}P_r\mathtt{,}\ U_1\mathtt{,}\ U_2\mathtt{,}\ U_3\mathtt{)}: \theta_c \rightarrow \theta_c\sigma$ is derivable from the above derivation tree in the logic program $D \cup \mathtt{MMCtrans}$. From the **Prefix** rule, for any one-to-one function ψ mapping MMC variables to π-calculus names, $f_\psi(\mathtt{A}).f_\psi(\mathtt{P}) \overset{true,f_\psi(\mathtt{A})}{\longrightarrow} f_\psi(\mathtt{P})$ is an inferable transition in the symbolic semantics. Since $vars(\mathtt{pref(A,P)}) \subseteq domain(\psi) \subseteq vars(\theta_c)$, $vars(\mathtt{P}) \subseteq domain(\psi) \subseteq vars(\theta_c\sigma)$. Since $P_r\theta_c = \mathtt{pref(A,P)}$, $U_1\theta_a = \mathtt{A}$, $U_2\theta_a = \mathtt{true}$, and $U_3\theta_a = \mathtt{P}$, $f_\psi(P_r\theta_c), \overset{fc_\psi(U_2\theta_a),f_\psi(U_1\theta_a)}{\longrightarrow} f_\psi(U_3\theta_a)$ is also an inferable transition in the symbolic semantics.

Sum: If $P_r\theta_c = \mathtt{choice(P1,P2)}$, then the derivation tree is of the following form:

$$\frac{\begin{array}{c}\vdots\\ \hline \mathtt{trans}(P_1',A',M',Q_1'): \ \theta_c\sigma \rightarrow \theta_a\end{array}}{\mathtt{trans}(P_r,U_1,U_2,U_3): \ \theta_c \rightarrow \theta_a}$$

where

i. $\mathtt{trans(choice(}P_1'\mathtt{,}P_2'\mathtt{),}\ A'\mathtt{,}\ M'\mathtt{,}\ Q_1'\mathtt{)}$:- $\mathtt{trans(}P_1'\mathtt{,}\ A'\mathtt{,}\ M'\mathtt{,}\ Q_1'\mathtt{).}$ is a variant of the first of the two **Sum** clauses. The symmetric clause of **Sum** can be similarly proved;
ii. $\{P_1', P_2', A', M', Q_1'\} \cap (\{P_r, U_1, U_2, U_3\} \cup vars(\theta_c)) = \emptyset$;
iii. $\sigma = \{P_1' = \mathtt{P1}, P_2' = \mathtt{P2}, U_1 = A', U_2 = M', U_3 = Q_1'\}$.

Part 3: Let $\mathtt{T1} = U_3\theta_a$. Then from iii, $\mathtt{T1} = Q_1'\theta_a$. We need to prove that for every $Y \in bn(Q_1'\theta_a)$, Y is a variable, and ($Y \in bn(\mathtt{choice(P1,P2)})$ or $Y \notin vars(\theta_c)$). By the induction hypothesis, for every $Y \in bn(Q_1'\theta_a)$, Y is a variable, and ($Y \in bn(P_1'\theta_c\sigma)$ or $Y \notin vars(\theta_c\sigma)$). Since $bn(P_1'\theta_c\sigma) \subseteq bn(\mathtt{choice(P1,P2)})$), and $vars(\theta_c) \subset vars(\theta_c\sigma)$, 3 holds.
Part 4 can be proved directly from the induction hypothesis.
Part 5: Let $\mathtt{T1} = U_3\theta_a$. Then by iii, $\mathtt{T1} = Q_1'\theta_a$. We need to show that `T1` is a valid process expression. Since `choice(P1,P2)` is a valid process expression, `P1` is a valid process expression, i.e., $P_1'\theta_c\sigma$ is a valid process expression. By the induction hypothesis, $Q_1'\theta_a$ is a valid process expression.
Part 6: Suppose that $\mathtt{trans}(P_r, U_1, U_2, U_3): \theta_c \rightarrow \theta_a$ is an answer derived from the above derivation tree in the logic program $D \cup \mathtt{MMCtrans}$. By the induction hypothesis, given a one-to-one function ψ where $vars(P_r\theta_c) \subseteq domain(\psi) \subseteq vars(\theta_c)$, there exists an extension ψ' of ψ such that $f_\psi(P_1'\theta_c\sigma) \overset{fc_{\psi'}(M'\theta_a),f_{\psi'}(A'\theta_a)}{\longrightarrow} f_{\psi'}(Q_1'\theta_a)$ is an inferable transition in the symbolic semantics where $vars(Q_1'\theta_a) \subseteq domain(\psi') \subseteq vars(\theta_a)$. From the **Sum** rule, $f_\psi(P_1'\theta_c\sigma) + f_\psi(P_2'\theta_c\sigma) \overset{fc_{\psi'}(M'\theta_a),f_{\psi'}(A'\theta_a)}{\longrightarrow} f_{\psi'}(Q_1'\theta_a)$ is an inferable transition in the symbolic semantics. Since $P_r\theta_c = \mathtt{choice}(P_1'\theta_c\sigma, P_2'\theta_c\sigma)$, $U_1\theta_a = A'\theta_a$, $U_2\theta_a = M'\theta_a$, $U_3\theta_a = Q_1'\theta_a$, $f_\psi(P_r\theta_c) \overset{fc_{\psi'}(U_2\theta_a),f_{\psi'}(U_1\theta_a)}{\longrightarrow} f_{\psi'}(U_3\theta_a)$ is also an inferable transition in the symbolic semantics.

Par: If $P_r\theta_c = \texttt{par(P1,P2)}$, then the derivation tree is of the following form:

$$\cfrac{\cfrac{\vdots}{\texttt{trans}(P_1',A',M',Q_1')\texttt{:}\ \theta_c\sigma \to \theta_a}}{\texttt{trans}(P_r,U_1,U_2,U_3)\texttt{:}\ \theta_c \to \theta_a}$$

where

i. $\texttt{trans(par(}P_1'\texttt{,}P_2'\texttt{), } A'\texttt{, } M'\texttt{, par(}Q_1'\texttt{,}P_2'\texttt{))}$:- $\texttt{trans(}P_1'\texttt{, } A'\texttt{, } M'\texttt{, } Q_1'\texttt{).}$ is a variant of the first of the two **Par** clauses. The symmetric **Par** clause can be proved along the same lines;
ii. $\{P_1', P_2', A', M', Q_1'\} \cap (\{P_r, U_1, U_2, U_3\} \cup \mathit{vars}(\theta_c)) = \emptyset$;
iii. $\sigma = \{P_1' = \texttt{P1},\ P_2' = \texttt{P2},\ U_1 = A',\ U_2 = M',\ U_3 = \texttt{par}(Q_1', P_2')\}$.

Part 3: Let $\texttt{T1} = U_3\theta_a$. Then from iii, $\texttt{T1} = \texttt{par}(Q_1', P_2')\theta_a$. We need to show that for every $Y \in \mathit{bn}(\texttt{par}(Q_1', P_2')\theta_a)$, Y is a variable, and $(Y \in \mathit{bn}(\texttt{par(P1,P2)})$ or $Y \notin \mathit{vars}(\theta_c))$. Since $\texttt{par(}P_1'\theta_c\sigma\texttt{, }P_2'\theta_c\sigma\texttt{)}$ is a valid process expression, $\mathit{bn}(P_1'\theta_c\sigma) \cap \mathit{bn}(P_2'\theta_c\sigma) = \emptyset$, and hence $P_2'\theta_a = P_2'\theta_c\sigma = \texttt{P2}$. From Proposition 1, $P_2'\theta_a$ is a valid process expression, and hence by Definition 3 the bound names of $P_2'\theta_a$ are variables. By the induction hypothesis, for every $Y \in \mathit{bn}(Q_1'\theta_a)$, Y is a variable and $(Y \in \mathit{bn}(P_1'\theta_c\sigma)$ or $Y \notin \mathit{vars}(\theta_c\sigma))$. Thus, the bound names of T1 are variables. Since $\mathit{bn}(P_1'\theta_c\sigma) \subseteq \mathit{bn}(\texttt{par(P1,P2)}))$ and $\mathit{bn}(P_2'\theta_a) \subseteq \mathit{bn}(\texttt{par(P1,P2)}))$ and $\mathit{vars}(\theta_c) \subset \mathit{vars}(\theta_c\sigma)$, 3 holds.
Part 4 can be proved directly from the induction hypothesis.
Part 5: Let $\texttt{T1} = U_3\theta_a$. Then from iii, $\texttt{T1} = \texttt{par}(Q_1', P_2')\theta_a$. We need to show that T1 is a valid process expression. Since $\texttt{par(P1,P2)}$ is a valid process expression, P1 is a valid process expression, i.e., $P_1'\theta_c\sigma$ is a valid process expression. By the induction hypothesis, $Q_1'\theta_a$ is a valid process expression. We need to show that (1) $\mathit{fn}(Q_1'\theta_a) \cap \mathit{bn}(P_2'\theta_a) = \emptyset$; (2) $\mathit{bn}(Q_1'\theta_a) \cap \mathit{fn}(P_2'\theta_a)$; and (3) $\mathit{bn}(Q_1'\theta_a) \cap \mathit{bn}(P_2'\theta_a) = \emptyset$. We give the proof of (1). Assume that $X \in \mathit{fn}(Q_1'\theta_a)$. Then, by Part 2, $X \in \mathit{fn}(P_1'\theta_c\sigma)$ or $X \in \mathit{bn}(A'\theta_a)$. Assume that $Y \in \mathit{bn}(P_2'\theta_a)$. Then, as we discussed above, $P_2'\theta_c\sigma = P_2'\theta_a$ and hence $Y \in \mathit{bn}(P_2'\theta_c\sigma)$. If $X \in \mathit{fn}(P_1'\theta_c\sigma)$, then since $\texttt{par}(P_1', P_2')\theta_c\sigma$ is a valid process expression, $X \neq Y$. If $X \in \mathit{bn}(A'\theta_a)$, then by Part 3, $X \in \mathit{bn}(P_1'\theta_c\sigma)$ or $X \notin \mathit{vars}(\theta_c\sigma)$. In the former case, since $\texttt{par}(P_1',P_2')\theta_c\sigma$ is a valid process expression, $X \neq Y$. In the latter case, since $Y \in \mathit{vars}(\theta_c\sigma)$, $X \neq Y$.
Part 6: Suppose that $\texttt{trans}(P_r, U_1, U_2, U_3) : \theta_c \to \theta_a$ is an answer derived from the above derivation tree in the logic program $D \cup \texttt{MMCtrans}$. Given a one-to-one function ψ where $\mathit{vars}(P_r\theta_c) \subseteq \mathit{domain}(\psi) \subseteq \mathit{vars}(\theta_c)$, since $\mathit{vars}(P_1'\theta_c\sigma) \subseteq \mathit{vars}(P_r\theta_c)$, $\mathit{vars}(P_1'\theta_c\sigma) \subseteq \mathit{domain}(\psi) \subseteq \mathit{vars}(\theta_c)$. By the induction hypothesis, there exists an extension ψ' of ψ such that $f_\psi(P_1'\theta_c\sigma) \xrightarrow{fc_{\psi'}(M'\theta_a), f_{\psi'}(A'\theta_a)} f_{\psi'}(Q_1'\theta_a)$ is an inferable transition in the symbolic semantics where $\mathit{vars}(Q_1'\theta_a) \subseteq \mathit{domain}(\psi') \subseteq \mathit{vars}(\theta_a)$. Since $P_2'\theta_a = P_2'\theta_c\sigma$ and $\mathit{vars}(P_2'\theta_c\sigma) \subseteq \mathit{domain}(\psi)$, $\mathit{vars}(\texttt{par}(Q_1', P_2')\theta_a) \subseteq \mathit{domain}(\psi') \subseteq \mathit{vars}(\theta_a)$. By the fact that $P_r\theta_c$ is a valid process expression and by Part 3, we can infer that $\mathit{bn}(A'\theta_a) \cap \mathit{fn}(P_2'\theta_c\sigma) = \emptyset$. So we do not need to explicitly check this side condition in MMC. Because ψ' is a one-to-one function, the side condition of the **Par** rule $\mathit{bn}(f_{\psi'}(A'\theta_a)) \cap \mathit{fn}(f_{\psi'}(P_2'\theta_c\sigma)) = \emptyset$ holds. Therefore, $f_\psi(P_1'\theta_c\sigma) \mid f_\psi(P_2'\theta_c\sigma) \xrightarrow{fc_{\psi'}(M'\theta_a), f_\psi'(A'\theta_a)} f_{\psi'}(Q_1'\theta_a)$ is an inferable transition in the symbolic semantics. Since $P_r\theta_c = \texttt{par}(P_1'\theta_c\sigma, Q_1'\theta_c\sigma)$, $U_1\theta_a = A'\theta_a$, $U_2\theta_a = M'\theta_a$, $U_3\theta_a = Q_1'\theta_a$, $f_\psi(P_r\theta_c) \xrightarrow{fc_{\psi'}(U_2\theta_a), f_{\psi'}(U_1\theta_a)} f_{\psi'}(U_3\theta_a)$ is an inferable transition in the symbolic semantics.

