Executable Semantics and Type Checking for Session-Based Concurrency in Maude

Carlos Alberto Ramírez Restrepo[1] and Jorge A. Pérez[2]

[1] Pontificia Universidad Javeriana Cali, Colombia
[2] University of Groningen, The Netherlands

Abstract. Session types are a well-established approach to communication correctness in message-passing programs. We present an executable specification of the operational semantics of a session-typed π-calculus, implemented in Maude. We also develop an executable specification of its associated algorithmic type checking, and describe how both specifications can be integrated. We further explore how our executable specification enables us to detect well-typed but deadlocked processes by leveraging reachability and model checking tools in Maude. Our developments define a promising new approach to the (semi)automated analysis of communication correctness in message-passing concurrency.

1 Introduction

This paper presents an executable rewriting semantics for a π-calculus equipped with *session types*. Widely known as the paradigmatic calculus of interaction, the π-calculus offers a rigorous platform for reasoning about message-passing concurrency. Session types are arguably the most prominent representative of *behavioral type systems*, which can statically ensure that processes respect their ascribed *interaction protocols* and never exhibit errors and mismatches.

The integration of (variants of) the π-calculus with different formulations of session types has received much attention from foundational and applied perspectives. As a result, our understanding about (abstract) communicating processes and their typing disciplines steadily reaches maturity. Despite this progress, rigorous connections with more concrete representation models fall short. In particular, the study of session-typed π-calculi within frameworks like Maude [1] seems to remain unexplored. This gap is an opportunity to investigate the formal systems underlying session-typed π-calculi (reduction semantics and type systems) from a fresh yet rigorous perspective, taking advantage of the concrete representation given by executable semantics in Maude.

Looking at session-typed π-calculi from the perspective of Maude is insightful, for several reasons. First, Maude enables the systematic validation of such formal systems and their results, improving over pen-and-paper developments. Second, as there is not a canonical session-typed π-calculus, but actually many different formulations (with varying features and properties), an implementation in Maude could provide a concrete platform for uniformly representing them all.

Third, resorting to Maude as a host representation framework for session-typed π-calculi could help in addressing known limitations of static type checking for deadlock detection, leveraging tools already available in Maude.

This paper reports our work on pursuing these three directions. We adopt the session-typed π-calculus developed by Vasconcelos in [10] as the basis for our implementation in Maude. For this typed language, dubbed $\mathsf{s}\pi$, we first implement its (untyped) reduction semantics as a rewriting semantics, essentially extending prior work on representing the π-calculus in Maude. Then, we implement its associated algorithmic type system, also given in [10]. Well-typedness in [10] ensures *fidelity* (i.e., well-typed processes respect at runtime their ascribed protocols) but does not rule out deadlocks and other kinds of insidious circular dependencies. To address this, we leverage reachability and model checking in Maude. Our Maude developments are publicly available online.[3]

To our knowledge, we are the first to represent session-typed π-calculi using Maude. Prior works have used rewriting logic to investigate the operational semantics for variants of the π-calculus. In [13] and [12], the reduction semantics of a synchronous π-calculus is defined as a rewrite theory, which is implemented in ELAN. The work [9] considers an untyped, asynchronous π-calculus, whose labeled transition semantics is implemented as a rewrite theory, which is used to formalize an associated may-testing preorder. The work [4] concerns a typed process calculus but in a different context, in which types are used to enforce privacy properties. Indeed, such work gives a Maude implementation of the labeled transition semantics of a privacy-oriented variant of the π-calculus and a Maude implementation of its associated type system, which is implemented as a membership equational theory.

The rest of this paper is organized as follows. Next, Section 2 summarizes the syntax and semantics of $\mathsf{s}\pi$. Section 3 describes the definition of our rewriting semantics for $\mathsf{s}\pi$ in Maude, whereas Section 4 presents the rewriting implementation of the algorithmic type checking. Section 5 presents our developments on deadlock detection. Section 6 closes with some concluding remarks. Additional material has been collected in the appendices.

2 The Typed Process Model

The typed process calculus $\mathsf{s}\pi$, formalized by Vasconcelos [10], is a variant of the synchronous π-calculus (cf. [6]) with constructs for session-based concurrency. Here we summarize its syntax and semantics.