Ide: If $P_r\theta_c = \texttt{proc(PN)}$, then the derivation tree is of the following form:

$$\cfrac{\cfrac{\cfrac{\texttt{true:}\ \theta_c\sigma\sigma_1 \to \theta_c\sigma\sigma_1}{\texttt{def}(PN'\texttt{,}P')\texttt{:}\ \theta_c\sigma \to \theta_c\sigma\sigma_1,} \quad \cfrac{\vdots}{\texttt{trans}(P'\texttt{,}A'\texttt{,}M'\texttt{,}Q')\texttt{:}\ \theta_c\sigma\sigma_1 \to \theta_a}}{\texttt{def}(PN'\texttt{,}P')\texttt{, trans}(P'\texttt{,}A'\texttt{,}M'\texttt{, }Q')\texttt{:}\ \theta_c\sigma \to \theta_a}}{\texttt{trans}(P_r\texttt{,}U_1\texttt{,}U_2\texttt{,}U_3)\texttt{:}\ \theta_c \to \theta_a}$$

where

i. $\texttt{trans(proc(}PN'\texttt{),}A'\texttt{,}M'\texttt{,}Q'\texttt{)}$:- $\texttt{def(}PN'\texttt{, }P'\texttt{),}$ $\texttt{trans(}P'\texttt{,}A'\texttt{,}M'\texttt{,}Q'\texttt{).}$ is a variant of the **Ide** clause;
ii. $(\mathit{vars}(PN') \cup \{A', M', P', Q'\}) \cap (\{P_r, U_1, U_2, U_3\} \cup \mathit{vars}(\theta_c)) = \emptyset$;
iii. $\sigma = \mathit{mgu}((\texttt{PN}, U_1, U_2, U_3), (PN', A', M', Q'))$;
iv. $\texttt{def(PN}''\texttt{,P}''\texttt{)}$ is a variant of the definition of PN;
v. $(\mathit{vars}(\texttt{P}'') \cup \mathit{vars}(\texttt{PN}'')) \cap (\{P', PN'\} \cup \mathit{vars}(\theta_c\sigma)) = \emptyset$;
vi. $\sigma_1 = \mathit{mgu}((\texttt{PN}'', \texttt{P}''), (PN', P'))$.

We now show that $P'\theta_c\sigma\sigma_1$ is a valid process expression. By iii and vi, $P'\theta_c\sigma\sigma_1 = \texttt{P}''\sigma_1$. So it suffices to show that $\texttt{P}''\sigma_1$ is a valid process expression. Recall that all process definitions are valid in MMC and $\mathit{vars}(\texttt{PN}'')$ are the only free names occurring in $\texttt{P}''$. From v we can infer that all bound names of $\texttt{P}''\sigma_1$ are variables and $\mathit{bn}(\texttt{P}''\sigma_1) \cap \mathit{vars}(\theta_c\sigma) = \emptyset$. By iii and vi, $\texttt{P}''\sigma_1$ is a valid process expression.
Part 3: Let $\texttt{T1} = U_3\theta_a$. We need to show that for every $Y \in \mathit{bn}(\texttt{T1})$, Y is a variable and $(Y \in \mathit{bn}(\texttt{PN})$ or $Y \notin \mathit{vars}(\theta_c))$. Since PN does not contain bound names, we need to show that Y is a variable and $Y \notin \mathit{vars}(\theta_c)$. From iii, $\texttt{T1} = Q'\theta_a$. By the induction hypothesis, Y is a variable and $(Y \in \mathit{bn}(P'\theta_c\sigma\sigma_1)$, or $V \notin \mathit{vars}(\theta_c\sigma\sigma_1))$. If $Y \in \mathit{bn}(P'\theta_c\sigma\sigma_1)$, then since $P'\theta_c\sigma\sigma_1 = \texttt{P}''\sigma_1$ and $\mathit{bn}(\texttt{P}''\sigma_1) \cap \mathit{vars}(\theta_c) = \emptyset$, $Y \notin \mathit{vars}(\theta_c)$. If $Y \notin \mathit{vars}(\theta_c\sigma\sigma_1)$, then since $\mathit{vars}(\theta_c) \subset \mathit{vars}(\theta_c\sigma\sigma_1)$, $Y \notin \mathit{vars}(\theta_c)$.
Part 4 can be proved directly from the induction hypothesis.
Part 5: Let $\texttt{T1} = U_3\theta_a$. We need to show that T1 is a valid process expression. From the induction hypothesis, $Q'\theta_a$ is a valid process expression. Since $\texttt{T1} = Q'\theta_a$, T1 is a valid process expression.
Part 6: Suppose that $\texttt{trans}(P_r, U_1, U_2, U_3) : \theta_c \to \theta_a$ is an answer derived from the above derivation tree in the logic program $D \cup \texttt{MMCtrans}$. We can construct an extension ψ_1 of ψ such that ψ_1 maps all bound variables

of $P'\theta_c\sigma\sigma_1$ to π-calculus names that do not occur in *range*(ψ). Since *vars*$(P_r\theta_c) \subseteq$ *domain*$(\psi) \subseteq$ *domain*(θ_c) and *fn*$(P'\theta_c\sigma\sigma_1) =$ *fn*$(P_r\theta_c)$, *vars*$(P'\theta_c\sigma\sigma_1) \subseteq$ *domain*$(\psi_1) \subseteq$ *vars*$(\theta_c\sigma\sigma_1)$. By the induction hypothesis, there exists an extension ψ' of ψ_1 such that $f_{\psi_1}(P'\theta_c\sigma\sigma_1) \xrightarrow{fc_{\psi'}(M'\theta_a),f_{\psi'}(A'\theta_a)} f_{\psi'}(Q'\theta_a)$ is an inferable transition in the symbolic semantics where *vars*$(Q'\theta_a) \subseteq$ *domain*$(\psi') \subseteq$ *vars*(θ_a). Assume that $\texttt{PN} = \texttt{r}(Y_1, \ldots, Y_n)$ and the process definition for PN is $\texttt{def(r(}X_1, \ldots, X_n\texttt{)},P\texttt{)}$; then $r(\eta(X_1), \ldots, \eta(X_n))) \overset{\text{def}}{=} \eta(P)$ is the corresponding π-calculus process definition. Since $P'\theta_c\sigma\sigma_1$ is alpha-equivalent to $P\{Y_1, \ldots, Y_n/X_1, \ldots, X_n\}$ and η is a one-to-one function, and by the way that ψ_1 is constructed, $f_{\psi_1}(P'\theta_c\sigma\sigma_1)$ is alpha-equivalent to $\eta(P)\{f_\psi(Y_1), \ldots, f_\psi(Y_n)/\eta(X_1), \ldots, \eta(X_n)\}$. Therefore, $\eta(P)\delta \xrightarrow{fc_{\psi'}(M'\theta_a),f_{\psi'}(A'\theta_a)} f_{\psi'}(Q'\theta_a)$ is an inferable transition in the symbolic semantics where $\delta = \{f_\psi(Y_1), \ldots, f_\psi(Y_n)/\eta(X_1), \ldots, \eta(X_n)\}$. Thus, $f_\psi(\texttt{proc}(\texttt{r}(Y_1, \ldots, Y_n))) \xrightarrow{fc_{\psi'}(M'\theta_a),f_{\psi'}(A'\theta_a)} f_{\psi'}(Q'\theta_a)$ is also an inferable transition in the symbolic semantics. Finally, since $P_r\theta_c = \texttt{proc}(\texttt{r}(Y_1, \ldots, Y_n))$, $U_1\theta_a = A'\theta_a$, $U_2\theta_a = M'\theta_a$, and $U_3\theta_a = Q'\theta_a$, $f_\psi(P_r\theta_c) \xrightarrow{fc_{\psi'}(U_2\theta_a),f_{\psi'}(U_1\theta_a)} f_{\psi'}(U_3\theta_a)$ is an inferable transition in the symbolic semantics.

Match: If $P_r\theta_c = \texttt{match(X=Y,P)}$, then we obtain the following two derivation trees:

$$(1)\quad \dfrac{\dfrac{X'{=}{=}Y'{:}\theta_c\sigma \to \theta_c\sigma,\ \dfrac{\vdots}{\texttt{trans}(P',A',ML',Q'){:}\theta_c\sigma \to \theta_a}}{X'{=}{=}Y',\ \texttt{trans}(P',A',ML',Q'){:}\theta_c\sigma \to \theta_a}}{\texttt{trans}(P_r,U_1,U_2,U_3){:}\theta_c \to \theta_a}$$

$$(2)\quad \dfrac{\dfrac{X'\backslash{=}{=}Y'{:}\theta_c\sigma_1 \to \theta_c\sigma_1,\ \dfrac{\vdots}{\texttt{trans}(P',A',M',Q'){:}\theta_c\sigma_1 \to \theta_{a1}}}{X'\backslash{=}{=}Y',\texttt{trans}(P',A',M',Q'){:}\theta_c\sigma_1 \to \theta_{a1}}}{\texttt{trans}(P_r,U_1,U_2,U_3){:}\theta_c \to \theta_{a1}}$$

where

i. $\texttt{trans(match(}X' = Y'\texttt{,}P'\texttt{)},\ A',\ ML',\ Q'\texttt{)}$:– $X' == Y'$, $\texttt{trans(}P',\ A',\ ML',\ Q'\texttt{)}$. and $\texttt{trans(}$ $\texttt{match(}X' = Y',\ P'\texttt{)},\ A',\ (X' = Y',M'),\ Q'\texttt{)}$:– $X'\backslash == Y'$, $\texttt{trans(}P',\ A',\ M',\ Q'\texttt{)}$. are variants of the **Match** clause;

ii. $\{X',\ Y',\ P',\ A',\ ML',\ Q'\} \cap (\{P_r,\ U_1,\ U_2,\ U_3\} \cup$ *vars*$(\theta_c)) = \emptyset$ [derivation tree (1)]; $\{X', Y', P', A', M', Q'\} \cap (\{P_r,\ U_1,\ U_2,\ U_3\} \cup$ *vars*$(\theta_c)) = \emptyset$ [derivation tree (2)];

iii. $\sigma = \{X' = \texttt{X}, Y' = \texttt{Y}, P' = \texttt{P}, U_1 = A', U_2 = ML', U_3 = Q'\}$ [derivation tree (1)]; $\sigma_1 = \{X' = \texttt{X}, Y' = \texttt{Y}, P' = \texttt{P}, U_1 = A', U_2 = (X' = Y', M'), U_3 = Q'\}$ [derivation tree (2)].

Parts 3, 4, and 5 for these two derivation trees can be proved directly by the induction hypothesis.

Part 6: Suppose that $\texttt{trans(}P_r,\ U_1,\ U_2,\ U_3\texttt{)}{:}\ \theta_c \to \theta_a$ is derived from derivation tree (1) in the logic program $D \cup \texttt{MMCtrans}$where $X'\theta_c\sigma$ is identical to $Y'\theta_c\sigma$. By the induction hypothesis, given a one-to-one function ψ where *vars*$(P_r\theta_c) \subseteq$ *domain*$(\psi) \subseteq$ *vars*(θ_c), there exists an extension ψ' of ψ such that $f_\psi(P'\theta_c\sigma) \xrightarrow{fc_{\psi'}(ML'\theta_a),f_{\psi'}(A'\theta_a)} f_{\psi'}(Q'\theta_a)$ is an inferable transition in the symbolic semantics where *vars*$(Q'\theta_a) \subseteq$ *domain*$(\psi') \subseteq$ *vars*(θ_a). Since ψ is a one-to-one function, $f_\psi(X'\theta_c\sigma)$ is equal to $f_\psi(Y'\theta_c\sigma)$ and hence $[f_\psi(X'\theta_c\sigma) = f_\psi(Y'\theta_c\sigma)]f_\psi(P'\theta_c\sigma) \xrightarrow{fc_{\psi'}(ML'\theta_a),f_{\psi'}(A'\theta_a)} f_{\psi'}(Q'\theta_a)$ is an inferable transition in the symbolic semantics. Since $P_r\theta_c = \texttt{match}(X' = Y', P')\theta_c\sigma$, $U_1'\theta_a = A'\theta_a$, $U_2'\theta_a = ML'\theta_a$, and $U_3'\theta_a = Q'\theta_a$, $f_\psi(P_r\theta_c) \xrightarrow{fc_{\psi'}(U_2\theta_a),f_{\psi'}(U_1\theta_a)} f_{\psi'}(U_3\theta_a)$ is also an inferable transition in the symbolic semantics. The case where $\texttt{trans(}P_r,\ U_1,\ U_2,\ U_3\texttt{)}{:}\ \theta_c \to \theta_a$ is derived from derivation tree (2) can be similarly proved.

Res: If $P_r\theta_c = \texttt{nu(Y,P)}$, then the derivation tree is of the following form:

$$\dfrac{\dfrac{\dfrac{\vdots}{\texttt{trans}(P',A',M',Q'){:}\ \theta_c\sigma \to \theta_1,\ \textit{side cond}{:}\theta_1{\to}\theta_a}}{\texttt{trans}(P',A',M',Q'),\ \textit{side cond}{:}\ \theta_c\sigma{\to}\theta_a}}{\texttt{trans}(P_r,U_1,U_2,U_3){:}\ \theta_c \to \theta_a}$$

where

i. $\texttt{trans(nu(}Y',P'\texttt{)},A',M'\texttt{,nu(}Y',Q'\texttt{))}$:– $\texttt{trans(}P'$, A', M', $Q'\texttt{)}$, *side cond.* is a variant of the **Res** clause;

ii. $(\{Y', P', A', M', Q'\} \cap (\{P_r, U_1, U_2, U_3\} \cup$ *vars*$(\theta_c))) = \emptyset$;

iii. $\sigma = (Y' = \texttt{Y}, P' = \texttt{P}, U_1 = A', U_2 = M', U_3 = \texttt{nu}(Y', Q'))$.

Here "*side cond*" refers to the queries $\texttt{not\_in\_action(}Y'$, $A'\texttt{)}$ and $\texttt{not\_in\_constraint(}Y',\ M'\texttt{)}$. We now show that $Q'\theta_1 = Q'\theta_a$. Note that $Y'\theta_1 = \texttt{Y}$. Let $A'\theta_1 = \texttt{in(X, Z)}$. We give the derivation tree for the query $\texttt{not\_in\_action(}$ $Y'\texttt{,}A'\texttt{)}$. The proof is similar when $A'\theta_1$ is an output or internal action.

$$\dfrac{\dfrac{Y''\backslash == X'' : \theta_1\sigma_2 \ \to\ \theta_1\sigma_2,\ \ Y''\backslash == Z'' : \theta_1\sigma 2 \ \to\ \theta_1\sigma_2}{Y''\backslash{=}{=}X'',\ Y''\backslash{=}{=}Z''{:}\theta_1\sigma_2 \to \theta_1\sigma_2}}{\texttt{not\_in\_action}(Y',A'){:}\theta_1 \to \theta_1\sigma_2}$$

where

iv. $\texttt{not\_in\_action(}Y''\texttt{, in(}X''\texttt{, }Z''\texttt{))}$ is a variant of the $\texttt{not\_in\_action}$ clause;

v. $\{Y'', X'', Z''\} \cap (\{Y', A'\} \cup$ *vars*$(\theta_1)) = \emptyset$;

vi. $\sigma_2 = \{Y'' = \texttt{Y}, X'' = \texttt{X}, Z'' = \texttt{Z}\}$;

vii. $\exists\delta$ such that $\theta_1\sigma_2\delta$ is consistent, $Y''\theta_1\sigma_2\delta \backslash = X''\theta_1\sigma_2\delta$ and $Y''\theta_1\sigma_2\delta \backslash = Z''\theta_1\sigma_2\delta$.

Clearly, σ_2 does not affect names in $Q'\theta_1$ and hence $Q'\theta_1 = Q'\theta_1\sigma_2$. Similarly, we can show that the query $\texttt{not\_in\_constraint(}A'\texttt{,}M'\texttt{)}$ does not affect names in $Q'\theta_1\sigma_2$. Thus, $Q'\theta_1 = Q'\theta_a$. Similarly, $A'\theta_1 = A'\theta_a$ and $M'\theta_1 = M'\theta_a$.

Part 3: Let $\texttt{T1} = U_3\theta_a$. From iii, $\texttt{T1} = \texttt{nu}(Y', Q')\theta_a$. We need to show that for every $Y \in$ *bn*$(\texttt{nu}(Y', Q')\theta_a)$, Y is a variable and ($Y \in$ *bn*$(\texttt{nu(Y,P)})$ or $Y \notin$ *vars*(θ_c)). If $Y = Y'\theta_a$, then by Part 1, $Y'\theta_a = \texttt{Y}$ and hence $Y = \texttt{Y}$. Since $\texttt{nu(Y,P)}$ is a valid process expression, $\texttt{Y}$ is a variable. Further, since $\texttt{Y} \in$ *bn*$(\texttt{nu(Y, P)})$, $Y \in$ *bn*$(\texttt{nu(Y, P)})$. If $Y \in$

$bn(Q'\theta_a)$, then since $Q'\theta_a = Q'\theta_1$, by the induction hypothesis, for every $V \in bn(Q'\theta_a)$, V is a variable and ($V \in bn(P'\theta_c\sigma)$ or $V \notin vars(\theta_c\sigma)$). Thus Y is a variable. Since $bn(P'\theta_c\sigma) \subseteq bn(\texttt{nu(Y,P)})$, and $vars(\theta_c) \subseteq vars(\theta_c\sigma)$, $Y \in bn(\texttt{nu(Y, P)})$ or $Y \notin vars(\theta_c)$.

Part 4: We need to show that $bn(\texttt{nu}(Y', Q')\theta_a) \cap bn(A'\theta_a) = \emptyset$. It suffices to show that $Y'\theta_a \notin bn(A'\theta_a)$ and $bn(Q'\theta_a) \cap bn(A'\theta_a) = \emptyset$. By the induction hypothesis, $bn(Q'\theta_1) \cap bn(A'\theta_1) = \emptyset$. Since $A'\theta_1 = A'\theta_a$ and $Q'\theta_1 = Q'\theta_a$, $bn(Q'\theta_a) \cap bn(A'\theta_a) = \emptyset$. By vii, $Y'\theta_1 \notin vars(A'\theta_1)$. This means that $Y'\theta_a \notin vars(A'\theta_a)$. Thus, 4 holds.