The calculus $\mathsf{s}\pi$ relies on a base set of *variables*, ranged over by $x, y, \ldots$. Variables denote *channels* (or *names*). Processes interact to exchange values, which can be variables or booleans. Variables can be seen as consisting of (dual) *endpoints* on which interaction takes place. Rather than non-deterministic choices among prefixed processes, there are two complementary operators: one for offering a finite set of alternatives (called *branching*) and one for choosing one of

[3] See `https://gitlab.com/calrare1/session-types`

$P \mid Q \;\equiv\; Q \mid P$	$P \mid \mathbf{0} \;\equiv\; P$	
$P \mid (Q \mid R) \;\equiv\; (P \mid Q) \mid R$	$(\boldsymbol{\nu} xy)\ \mathbf{0} \;\equiv\; \mathbf{0}$	
$(\boldsymbol{\nu} xy)(\boldsymbol{\nu} wz)P \;\equiv\; (\boldsymbol{\nu} wz)(\boldsymbol{\nu} xy)P$	$(\boldsymbol{\nu} xy)P \mid Q \;\equiv\; (\boldsymbol{\nu} xy)(P \mid Q)$	If $x, y \notin \mathtt{fn}(Q)$
$\mathsf{if\ true\ then}\ P_1\ \mathsf{else}\ P_2 \;\equiv\; P_1$	$\mathsf{if\ false\ then}\ P_1\ \mathsf{else}\ P_2 \;\equiv\; P_2$	

Fig. 1. Structural congruence Rules for $\mathsf{s}\pi$

such alternatives (*selection*). More formally, the syntax of *values*, *qualifiers*, and *processes* is presented below:

$$
\begin{array}{l}
v ::= x \quad | \quad \mathsf{true} \quad | \quad \mathsf{false} \qquad\qquad q ::= \mathsf{un} \quad | \quad \mathsf{lin} \\
P ::= \mathbf{0} \quad | \quad \overline{x}v.P \quad | \quad q\ x(y).P \quad | \quad P_1 \mid P_2 \quad | \quad (\boldsymbol{\nu} xy)P \quad | \\
\qquad\ \mathsf{if}\ v\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 \quad | \quad x \triangleleft l.P \quad | \quad x \triangleright \{l_i : P_i\}_{i \in I}
\end{array}
$$

The inactive process is denoted as $\mathbf{0}$. The output process $\overline{x}v.P$ sends the value v along x and continues as P. Process $q\ x(y).P$ denotes an input action on x, which prefixes P. The qualifier q is used for inputs, which can be linear (to be executed exactly once) or shared. Process $\mathsf{un}\ x(y).P$ denotes a persistent input action, which corresponds to (input-guarded) replication in the π-calculus. The parallel composition $P_1 \mid P_2$ denotes the concurrent execution of P_1 and P_2. Process $(\boldsymbol{\nu} xy)P$ declares the scope of *co-variables* x and y to be P. These co-variables are intended to be the output and input ends of a communication channel. Given a boolean v, process $\mathsf{if}\ v\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2$ continues as P_1 if v is true; otherwise it continues as P_2. Finally, selection process $x \triangleleft l.P$ chooses an option l offered by a process prefixed at the co-variable and branching process $x \triangleright \{l_i : P_i\}_{i \in I}$ offers multiple alternatives, which are labeled $l_1, l_2, \ldots$; the selection process continues with P and the branching process with a process P_j.

As usual, $q\ x(y).P$ binds y in P and $(\boldsymbol{\nu} xy)P$ binds x, y in P. The set of free and bound names of a process P, denoted $\mathtt{fn}(P)$ and $\mathtt{bn}(P)$, is as expected.

The operational semantics for $\mathsf{s}\pi$ is given as a reduction semantics, which, as customary, relies on a structural congruence relation, the smallest congruence relation on processes that satisfy the axioms in Fig. 1. Structural congruence includes the usual axioms for inaction and parallel composition as well as adapted axioms for scope restriction, scope extrusion, and conditionals. Armed with structural congruence, the rules of the reduction semantics are presented in Fig. 2. Rules [R-LinCom] and [R-UnCom] induce different patterns for process communication, depending on the qualifier of their corresponding input action. Indeed, processes $\overline{x}v.P$ and $q\ y(z).Q$ can synchronize if x and y are co-variables. This is only possible if both processes are underneath a scope restriction $(\boldsymbol{\nu} xy)$. When this occurs, processes $\overline{x}v.P$ and $q\ y(z).Q$ continue respectively as P and $Q[v/z]$, i.e., the process obtained from Q by substituting the free occurrences of z with v. When $q = \mathsf{un}$ then process $q\ y(z).Q$ remains (Rule [R-UnCom]); otherwise, process $q\ y(z).Q$ disappears (Rule [R-LinCom]). Rule [R-Case] stands

$$\frac{}{(\boldsymbol{\nu} xy)(\overline{x}v.P \mid \mathsf{lin}\ y(z).Q \ \mid\ R) \longrightarrow (\boldsymbol{\nu} xy)(P \ \mid\ Q[v/z] \mid R)} \quad \text{[R-LinCom]}$$