Part 5: Let $\texttt{T1} = U_3\theta_a$. We need to show that T1 is a valid process expression. By the induction hypothesis, $Q'\theta_a$ is a valid process expression. Since $\texttt{T1} = \texttt{nu}(Y', Q')\theta_a$, it suffices to show that $Y'\theta_a \notin bn(Q'\theta_a)$, i.e., $\texttt{Y} \notin bn(Q'\theta_a)$. By Part 3, for every $V \in bn(Q'\theta_a)$, $V \in bn(P'\theta_c\sigma)$ (i.e., $V \in bn(\texttt{P})$) or $V \notin vars(\theta_c\sigma)$. If $V \in bn(\texttt{P})$, then since nu(Y,P) is a valid process expression, $\texttt{Y} \notin bn(\texttt{P})$ and hence $\texttt{Y} \neq V$. If $V \notin vars(\theta_c\sigma)$), then since $\texttt{Y} \in vars(\theta_c\sigma)$, $\texttt{Y} \neq V$.

Part 6: Suppose that $\texttt{trans}(P_r, U_1, U_2, U_3)\colon \theta_c \to \theta_a$ is derived from the above derivation tree in the logic program $D \cup \texttt{MMCtrans}$. By the induction hypothesis, given a one-to-one function ψ where $var(P_r\theta_c) \subseteq domain(\psi) \subseteq vars(\theta_c)$, there exists an extension ψ' of ψ such that $f_\psi(P'\theta_c\sigma) \xrightarrow{fc_{\psi'}(M'\theta_1), f_{\psi'}(A'\theta_1)} f_{\psi'}(Q'\theta_1)$ is an inferable transition in the symbolic semantics where $vars(Q'\theta_1) \subseteq domain(\psi') \subseteq vars(\theta_1)$. As discussed above, $Q'\theta_1 = Q'\theta_a$, $A'\theta_1 = A'\theta_a$, and $M'\theta_1 = M'\theta_a$. Thus $f_\psi(P'\theta_c\sigma) \xrightarrow{fc_{\psi'}(M'\theta_a), f_{\psi'}(A'\theta_a)} f_{\psi'}(Q'\theta_a)$ is an inferable transition where $vars(Q'\theta_a) \subseteq domain(\psi') \subseteq vars(\theta_a)$. Since ψ' is a one-to-one function, side conditions $f_{\psi'}(\texttt{Y}) \notin n(f_{\psi'}(A'\theta_a))$ (corresponding to not_in_action/2) and $f_{\psi'}(\texttt{Y}) \notin n(f_{\psi'}(M'\theta_a))$ (corresponding to not_in_constraint/2) hold. Since ψ' is an extension of ψ, $f_\psi(\texttt{Y}) = f_{\psi'}(\texttt{Y})$. By the **Res** rule, $(\nu f_\psi(\texttt{Y}))f_\psi(P'\theta_c\sigma) \xrightarrow{fc_{\psi'}(M'\theta_a), f_{\psi'}(A'\theta_a)} (\nu f_{\psi'}(\texttt{Y}))f_{\psi'}(Q'\theta_a)$ is an inferable transition. Since $P_r\theta_c = \texttt{nu}(\texttt{Y}, P'\theta_c\sigma)$, $U_1\theta_a = A'\theta_a$, $U_2\theta_a = M'\theta_a$, and $U_3\theta_a = \texttt{nu}(\texttt{Y}, Q'\theta_a)$, $f_\psi(P_r\theta_c) \xrightarrow{fc_{\psi'}(U_2\theta_a), f_{\psi'}(U_1\theta_a)} f_{\psi'}(U_3\theta_a)$ is also an inferable transition in the symbolic semantics.

Open: The proof is similar to that for the **Res** rule.

Close: The proof for the **Close** rule is more complicated than the others. There are four cases to consider, corresponding to the four defining clauses for predicate comp_bound/2. We treat only the first as the proofs of other cases are similar. Let $P_r\theta_c = \texttt{par(P1,P2)}$ and assume that a **Close** rule is applied in the last step of the derivation. In this case, the derivation tree is of the following form:

$$\frac{\dfrac{\vdots}{\begin{array}{r}\texttt{trans}(P_1', A', M', Q_1') : \theta_c\sigma \to \theta_c\sigma\sigma_1,\\ \texttt{trans}(P_2', B', N', Q_2') : \theta_c\sigma\sigma_1 \to \theta_1,\\ \text{“}\textit{side cond}\text{”} : \theta_1 \to \theta_a\end{array}}}{\dfrac{\texttt{trans}(P_1', A', M', Q_1'), \texttt{trans}(P_2', B', N', Q_2'),\ \text{“}\textit{side cond}\text{”}:\ \theta_c\sigma \to \theta_a}{\texttt{trans}(P_r, tau, U_1, U_2) :\ \theta_c \ \to\ \theta_a}}$$

where

i. $\texttt{trans(par(}P_1'\texttt{,}\ P_2'\texttt{), tau,}\ MNL'\texttt{, nu(}W'\texttt{, par(}Q_1'\texttt{,}\ Q_2'\texttt{)))}$:– $\texttt{trans(}P_1', A', M', Q_1'\texttt{), trans(}P_2', B', N', Q_2'\texttt{),}$ *side cond.* is a variant of the **Close** clause;

ii. $(\{P_1', P_2', A', B', M', N', W', MNL', Q_1', Q_2'\} \cap (\{P_r, U_1, U_2\} \cup vars(\theta_c))) = \emptyset$;

iii. $\sigma = (P_1' = \texttt{P1},\ P_2' = \texttt{P2},\ U_1 = MNL',\ U_2 = \texttt{nu}(W', \texttt{par}(Q_1', Q_2')))$.

Here, “*side cond*” refers to the query $\texttt{comp\_bound(}A'\texttt{,}\ B'\texttt{,}\ U_3\texttt{,}\ U_4\texttt{)}$.

We first show that $P_1'\theta_c\sigma$ and $P_2'\theta_c\sigma\sigma_1$ are valid process expressions that must be performed to inductively prove Parts 1–6 for this case. In particular, this will enable us to prove that $\texttt{par}(Q_1', Q_2')\theta_1$, $\texttt{par}(Q_1'\texttt{,}Q_2')\theta_a$, and $\texttt{nu}(W'\texttt{,par}(Q_1'\texttt{,}Q_2'))\theta_a$ are valid process expressions (Part 5).

Since process par(P1,P2) is a valid process expression, by Proposition 1, P1 and P2 are valid process expressions, i.e., $P_1'\theta_c\sigma$ and $P_2'\theta_c\sigma$ are valid process expressions. Further, since $P_1'\theta_c\sigma$ and $P_2'\theta_c\sigma$ have only free names in common, $bn(P_2'\theta_c\sigma) = bn(P_2'\theta_c\sigma\sigma_1)$. By Part 1, free names do not change in the evaluation, which gives us $P_2'\theta_c\sigma\sigma_1 = P_2'\theta_c\sigma$ and hence $P_2'\theta_c\sigma\sigma_1$ is a valid process expression. By the induction hypothesis, $Q_1'\theta_c\sigma\sigma_1$ and $Q_2'\theta_1$ are valid process expressions.

Next, we show that $Q_1'\theta_c\sigma\sigma_1 = Q_1'\theta_1$. This is achieved by first proving that $bn(Q_1'\theta_c\sigma\sigma_1) \cap bn(P_2'\theta_c\sigma\sigma_1) = \emptyset$, i.e., by proving that $bn(Q_1'\theta_c\sigma\sigma_1) \cap bn(P_2'\theta_c\sigma) = \emptyset$. For every $V_1 \in bn(Q_1'\theta_c\sigma\sigma_1)$, by Part 3, $V_1 \in bn(P_1'\theta_c\sigma)$ (i.e., $V_1 \in bn(\texttt{P1})$) or $V_1 \notin vars(\theta_c\sigma)$. For every $V_2 \in bn(P_2'\theta_c\sigma)$, $V_2 \in bn(\texttt{P2})$. If $V_1 \in bn(\texttt{P1})$, then since par(P1,P2) is a valid process expression, $V_1 \neq V_2$. If $V_1 \notin vars(\theta_c\sigma)$, then since $V_2 \in vars(\theta_c\sigma)$, $V_1 \neq V_2$. Thus, $bn(Q_1'\theta_c\sigma\sigma_1) \cap bn(P_2'\theta\sigma\sigma_1) = \emptyset$. By Part 1, we know that the free names of $Q_1'\theta_c\sigma\sigma_1$ are not affected by the evaluation of query $\texttt{trans(}P_2', B', N', Q_2'\texttt{)}$. Thus $Q_1'\theta_c\sigma\sigma_1 = Q_1'\theta_1$. Similarly, $A'\theta_c\sigma\sigma_1 = A'\theta_1$.

We now proceed with the proofs of Parts 3–6.

Part 3: Let $\texttt{T1} = U_2\theta_a$. We need to show that for every $V \in bn(\texttt{T1})$, V is a variable, and ($V \in bn(\texttt{par(P1,P2)})$ or $V \notin \theta_c$). Since $\texttt{T1} = \texttt{nu}(W'\texttt{,par}(Q_1', Q_2'))\theta_a$, $V = W'\theta_a$ (i.e., $V = \texttt{W}$), or $V \in bn(Q_1'\theta_a)$ or $V \in bn(Q_2'\theta_a)$. We first show that $bn(Q_1'\theta_a) = bn(Q_1'\theta_1)$ and $bn(Q_2'\theta_a) = bn(Q_2'\theta_1)$.

Assume that $A'\theta_1 = \texttt{outbound(X,W)}$ and $B'\theta_1 = \texttt{in(Y,Z)}$. The derivation tree for the query $\texttt{comp\_bound(}A', B', U_3, U_4)$ is the following:

$$\frac{X'' == Y'' : \theta_1\sigma_2 \ \to\ \theta_1\sigma_2}{\texttt{comp\_bound}(A', B', U_3, U_4) : \theta_1 \ \to \theta_1\sigma_2}$$

where

iv. $\texttt{comp\_bound(outbound(}X''\texttt{,}W''\texttt{), in(}Y''\texttt{,}W''\texttt{),}\ W''\texttt{, true)}$:– $X'' == Y''$. is a variant of the (first clause of the) **Close** rule;

v. $(\{X'', Y'', W''\} \cap (\{A', B', U_3, U_4\} \cup vars(\theta_c))) = \emptyset$;

vi. $\sigma_2 = (X'' = \texttt{X},\ W'' = \texttt{W},\ Y'' = \texttt{Y},\ W'' = \texttt{Z},\ U_3 = W'',\ U_4 = \texttt{true})$;

vii. $\theta_1\sigma_2 = \theta_a$.

By iv–vi, $Q'_1\theta_a = Q'_1\theta_1$ and $Q'_2\theta_a = Q'_2\theta_1\sigma_2$. Thus $bn(Q'_1\theta_a) = bn(Q'_1\theta_1)$ holds. Further, since $\mathtt{Z} \notin bn(Q'_2\theta_1)$ (by Part 4), $bn(Q'_2\theta_a) = bn(Q'_2\theta_1)$.

If $V = \mathtt{W}$, then since $\mathtt{W}$ is a bound name in action `outbound(X, W)`, by the induction hypothesis, $\mathtt{W}$ is a variable, and ($\mathtt{W} \in bn(\mathtt{P1})$, or $\mathtt{W} \notin vars(\theta_c\sigma)$). Since $bn(\mathtt{P1}) \subseteq bn(\mathtt{par(P1,P2)})$, and $vars(\theta_c) \subseteq vars(\theta_c\sigma)$, $\mathtt{W} \in bn(\mathtt{par(P1, P2)})$ or $\mathtt{W} \notin vars(\theta_c)$. If $V \in bn(Q'_1\theta_1)$, then by the induction hypothesis, V is a variable, and ($V \in bn(P'_1\theta_c\sigma)$ (i.e., $V \in bn(\mathtt{P1})$)) or $V \notin vars(\theta_c\sigma)$. Since $bn(\mathtt{P1}) \subseteq bn(\mathtt{par(P1,P2)}$ and $vars(\theta_c) \in vars(\theta_c\sigma)$, $V \in bn(\mathtt{par(P1, P2)}$ or $V \notin vars(\theta_c)$. Similarly, we can prove the case where $V \in bn(Q'_2\theta_1)$.

Part 4 holds because $bn(tau) = \emptyset$.

Part 5: Let $\mathtt{T1} = U_2\theta_a$. We need to prove that `T1` is a valid process expression. By iii, $\mathtt{T1} = \mathtt{nu}(W', \mathtt{par}(Q'_1, Q'_2))\theta_a$. First, we prove that $\mathtt{par}(Q'_1, Q'_2)\theta_1$ is a valid process expression. It suffices to show the following:

- $fn(\mathtt{par}(Q'_1, Q'_2)\theta_1) \cap bn(\mathtt{par}(Q'_1, Q'_2)\theta_1) = \emptyset$.
  We need to prove (a) $fn(Q'_1\theta_1) \cap bn(Q'_2\theta_1) = \emptyset$ and (b) $bn(Q'_1\theta_1) \cap fn(Q'_2\theta_1) = \emptyset$. Below we give the proof of (a). (b) can be similarly proved.
  Here we consider only the case where none of $fn(Q'_1\theta_1)$, $fn(Q'_2\theta_1)$, $bn(Q'_1\theta_1)$, and $bn(Q'_2\theta_1)$ is empty, since otherwise the above argument holds immediately. Let X be a free name of process $Q'_1\theta_1$. By Part 2, X is either a free name of process $P'_1\theta_c\sigma$, i.e., a free name of process `P1`, or a bound name of action $A'\theta_1$. If Y is a bound name of process $Q'_2\theta_1$, by Part 3, $Y \in bn(P'_2\theta_c\sigma\sigma_1)$ or $Y \notin vars(\theta_c\sigma\sigma_1)$. We now consider the case where X is a free name of process `P1`.
  – If $Y \in bn(P'_2\theta_c\sigma\sigma_1)$, then as described above, $Y \in bn(P'_2\theta_c\sigma)$, i.e., $Y \in bn(\mathtt{P2})$. Since `par(P1,P2)` is a valid process, $X \neq Y$.
  – If $Y \notin vars(\theta_c\sigma\sigma_1)$, then since $X \in vars(\theta_c\sigma\sigma_1)$ or X is not a variable, $X \neq Y$.
  Similarly, we can prove the case where X is a bound name of action $A'\theta_1$.
- $ubn(\mathtt{par}(Q'_1\theta_1, Q'_2\theta_1)) = bn(\mathtt{par}(Q'_1\theta_1, Q'_2\theta_1))$.
  This suffices to show $bn(Q'_1\theta_1) \cap bn(Q'_2\theta_1) = \emptyset$. The proof is similar to that of (a).

Next, we show that process $\mathtt{par}(Q'_1, Q'_2)\theta_a$ is a valid process expression. As with the above proof, we need to establish the following three equations: $bn(Q'_1\theta_a) \cap fn(Q'_2\theta_a) = \emptyset$, $fn(Q'_1\theta_a) \cap bn(Q'_2\theta_a) = \emptyset$, and $bn(Q'_1\theta_a) \cap bn(Q'_2\theta_a) = \emptyset$. Assume that $A'\theta_c\sigma\sigma_1 = \mathtt{outbound(X,W)}$ and $B'\theta_1 = \mathtt{in(Y,Z)}$. Then $bn(Q'_1\theta_a) = bn(Q'_1\theta_1)$ and $fn(Q'_2\theta_a) \subseteq (fn(Q'_2\theta_1) \cup \{\mathtt{W}\})$. Since $\mathtt{W} \notin bn(Q'_1\theta_1)$ (by Part 4) and $\mathtt{par}(Q'_1, Q'_2)\theta_1$ is a valid process expression, $\mathtt{par}(Q'_1, Q'_2)\theta_a$ is a valid process expression.

Finally, we show that $\mathtt{nu}(W', \mathtt{par}(Q'_1, Q'_2))\theta_a$ is a valid process expression. Since $W'\theta_a = \mathtt{W}$, it suffices to show that $\mathtt{W} \notin bn(\mathtt{par}(Q'_1, Q'_2)\theta_a)$, i.e, $\mathtt{W} \notin bn(Q'_1\theta_a)$ and $\mathtt{W} \notin bn(Q'_2\theta_a)$. Since $Q'_1\theta_a = Q'_1\theta_c\sigma\sigma_1$, by Part 4, $\mathtt{W} \notin bn(Q'_1\theta_a)$. We now show that $\mathtt{W} \notin bn(Q'_2\theta_a)$. By Part 3, $\mathtt{W} \in bn(P'_1\theta_c\sigma)$ (i.e., $\mathtt{W} \in bn(\mathtt{P1})$) or $\mathtt{W} \notin var(\theta_c\sigma)$. We consider the case where $\mathtt{W} \in bn(\mathtt{P1})$. Since $bn(Q'_1\theta_a) = bn(Q'_1\theta_1)$, by Part 3, for every $V \in bn(Q'_2\theta_a)$, $V \in bn(P'_2\theta_c\sigma\sigma_1)$ (i.e., $V \in bn(\mathtt{P2})$) or $V \notin vars(\theta_c\sigma\sigma_1)$. If $V \in bn(\mathtt{P2})$, then since `par(P1,P2)` is a valid process expression, $\mathtt{W} \neq V$. If $V \notin vars(\theta_c\sigma\sigma_1)$, then since $\mathtt{W} \in vars(\theta_c\sigma\sigma_1)$, $\mathtt{W} \neq V$.

Part 6: Suppose that $\mathtt{trans}(P_r\mathtt{,\ tau,\ }U_1\mathtt{,\ }U_2)\colon \theta_c \to \theta_a$ is an answer derived from the above derivation tree in the logic program $D\cup$`MMCtrans`. By the induction hypothesis, given a one-to-one function ψ where $vars(P_r\theta_c) \subseteq domain(\psi) \subseteq vars(\theta_c)$, there exists an extension ψ' of ψ such that $f_\psi(P'_1\theta_c\sigma) \xrightarrow{fc_{\psi'}(M'\theta_c\sigma\sigma_1), f_{\psi'}(A'\theta_c\sigma\sigma_1)} f_{\psi'}(Q'_1\theta_c\sigma\sigma_1)$ and $f_\psi(P'_2\theta_c\sigma\sigma_1) \xrightarrow{fc_{\psi'}(N'\theta_1), f_{\psi'}(B'\theta_1)} f_{\psi'}(Q'_2\theta_1)$ are inferable transitions in the symbolic semantics where $vars(Q'_1\theta_c\sigma\sigma_1) \subseteq domain(\psi') \subseteq vars(\theta_c\sigma\sigma_1)$ and $vars(Q'_2\theta_1) \subseteq domain(\psi') \subseteq vars(\theta_1)$. Assume that $A'\theta_c\sigma\sigma_1 = \mathtt{outbound(X,W)}$, $B'\theta_1 = \mathtt{in(Y,Z)}$ and $\mathtt{X} = \mathtt{Y}$. Then, since ψ' is a one-to-one function, $f_{\psi'}(\mathtt{X})$ and $f_{\psi'}(\mathtt{Y})$ are the same π-calculus name. Note that by applying alpha-conversion to the **Close** rule, the **Close** rule can be rewritten as follows:

$$\frac{P_1 \overset{M,\overline{x}\nu w}{\longrightarrow} Q_1,\ P_2 \overset{N,y(z)}{\longrightarrow} Q_2}{P_1 \mid P_2 \overset{MNL,\tau}{\longrightarrow} (\nu w)(Q_1 \mid Q_2\{w/z\})} \quad L = \begin{cases} \emptyset & \text{if } x = y; \\ x = y & \textit{otherwise}, \end{cases}$$

where $w \notin fn(Q_2)$.

Thus $f_\psi(P'_1\theta_c\sigma) \mid f_\psi(P'_2\theta_c\sigma\sigma_1) \xrightarrow{fc_{\psi'}(M'\theta_c\sigma\sigma_1 N'\theta_1),\tau} (\nu f_{\psi'}(\mathtt{W}))$ $(f_{\psi'}(Q'_1\theta_c\sigma\sigma_1) \mid f_{\psi'}(Q'_2\theta_1\{\mathtt{W/Z}\}))$ is an inferable transition in symbolic semantics. Because names in $M'\theta_c\sigma\sigma_1$ and $N'\theta_1$ are free names of $P'_1\theta_c\sigma$ and $P'_2\theta_c\sigma\sigma_1$ respectively, $M'\theta_c\sigma\sigma_1 = M'\theta_a$ and $N'\theta_1 = N'\theta_a$. Further, as discussed above, $P'_2\theta_c\sigma\sigma_1 = P'_2\theta_c\sigma$, $Q'_1\theta_c\sigma\sigma_1 = Q'_1\theta_a$ and $Q'_2\theta_1\{\mathtt{W/Z}\} = Q'_2\theta_a$. Thus $f_\psi(P'_1\theta_c\sigma) \mid f_\psi(P'_2\theta_c\sigma)$ $\xrightarrow{fc_{\psi'}(M'N')\theta_a,\tau} (\nu f_{\psi'}(\mathtt{W}))(f_{\psi'}(Q'_1\theta_a) \mid f_{\psi'}(Q'_2\theta_a))$ is an inferable transition in symbolic semantics where $vars(\mathtt{par}(Q'_1, Q'_2)\theta_a) \subseteq domain(\psi') \subseteq vars(\theta_a)$. Since $P_r\theta_c = \mathtt{par}(P'_1, P'_2)\theta_c\sigma$, $f_{\psi'}(\mathtt{tau}) = \tau$, $U_1\theta_a = (M'N')\theta_a$, and $U_2\theta_a = \mathtt{nu}(\mathtt{W}, \mathtt{par}(Q'_1, Q'_2))\theta_a$, $f_\psi(P_r\theta_c) \xrightarrow{fc_{\psi'}(U_1\theta_a), f_{\psi'}(\mathtt{tau})} f_{\psi'}(U_2\theta_a)$ is also an inferable transition in symbolic semantics. The proofs for the other clauses of `comp_bound/2` are similar.