$$\frac{}{(\boldsymbol{\nu} xy)(\overline{x}v.P \mid \mathsf{un} y(z).Q \mid R) \longrightarrow (\boldsymbol{\nu} xy)(P \mid Q[v/z] \mid \mathsf{un}\ y(z).Q \ \mid R)} \quad \text{[R-UnCom]}$$

$$\frac{j \in I}{(\boldsymbol{\nu} xy)(x \triangleleft l_j.P \mid y \triangleright \{l_i : Q_i\}_{i\in I} \mid R) \longrightarrow (\boldsymbol{\nu} xy)(P \mid Q_j \mid R)} \quad \text{[R-Case]}$$

$$\frac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q} \qquad \frac{P \longrightarrow P'}{(\boldsymbol{\nu} xy)P \ \longrightarrow (\boldsymbol{\nu} xy)P'} \quad \text{[R-Par] [R-Res]}$$

$$\frac{P \equiv P' \qquad P' \longrightarrow Q' \qquad Q \equiv Q'}{P \longrightarrow Q} \quad \text{[R-Struct]}$$

Fig. 2. Reduction semantics for $\mathsf{s}\pi$

for the case synchronization: processes $x \triangleleft l_j.P$ and $y \triangleright \{l_i : Q_i\}_{i\in I}$ can synchronize if they are underneath a scope restriction $(\boldsymbol{\nu} xy)$. Process $x \triangleleft l_j.P$ reduces to process P and process $y \triangleright \{l_i : Q_i\}_{i\in I}$ reduces to process Q_j. Rules for parallel composition, scope restriction and structurally congruent processes are the usual from π-calculus (Rules [R-Par], [R-Res], [R-Struct]).

As an example, consider the processes:

$$P_1 = \mathsf{un}\ y_1(t).\overline{t}\mathsf{false}.\mathbf{0} \qquad P_2 = \mathsf{lin}\ y_1(w).\overline{w}\mathsf{true}.\mathbf{0} \qquad P_3 = \overline{x_1}x_2.y_2(z).\overline{a}z.\mathbf{0}$$
$$P = (\boldsymbol{\nu} x_1y_1)(\boldsymbol{\nu} x_2y_2)(P_1 \mid P_2 \mid P_3)$$

Starting from P, there are two possible sequences of reductions depending on the processes involved in the initial synchronization in the co-variables x_1, y_1. If the synchronization involves P_1 and P_3 then we have:

$$P \longrightarrow \ \ldots \ \longrightarrow \ (\boldsymbol{\nu} x_1y_1)(\boldsymbol{\nu} x_2y_2)(P_1 \mid P_2 \mid \overline{a}\mathsf{false}.\mathbf{0})$$

On the other hand, if P_2 and P_3 synchronize then we have:

$$P \longrightarrow \ \ldots \ \longrightarrow \ (\boldsymbol{\nu} x_1y_1)(\boldsymbol{\nu} x_2y_2)(P_1 \mid \overline{a}\mathsf{true}.\mathbf{0})$$

The *standard form* of a process, defined in [10], will be crucial for the executable specification of the reduction semantics. Intuitively, a process is in standard form whenever restrictions are expanded as much as possible. More precisely, we say P is in standard form if it matches the pattern expression $(\boldsymbol{\nu} x_1y_1)(\boldsymbol{\nu} x_2y_2)\ldots(\boldsymbol{\nu} x_ny_n)(P_1 \mid P_2 \mid \ \ldots \ \mid P_k)$, where each P_i is a process of the form $\overline{x}v.Q$, $qx(y).Q$, $x \triangleleft l.Q$ or $\triangleright\{l_i : Q_i\}_{i\in I}$. Every process is structurally congruent to a process in standard form.