Com: The proof is similar to that for the **Close** clause.

□

B Proof of Theorem 3

We first present a *constructive symbolic semantics* for the π-calculus where (i) alpha-conversion is limited to the application of the **Ide** inference rule of Fig. 1 and (ii) when applying the **Ide** rule, bound names are always renamed to *fresh* names not previously encountered in the derivation. We show that every derivation derivable in the original symbolic semantics has an equivalent derivation in the constructive semantics.

Next, we show that every transition derivable in the constructive semantics has a corresponding transition derivable in MMC such that the names of π-calculus process expressions have a one-to-one mapping to the names of the corresponding MMC process expressions.

We then prove the completeness of the constructive semantics of Fig. 7 with respect to the semantics of Fig. 1. That is, whenever there is a derivation in the original semantics, then there is an equivalent derivation in the constructive semantics. We first establish several fundamental properties of derivations in the constructive semantics used in the proof of completeness.

Proposition 5. *Let D_π be a set of π-calculus process definitions and P be a valid process expression. Let V_1 be a set of names where $(n(P) \cup n(D_\pi)) \subseteq V_1$. If $V_1, V_2 : P \overset{M,\alpha}{\longmapsto} P'$ is a derivation in the semantics in Fig. 7, then the following hold:*

1. *For every $y \in (fn(\alpha) \cup n(M))$, $y \in fn(P)$; for every $y \in fn(P')$, $y \in fn(P)$ or $y \in bn(\alpha)$.*
2. *For every $y \in (bn(\alpha) \cup bn(P'))$, $y \in bn(P)$ or $y \notin V_1$.*
3. *$bn(\alpha) \cap bn(P') = \emptyset$.*
4. *$V_1 \subseteq V_2$.*
5. *P' is a valid process expression.*

Proof. The proof is by induction on the number of steps used to derive a transition according to the symbolic semantics of Fig. 7. Part 4 is a straightforward consequence of the way the sets V_1 and V_2 are maintained and manipulated by the rules of Fig. 7. The proofs for Parts 1, 2, 3, and 5 can be carried out along the lines of the proof for Theorem 2 and are omitted. $\square$

Theorem 6 (Completeness of Constructive Semantics). *Let D_π be a set of π-calculus process definitions and S a process expression. Let T be a valid process expression and σ a renaming function such that $T\sigma \equiv S$. Also, let V_1 be a set of names such that $(n(T) \cup n(D_\pi)) \subseteq V_1$ and $domain(\sigma) \subseteq V_1$. If $S \overset{M,\alpha}{\longrightarrow} S'$ is a derivation in the semantics of Fig. 1, then there exists a derivation $V_1, V_2 : T \overset{M',\alpha'}{\longmapsto} T'$ in the semantics of Fig. 7 and a renaming function σ' such that*

– $\sigma'(v') = \begin{cases} v & \text{if } \alpha' = x'(v') \text{ and } \alpha = \sigma(x')(v), \\ & \text{or } \alpha' = x'\nu v' \text{ and } \alpha = \sigma(x')\nu v; \\ \sigma(v') & \text{otherwise}; \end{cases}$

– *$M'\sigma' = M$ and $T'\sigma' \equiv S'$;*

– *$n(T') \subseteq V_2$ and $domain(\sigma') \subseteq V_2$.*

Proof. By induction on the number of steps used to derive a transition according to the original symbolic semantics of Fig. 1. We give the proofs for **Pref**, **Ide**, **Par**, **Open**, **Com**, and **α-conversion** rules. The other rules can be similarly proved.

Prefix: Let $S = \alpha.P$. We first consider the case where α is an output or internal prefix. Since $T\sigma \equiv S$, T is of the form $\alpha'.P'$, where $\alpha'\sigma = \alpha$ and $P'\sigma \equiv P$. Assume that $\alpha.P \overset{true,\alpha}{\longrightarrow} Q$ is a derivation in the semantics of Fig. 1. Since the **Prefix** rule of Fig. 7 is an axiom, $V_1, V_1 : \alpha'.P' \overset{true,\alpha'}{\longmapsto} P'$ is a derivation in the semantics of Fig. 7. In this case, $\sigma = \sigma'$. Since $n(\alpha'.P') \subseteq V_1$, $n(P') \subseteq V_1$. Further, since $domain(\sigma) \subseteq V_1$, the theorem holds.

If α is an input action $x(y)$ and $\alpha' = x'(y')$ such that $x = \sigma(x')$, then there exists a renaming function σ' such that $\sigma'(v') = \begin{cases} y & \text{if } v' = y'; \\ \sigma(v') & \text{otherwise.} \end{cases}$
Clearly, $\alpha'\sigma' = \alpha$. Since $fn(P') \subseteq fn(\alpha'.P') \cup \{y'\}$, by the definition of σ', $P'\sigma' \equiv P$. Since $n(\alpha'.P') \subseteq V_1$, $n(P') \subseteq V_1$. Further, since $domain(\sigma') = domain(\sigma)$, $domain(\sigma') \subseteq V_1$.

Ide: Let $S = r(y_1, \ldots, y_n)$ and assume that $T = r(v_1, \ldots, v_n)$. Then, since $T\sigma \equiv S$, $\sigma(v_i) = y_i$ for $1 \le i \le n$. Assume that $r(y_1, \ldots, y_n) \overset{M,\alpha}{\longrightarrow} S'$ is a derivation in the semantics of Fig. 1 derived from $P\{y_1, \ldots, y_n/x_1, \ldots, x_n\} \overset{M,\alpha}{\longrightarrow} S'$, where $r(x_1, \ldots, x_n) \overset{\text{def}}{=} P$ is a definition. Let $\vartheta = \{z_i'/z_i \mid 1 \le i \le k\}$, where $\{z_1, \ldots, z_k\} = bn(P)$ and $z_i' \notin V_1$ and z_i' are pairwise distinct. Also, let $P' = P\{y_1, \ldots, y_n/x_1, \ldots, x_n\}\vartheta$ and $V_2 = V_1 \cup \{z_1', \ldots, z_k'\}$. It is easy to see that $n(P') \subseteq V_2$. Since all definitions are valid, $P'\vartheta$ is valid. Further, since y_i are the only free names in both $P\{y_1, \ldots, y_n/x_1, \ldots, x_n\}$ and in P', $P' \equiv P\{y_1, \ldots, y_n/x_1, \ldots, x_n\}$. Let σ_1 be an extension of σ such that $P'\sigma_1 = P\{y_1, \ldots, y_n/x_1, \ldots, x_n\}$ and $domain(\sigma_1) \subseteq V_2$. By the induction hypothesis, there exists a derivation $V_2, V_3 : P' \overset{M',\alpha'}{\longmapsto} T'$ in the semantics of Fig. 7 and a renaming function σ' such that

$$\sigma'(v') = \begin{cases} v & \text{if } \alpha' = x'(v') \text{ and } \alpha = \sigma_1(x')(v), \\ & \text{or } \alpha' = x'\nu v' \text{ and } \alpha = \sigma_1(x')\nu v; \\ \sigma_1(v') & \text{otherwise.} \end{cases}$$

$M'\sigma' = M$, $\alpha'\sigma' = \alpha$, $T'\sigma' \equiv S'$, $n(T') \subseteq V_3$, and $domain(\sigma') \subseteq V_3$. Thus there exists a derivation $r(y_1, \ldots, y_n) \overset{M',\alpha'}{\longmapsto} T'$ in the semantics of Fig. 7 and σ' described above such that $M'\sigma' = M$, $\alpha'\sigma' = \alpha$, $T'\sigma' \equiv S'$, $n(T') \subseteq V_3$, and $domain(\sigma') \subseteq V_3$.

Par: Let $S = P_1 \mid P_2$ and assume that the **Par** rule is applied in the last step of the derivation. Since $T\sigma \equiv S$, $T = Q_1 \mid Q_2$, where $Q_1\sigma \equiv P_1$ and $Q_2\sigma \equiv P_2$. Assume that $P_1 \mid P_2 \overset{M,\alpha}{\longrightarrow} P_1' \mid P_2$ is a derivation in the semantics of Fig. 1 derived from $P_1 \overset{M,\alpha}{\longrightarrow} P_1'$, where $bn(\alpha) \cap P_2 = \emptyset$. Since T is a valid process expression, Q_1 and Q_2 are valid. Further, since $n(T) \subseteq V_1$, $n(Q_1) \subseteq V_1$. By the induction hypothesis, there exists a derivation $V_1, V_2 : Q_1 \overset{M',\alpha'}{\longmapsto} Q_1'$ in the semantics of Fig. 7 and a renaming function σ' such that

$$\sigma'(v') = \begin{cases} v & \text{if } \alpha' = x'(v') \text{ and } \alpha = \sigma(x')(v), \\ & \text{or } \alpha' = x'\nu v' \text{ and } \alpha = \sigma(x')\nu v; \\ \sigma(v') & \text{otherwise.} \end{cases}$$

$M'\sigma' = M$, $\alpha'\sigma' = \alpha$, $Q'_1\sigma' \equiv P'_1$, $n(Q'_1) \subseteq V_2$ and $domain(\sigma') \subseteq V_2$. Since $n(T) \subseteq V_1$ and $n(Q'_1) \subseteq V_2$ and $V_1 \subseteq V_2$, $n(Q'_1 \mid Q_2) \subseteq V_2$. Further, since $Q'_1\sigma' \equiv P'_1$ and $Q_2\sigma' \equiv P_2$, $(Q'_1 \mid Q_2)\sigma' \equiv P'_1 \mid P_2$. We now show that $bn(\alpha') \cap fn(Q_2) = \emptyset$. Let $x \in bn(\alpha')$. By Proposition pro:pi-prop2, $x \in bn(Q_1)$ or $x \notin V_1$. If $x \in bn(Q_1)$, then since $Q_1 \mid Q_2$ is a valid process expression, $x \notin fn(Q_2)$. If $x \notin V_1$, then since $n(Q_2) \subseteq V_1$, $x \notin fn(Q_2)$. Thus $V_1, V_2 :$ $Q_1 \mid Q_2 \overset{M',\alpha'}{\longmapsto} Q'_1 \mid Q_2$ is a derivation in the semantics of Fig. 7 where $M'\sigma' = M$, $\alpha'\sigma' = \alpha$, $(Q'_1 \mid Q_2)\sigma' \equiv (P'_1 \mid P_2)$, $n(Q'_1 \mid Q_2) \subseteq V_2$ and $domain(\sigma') \subseteq V_2$.

Open: Let $S = (\nu y)P$ and assume that the **Open** rule is applied in the last step of the derivation. Since $T\sigma \equiv S$, T is of the form $(\nu y')Q$. Assume that $(\nu y)P \xrightarrow{M,\overline{x}y} P'$ is a derivation in the semantics of Fig. 1 derived from $P \xrightarrow{M,\overline{x}y} P'$ where $y \notin n(x, M)$. Since T is a valid process expression, Q is valid. Further, since $n(T) \subseteq V_1$, $n(Q) \subseteq V_1$. Let σ_1 be a renaming function such that

$$\sigma_1(v') = \begin{cases} y & \text{if } v' = y' \\ \sigma(v') & \text{otherwise.} \end{cases}$$

Then $Q\sigma_1 \equiv P$. By the induction hypothesis, there exists a derivation $V_1, V_2 :$ $Q \overset{M',\overline{x'}y'}{\longmapsto} Q'$ in the semantics of Fig. 7 and a renaming function $\sigma' = \sigma_1$ such that $M'\sigma' = M$, $\alpha'\sigma' = \alpha$, $Q'\sigma' \equiv P'$, $n(Q') \subseteq V_2$ and $domain(\sigma') \subseteq V_2$. We now show that $y' \notin n(x', M')$. By Proposition pro:pi-prop2, $x' \in fn((\nu y')Q)$ and $n(M') \subseteq fn((\nu y')Q)$. Since $(\nu y')Q$ is a valid process expression, $y' \notin n(x', M')$. Thus there exists a derivation $V_1, V_2 :$ $(\nu y')Q \overset{M',\overline{x'}\nu y'}{\longmapsto} Q'$ derived in the semantics in Fig. 7 and a renaming function σ' described above such that $M'\sigma' = M$, $\alpha'\sigma' = \alpha$, $Q'\sigma' \equiv P'$, $n(Q') \subseteq V_2$ and $domain(\sigma') \subseteq V_2$.

Com: Let $S = P_1 \mid P_2$, and assume that the **Com** rule is applied in the last step of derivation. Since $T\sigma \equiv S$, $T\sigma = Q_1 \mid Q_2$, where $Q_1\sigma \equiv P_1$ and $Q_2\sigma \equiv P_2$. Assume that $P_1 \mid P_2 \xrightarrow{MN,\tau} P'_1\{v/z\} \mid P'_2$ is a derivation in the semantics of Fig. 1 derived from $P_1 \xrightarrow{M,y(z)} P'_1$ and $P_2 \xrightarrow{N,\overline{x}v} P'_2$, where $x = y$. Since T is a valid process expression, Q_1 and Q_2 are valid. Since $n(T) \subseteq V_1$, $n(Q_2) \subseteq V_1$ and hence $n(Q_2) \subseteq V_2$. By the induction hypothesis, there exist derivations $V_1, V_2 :$ $Q_1 \overset{M',y'(z')}{\longmapsto} Q'_1$ and $V_2, V_3 :$ $Q_2 \overset{N',\overline{x'}v'}{\longmapsto} Q'_2$ in the semantics of Fig. 7 and σ' such that

$$\sigma'(w') = \begin{cases} z & \text{if } w' = z'; \\ \sigma(w') & \text{otherwise.} \end{cases}$$

$M'\sigma' = M$, $N'\sigma' = N$, $(y'(z'))\sigma' = y(z)$, $(\overline{x'}v')\sigma' = \overline{x}v$, $Q'_1\sigma' \equiv P'_1$, $Q'_2\sigma' \equiv P_2$, $(n(Q'_1) \cup n(Q'_2)) \subseteq V_3$ and $domain(\sigma') \subseteq V_3$. Since $v' \in fn(\overline{x'}v')$, by Proposition 5, $v' \in fn(Q'_2)$, and hence $v' \in V_3$. Thus, $n(Q'_1\{v'/z'\} \mid Q'_2) \subseteq V_3$. By Proposition 5, $z' \notin bn(Q'_1)$ and hence $z' \notin bn(Q'_1\{v'/z'\})$. Thus, $Q'_1\{v'/z'\}\sigma' = Q'_1\{v'/z'\}\sigma$. Further, since $\sigma(v') = v$, $Q'_1\{v'/z'\}\sigma \equiv P'_1\{v/z\}$. By Proposition pro:pi-prop2, we can infer that $z' \notin n(Q'_2)$. Thus, $Q'_2\sigma' = Q'_2\sigma$. Also, since names in M' and N' are free names of Q_1 and Q_2, respectively, and $Q_1 \mid Q_2$ is a valid process expression, $z' \notin n(M')$ and $z' \notin n(N')$. Thus, $M'N'\sigma = M'N'\sigma'$. Therefore, there exists a derivation $V_1, V_3 :$ $Q_1 \mid Q_2 \overset{M'N',\tau}{\longmapsto} Q'_1\{v'/z'\} \mid Q'_2$ in the semantics of Fig. 7 and $\sigma' = \sigma$ such that $M'N'\sigma' = MN$, $(Q'_1\{v'/z'\} \mid Q'_2)\sigma' \equiv (P'_1\{v/z\} \mid P'_2)$, $n(Q'_1\{v'/z'\} \mid Q'_2) \subseteq V_3$, and $domain(\sigma') \subseteq V_3$. The case where $x \neq y$ can be similarly proved.

Alpha: Assume that $S \xrightarrow{M,\alpha} S'$ is derived from $S_1 \xrightarrow{M,\alpha} S_2$ via alpha-conversion, where $S \equiv S_1$ and $S_2 \equiv S'$. Since $T\sigma \equiv S$, $T\sigma \equiv S_1$. By the induction hypothesis, there exists a derivation $V_1, V_2 :$ $T \overset{M',\alpha'}{\longmapsto} T'$ in the semantics of Fig. 7 and a renaming function σ' such that

$$\sigma'(v') = \begin{cases} v & \text{if } \alpha' = x'(v') \text{ and } \alpha = \sigma(x')(v), \\ & \text{or } \alpha' = x'\nu v' \text{ and } \alpha = \sigma(x')\nu v; \\ \sigma(v') & \text{otherwise.} \end{cases}$$

$M'\sigma' = M$, $\alpha'\sigma' = \alpha$, $T'\sigma' \equiv S_2$, $n(T') \subseteq V_2$, and $domain(\sigma') \subseteq V_2$. Since $S_2 \equiv S'$, $T'\sigma' \equiv S'$. Thus the theorem holds. □

We can now establish the completeness of the encoding. In particular, we show that each transition derivable using the constructive semantics of the π-calculus (Fig. 7) has an equivalent transition derivable via resolution using the logic program of Fig. 4. We call this program `MMCtrans`.

Theorem 7 (Completeness of MMC with respect to Constructive Semantics). *Let D_π be a set of π-calculus process definitions and S be a valid process expression. Also, let V_1 be a set of names, θ_c a call substitution, φ a one-to-one function mapping π-calculus names to Prolog variables such that $n(S) \subseteq domain(\varphi) \subseteq V_1$ and $range(\varphi) \subseteq vars(\theta_c)$. Finally, let P_r be a Prolog variable such that $P_r\theta_c = g_\varphi(S)$, and let U_1, U_2, and U_3 be three distinct Prolog variables not in $vars(\theta_c)$. If the transition $V_1, V_2 :$ $S \overset{M,\alpha}{\longmapsto} S'$ is derivable in the π-calculus semantics (Fig. 7), then there exists a derivation for* `trans`$(P_r, U_1, U_2, U_3) : \theta_c \to \theta_a$ *from the logic program $\zeta(D_\pi) \cup$*`MMCtrans`*and an extension φ' of φ such that $n(S') \subseteq domain(\varphi') \subseteq V_2$, $range(\varphi') \subseteq vars(\theta_a)$, $U_1\theta_a = g_{\varphi'}(\alpha)$, $U_2\theta_a = gc_{\varphi'}(M)$, and $U_3\theta_a = g_{\varphi'}(S')$.*

Proof. By induction on the number of steps used to derive a transition according to the symbolic semantics of

Fig. 7. We give the proof for the **Ide** rule. The proofs for the other rules are easier and can be similarly proved.

Ide: Suppose that $V_1, V_3 : r(y_1, \ldots, y_n) \overset{M,\alpha}{\longmapsto} S'$ is a derivation in the semantics of Fig. 7 derived from $V_2, V_3 : P_1 \overset{M,\alpha}{\longmapsto} S'$. Let φ be a one-to-one function such that $\{y_1, \ldots, y_n\} \subseteq \mathit{domain}(\varphi) \subseteq V_1$. Since $\mathit{domain}(\varphi) \subseteq V_1$ and $\mathit{range}(\varphi) \subseteq \mathit{vars}(\theta_c)$, $\mathit{bn}(P_1) \cap \mathit{domain}(\varphi) = \emptyset$ and $\mathit{bn}(P'\theta_c\sigma\sigma_1) \cap \mathit{range}(\varphi) = \emptyset$. (For $P'\theta_c\sigma\sigma_1$, refer to the derivation tree of the **Ide** rule in Theorem 2). Thus, we can construct a one-to-one function φ_1 that is an extension of φ such that φ_1 maps bound names of P_1 to the corresponding bound names in $P'\theta_c\sigma\sigma_1$. In this case, $g_{\varphi_1}(P_1) = P'\theta_c\sigma\sigma_1$. Since P_1 is a valid process expression and φ_1 is a one-to-one function, $P'\theta_c\sigma\sigma_1$ is a valid process expression. Further, since $\mathit{bn}(P_1) \subseteq V_2$, and $\mathit{bn}(P'\theta_c\sigma\sigma_1) \in \mathit{vars}(\theta_c\sigma\sigma_1)$, $\mathit{domain}(\varphi_1) \subseteq V_2$ and $\mathit{range}(\varphi_1) \subseteq \mathit{vars}(\theta_c\sigma\sigma_1)$ hold. By the induction hypothesis, there exists an extension φ' of φ_1 such that $\mathtt{trans}(P', U_1, U_2, U_3) : \theta_c\sigma\sigma_1 \to \theta_a$ is an answer derivable from the logic program $\zeta(D_\pi) \cup \mathtt{MMCtrans}$, where $n(S') \subseteq \mathit{domain}(\varphi') \subseteq V_2$, $\mathit{range}(\varphi') \subseteq \mathit{vars}(\theta_a)$, $\{U_1, U_2, U_3\} \cap \mathit{vars}(\theta_c\sigma\sigma_1) = \emptyset$, $U_1\theta_a = g_{\varphi'}(\alpha')$, $U_2\theta_a = gc_{\varphi'}(M')$, and $U_3\theta_a = g_{\varphi'}(S')$ and $g_{\varphi'}(S')$ is a valid process expression. Thus there exists an extension φ' of φ such that $\mathtt{trans}(P_r, U_1, U_2, U_3) : \theta_c \to \theta_a$ is an answer derivable from the logic program $\zeta(D_\pi) \cup \mathtt{MMCtrans}$, where $P_r\theta_c = g_\varphi(r(y_1, \ldots, y_n))$, $n(S') \subseteq \mathit{domain}(\varphi') \subseteq V_2$, $\mathit{range}(\varphi) \subseteq \mathit{vars}(\theta_a)$, $\{U_1, U_2, U_3\} \cap \mathit{vars}(\theta_c) = \emptyset$, $U_1\theta_a = g_{\varphi'}(\alpha')$, $U_2\theta_a = gc_{\varphi'}(M')$, and $U_3\theta_a = g_{\varphi'}(S')$. □

The proof of Theorem 3 directly follows from Theorems 6 and 7.