3 Rewriting Semantics for $\mathsf{s}\boldsymbol{\pi}$

Syntax Our rewriting semantics for $\mathsf{s}\pi$ adapts the one in [9], which is defined for an untyped π-calculus without sessions. There is a direct correspondence between the syntactic categories (values, variables, qualifiers, and terms) and Maude sorts (`Value`, `Chan`, `Qualifier`, and `Trm`, respectively). We also have some auxiliary sorts such as `Guard`, `Choice`, and `Choiceset`.

```
sorts Value Chan Qualifier Trm Guard Choice Choiceset .
subsort Choice < Choiceset .
subsort Chan < Value .

op _{_} : Qid Nat -> Chan [prec 1] .
ops lin un : -> Qualifier [ctor] .
ops True False : -> Value [ctor] .
op  __(_) : Qualifier Chan Qid -> Guard [ctor prec 5] .
op _<_> : Chan Value -> Guard [ctor prec 6] .
op nil : -> Trm [ctor] .
op new[__]_ : Qid Qid Trm -> Trm [ctor prec 10] .
op _|_  :  Trm Trm -> Trm [ctor assoc comm prec 12 id: nil] .
op if_then_else_fi : Value Trm Trm -> Trm [ctor prec 8] .
op _ << _._ : Chan Qid Trm -> Trm [ctor prec 15] .
op _ >> {_} : Chan Choiceset -> Trm [ctor prec 17] .
op  _._ : Guard Trm -> Trm [ctor prec 7] .
op _:_ : Qid Trm -> Choice .
op empty : -> Choiceset [ctor] .
op __ : Choiceset Choiceset -> Choiceset [ctor assoc comm id: empty] .
```

Following the syntax in Section 2, values can be variables or booleans. We represent booleans as the constructors `True` and `False` whereas we distinguish variables (sort `Chan`) as values through the subsort relation. The only constructor for variables `_{_}` takes a `Qid` and a natural number. Each production rule for processes is represented using a constructor, as expected. Notice that the constructor for input guards `__(_)` is preceded by a qualifier. Process $\mathbf{0}$ is denoted as `nil` and a single guarded term is represented by the constructor `_._`. The constructor for scope restriction `new[__]_` uses two instances of `Qid`, since it declares a pair of co-variables. The constructor for conditionals is parametric on an instance of `Value`. We add constructors for selection and branching process terms; their definition is as expected. In particular, the constructor for branching processes relies on instances of `Choiceset`, which consists of sets of pairs of `Qid` and process terms. We use instances of `Qid` to represent labels.

Substitutions As we have seen, the semantics of $\mathsf{s}\pi$ relies on substitutions of variables with values. To deal with substitutions in Maude, we follow Thati et al.'s approach [9] and use Stehr's CINNI calculus [8], an explicit substitution calculus, which provides a mechanism to implement α-conversion at the language level. The idea behind CINNI is to syntactically associate each use of a variable x to an index, which acts as a counter of the number of binders for x that are found before it is used. In CINNI, there are three types of substitution operations:

Type	**Meaning**
Simple substitution	$[\mathtt{a} := \mathtt{x}]\ \mathtt{a}\{0\} \mapsto \mathtt{x}$ $\quad [\mathtt{a} := \mathtt{x}]\ \mathtt{a}\{\mathtt{n}+1\} \mapsto \mathtt{a}\{\mathtt{n}\}$ $[\mathtt{a} := \mathtt{x}]\ \mathtt{b}\{\mathtt{m}\} \mapsto \mathtt{b}\{\mathtt{m}\}$
Shift substitution	$\uparrow_{\mathtt{a}} \mathtt{a}\{\mathtt{n}\} \mapsto \mathtt{a}\{\mathtt{n}+1\}$ $\quad \uparrow_{\mathtt{a}} \mathtt{b}\{\mathtt{m}\} \mapsto \mathtt{b}\{\mathtt{m}\}$
Lift substitution	$\Uparrow_{\mathtt{a}} (\mathtt{S})\ \mathtt{a}\{0\} \mapsto \mathtt{a}\{0\}$ $\quad \Uparrow_{\mathtt{a}} (\mathtt{S})\ \mathtt{a}\{\mathtt{n}+1\} \mapsto \uparrow_{\mathtt{a}} (\mathtt{S}\ \mathtt{a}\{\mathtt{n}\})$ $\Uparrow_{\mathtt{a}} (\mathtt{S})\ \mathtt{b}\{\mathtt{m}\} \mapsto \uparrow_{\mathtt{a}} (\mathtt{S}\ \mathtt{b}\{\mathtt{m}\})$

A simple substitution of a variable `a` for a variable `x` takes place if the index of `x` is 0; the index is decreased by 1 otherwise. A shift substitution over `a` increases by 1 the index and a substitution `S` can be lifted to skip one index. Any substitution over a variable `a` has no effect on other variables.