C Proof of Theorem 4

Theorem 4. *Let D be a set of process and formula definitions, S the logic program consisting of the clauses in Figs. 4 and 9, P a valid process expression, and F a formula. Also, let Pr and Fr be two distinct Prolog variables. Then there exists a δ mapping free names of P to free names of F such that $\mathtt{models}(Pr, Fr) : \theta_c \to \theta_a$ is an answer derivable from the logic program $D \cup S$ if and only if $P \vdash_\delta F$ is a derivation in the tableau of Fig. 8, where $Pr\theta_c = P\delta$ and $Fr\theta_c = F\delta$.*

Proof. For the "if" part of the theorem, the proof is by induction on the number of steps used to derive a goal in the tableau of Fig. 8. For the "only if" part, the proof is by induction on the number of steps needed to derive an answer using the logic program given in Fig. 9. Here we give the proof for rules **True**, **Match**, **Diam**, and **Lfp** for the "if" part. The "only if" part can be proved similarly.

True: Let $F = \mathtt{tt}$ and assume that the **True** rule is applied in the last step of the derivation. Since $P \vdash_\delta \mathtt{tt}$ is an axiom in the tableau of Fig. 8, $P \vdash_\delta \mathtt{tt}$ is derivable in the tableau. Let $Pr\theta_c = P\delta$ and $Fr\theta_c = \mathtt{tt}$. The derivation tree of the **True** clause is as follows:

$$\frac{\mathtt{true}: \ \theta_c\sigma \to \theta_c\sigma}{\mathtt{models}(Pr, Fr): \ \theta_c \to \theta_c\sigma}$$

where

i. $\mathtt{models}(P',\mathtt{tt}).$ is a variant of the **True** clause;
ii. $\{P'\} \cap (\{Pr, Fr\} \cup \mathit{vars}(\theta_c)) = \emptyset$;
iii. $\sigma = \mathit{mgu}(Pr\theta_c, P')$.

Since $\mathtt{true} : \theta_c\sigma \to \theta_c\sigma$ is an axiom in the derivation tree, $\mathtt{models}(Pr, Fr) \colon \theta_c \to \theta_c\sigma$ is derivable from logic program $D \cup S$.

Match: Let $F = \mathtt{pred}((X = Y), F_1)$ and assume that the **Match** rule is applied in the last step of the derivation, i.e., $P \vdash_\delta \mathtt{pred}((X = Y), F_1)$ is derived from $P \vdash_\delta F_1$ under the condition $X\delta = Y\delta$. Let $Pr\theta_c = P\delta$ and $Fr\theta_c = F\delta$. The derivation tree of the **Match** clause is as follows:

$$\frac{\dfrac{X'{=}{=}Y'{:}\theta_c\sigma \to \theta_c\sigma, \quad \dfrac{\vdots}{\mathtt{models}(P',F_1'){:}\theta_c\sigma \to \theta_a}}{X'{=}{=}Y',\ \mathtt{models}(P',F_1'){:}\theta_c\sigma \to \theta_a}}{\mathtt{models}(Pr,Fr){:}\theta_c \to \theta_a}$$

where

i. $\mathtt{models}(P',\ \mathtt{pred}((X' = Y'),F_1')\ \ \text{:-}\ \ X' == Y',$ $\mathtt{models}(P',\ F_1').$ is a variant of the **Match** clause;
ii. $\{X', Y', P', F_1'\} \cap (\{Pr, Fr\} \cup \mathit{vars}(\theta_c)) = \emptyset$;
iii. $\sigma = \mathit{mgu}((Pr\theta_c, Fr\theta_c), (P', \mathtt{pred}((X' = Y'), F_1')))$.

Since $Pr\theta_c = P\delta$ and $Fr\theta_c = F\delta$, by iii, $P'\theta_c\sigma = P\delta$ and $F_1'\theta_c\sigma = F_1\delta$. By the induction hypothesis, $\mathtt{models}(P',$ $F_1') \colon \theta_c\sigma \to \theta_a$ is derivable from logic program $D \cup S$. The identity check $(X' == Y')\theta_c\sigma$ in the derivation tree corresponds to the condition $X\delta = Y\delta$ in Fig. 8. Thus $\mathtt{models}(Pr, Fr) \colon \theta_c \to \theta_a$ is derivable from $D \cup S$.

Diam: Let $F = \mathtt{diam}(A, F_1)$ and assume that the **Diam** rule is applied in the last step of the derivation, i.e., $P \vdash_\delta \mathtt{diam}(A, F_1)$ is derived from $P_1 \vdash_{\delta'} F_1$ where P can perform a transition A_1 to P_1 and $\delta' = \delta \cup \mathit{mgu}(A, A_1)$. Let $Pr\theta_c = P\delta$ and $Fr\theta_c = F\delta$. The derivation tree of the **Diam** clause is as follows:

$$\frac{\dfrac{\dfrac{\vdots}{\begin{array}{r}\mathtt{trans}(P', A_1', C', P_1') : \theta_c\sigma \to \theta_1,\\ A_1' = A' : \theta_1 \to \theta_1\sigma_1,\\ \mathtt{models}(P_1', F_1') : \ \theta_1\sigma_1 \ \to \ \theta_a\end{array}}}{\mathtt{trans}(P', A_1', C', P_1'),\ A_1'{=}A',\ \mathtt{models}(P_1',F_1'){:}\theta_c\sigma \to \theta_a}}{\mathtt{models}(Pr, Fr) : \theta_c \ \to \theta_a}$$

where

i. $\mathtt{models}(P',\ \mathtt{diam}(A',F_1'))\ \ \text{:-}\ \ \mathtt{trans}(P',\ A_1',\ C_1',$ $P_1'),\ A' = A_1',\ \mathtt{models}(P_1',\ F_1').$ is a variant of the **Diam** clause;
ii. $\{P', A', F_1', A_1', C_1', P_1'\} \cap (\{Pr, Fr\} \cup \mathit{vars}(\theta_c)) = \emptyset$;
iii. $\sigma = \mathit{mgu}((Pr\theta_c, Fr\theta_c), (P', \mathtt{diam}(A', F_1')))$ and $\sigma_1 = \mathit{mgu}(A', A_1')$.

Since $Pr\theta_c = P\delta$ and $Fr\theta_c = F\delta$, by iii, $P'\theta_c\sigma = P\delta$ and $F_1'\theta_c\sigma = F_1\delta$. Since $\sigma_1 = \mathit{mgu}(A', A_1')$, $P_1'\theta_1\sigma_1 = P_1(\delta \cup \mathit{mgu}(A, A_1))$ and $F_1'\theta_1\sigma_1 = F_1(\delta \cup \mathit{mgu}(A, A_1))$. By the induction hypothesis, $\texttt{models(}P_1', F_1'\texttt{)}\colon \theta_1\sigma_1 \to \theta_a$ is derivable from $D \cup S$. Thus $\texttt{models(}Pr, Fr\texttt{)}\colon \theta_c \to \theta_a$ is derivable from $D \cup S$.

Lfp: Let $F = \texttt{form}(Z(\overrightarrow{V_1}))$ and assume that the **Lfp** rule is applied in the last step of the derivation. Suppose that $\texttt{fdef(}Z(\overrightarrow{V})\texttt{, lfp(}F_1\texttt{))}$ is the process definition of process $Z(\overrightarrow{V_1})$ and $P \vdash_\delta \texttt{form}(Z(\overrightarrow{V_1}))$ is derived from $P \vdash_\delta F_1[\overrightarrow{V_1}/\overrightarrow{V}]$. Let $Pr\theta_c = P\delta$ and $Fr\theta_c = F\delta$. The derivation tree of the **Lfp** clause is as follows:

$$\dfrac{\dfrac{\texttt{fdef}(D',\texttt{lfp}(F_1'))\texttt{,}\colon\theta_c\sigma \to \theta_1,\quad \dfrac{\vdots}{\texttt{models}(P',F_1')\colon\theta_1 \to \theta_a}}{\texttt{fdef}(D',\texttt{lfp}(F_1'))\texttt{, models}(P',F_1')\colon\theta_c\sigma \to \theta_a}}{\texttt{models}(Pr,Fr)\colon\theta_c \to \theta_a}$$

where

i. $\texttt{models(}P'\texttt{,form(}D'\texttt{))} \coloneq \texttt{def(}D'\texttt{,lfp(}F_1'\texttt{))}$, $\texttt{models(}P'\texttt{, }F_1'\texttt{)}$. is a variant of the **Lfp** clause;
ii. $\{P', D', F_1'\} \cap (\{Pr, Fr\} \cup \mathit{vars}(\theta_c)) = \emptyset$;
iii. $\sigma = \mathit{mgu}((Pr\theta_c, Fr\theta_c), (P', \texttt{form}(D')))$.

Since $Pr\theta_c = P\delta$ and $Fr\theta_c = F\delta$, by iii, $P'\theta_c\sigma = P\delta$ and $D'\theta_c\sigma = Z(\overrightarrow{V_1})\delta$. Thus $P'\theta_1 = P\delta$ and $F_1'\theta_1 = F_1\delta$. By the induction hypothesis, $\texttt{models(}P'\texttt{, }F_1'\texttt{)}\colon \theta_1 \to \theta_a$ is a derivation of the logic program $D \cup S$ and $\texttt{models(}Pr, Fr\texttt{)}\colon \theta_c \to \theta_a$ does not previously occur in the derivation. Thus $\texttt{models(}Pr, Fr\texttt{)}\colon \theta_c \to \theta_a$ is derivable from $D \cup S$. $\square$