We now present the definition of explicit subtitutions for $\mathtt{s}\pi$ using an approach similar to [8]. We firts present the definition of the variable substitutions. We use the sort `Subst` and the substitution application is performed by the operator `__`, which takes a substitution and a variable. We define the three substitutions above as presented there, by means of some equations.

```
sort Subst .
op [_:=_] : Qid Value -> Subst .
op [shiftup_] : Qid -> Subst .
op [lift__] : Qid Subst -> Subst .
op __ : Subst Chan -> Chan .

eq  [ a := v ] a{0} = v .
eq  [ a := v ] a{s(n)} = a{n} .
ceq [ a := v ] b{n} = b{n}  if a =/= b .
eq  [ shiftup a ] a{n} = a{s(n)} .
ceq [ shiftup a ] b{n} = b{n}  if a =/= b .
eq  [ lift a S ] a{0} = a{0} .
eq  [ lift a S ] a{s(n)} = [ shiftup a ] S a{n} .
ceq [ lift a S ] b{n} = [ shiftup a ] S b{n} if a =/= b .
```

Equipped with these elements, we adapt to the $\mathtt{s}\pi$ syntax the equations associated to the explicit substitutions for the process terms as follows:

```
op  __ :  Subst Trm -> Trm [prec 3] .
op  subst-aux : Subst Choiceset -> Choiceset .
eq  S nil = nil .
eq  S (new [x y] P) = new [x y] ([lift x S] [lift y S] P) .
eq  S (q a(y) . P ) = q (S a)(y) . ([lift y S] P) .
eq  S (a < b > . P) = (S a) < (S b) > . (S P) .
ceq S (a < v > . P) = (S a) < v > . (S P) if v == True or v == False .
ceq S (if v then P else Q fi) = if v then (S P) else (S Q) fi
                                if v == True or v == False .
eq  S (a >> {CH}) = (S a) >> { subst-aux(S, CH) } .
eq  S (a << x . P) = (S a) << x . (S P) .
eq  S (P | Q) = (S P) | (S Q) .
eq  subst-aux(S, empty) = empty .
eq  subst-aux(S, (x : P) CH) = (x : (S P)) subst-aux(S, CH) .
```

In each equation, we deal with a specific production rule for process terms. In each process, the substitution `S` is applied in each variable and each subprocess as expected. Particularly, a lift substitution is performed over `x`, `y` and `S` to skip the index 0 and perform the substitution in the remaining indices for the scope restriction operator. In this way, the substitution `S` has the expected effect.

Structural Congruence To represent the rules in Fig. 1, we exploit the Maude equational attributes `assoc`, `comm`, and `id` to declare the associative, commutative, and identity axioms for parallel composition, with process `nil` acting as its identity. This suffices to cover the rules on the two first lines of Fig. 1. The remaining rules are explicitly declared as equations below:

```
eq new[x y] nil = nil .
ceq P | new[x y] Q = new [x y] (Q | [shiftup x] [shiftup y] P)
     if P =/= nil /\ Q =/= nil /\ CS := freenames(P) /\
        x{0} in CS and y{0} in CS .
eq if True then P else Q fi = P .
eq if False then P else Q fi = Q .
ceq P | new[x y] Q = new [x y] (Q | [shiftup x] P)
     if P =/= nil /\ Q =/= nil /\ CS := freenames(P) /\
        x{0} in CS and not y{0} in CS .
ceq P | new[x y] Q = new [x y] (Q | [shiftup y] P)
     if P =/= nil /\ Q =/= nil /\ CS := freenames(P) /\
        not x{0} in CS and y{0} in CS .
ceq P | new[x y] Q = new [x y] (Q | P)
     if P =/= nil /\ Q =/= nil /\ CS := freenames(P) /\
        not x{0} in CS /\ not y{0} in CS .
```

In particular, scope extrusion is represented through four equations corresponding to the four cases in the presence of `x`, `y` in the free names of process `P`. Function `freenames` stands for the Maude implementation for function `fn` over processes.

Operational Semantics Combined, the Maude rewriting rules, the equational attributes, and the explicit equations associated to variables of sort `Trm` can appropriately express the reduction semantics of $\mathsf{s}\pi$ and manipulate terms in a compositional fashion. A process is reduced to a simpler equivalent form by virtue of the equational theory; a process is rewritten as long as it satisfies the structure required for a rule wherever the process is located. As a consequence, subprocesses are also rewritten and we do not need to explicitly represent the contextual rules ([R-Par] and [R-Res]).

A process is converted into standard form using the explicit congruence rules. This way, the scope of every unguarded occurrence of the `new` operator is extended to the top level.

Process interaction in $\mathsf{s}\pi$ can only occur through co-variables and therefore processes that are involved must be underneath a scope restriction over such co-variables. Nonetheless, since in the standard form the order of the unguarded ocurrences of the `new` operator is irrelevant, it would be necessary to explicitly look for the processes that are enabled to interact, which would affect the efficiency of the rewriting specification. To counter this, we include an auxiliary operator, dubbed `new*`, which declares a list of pairs of new co-variables, rather than just a single pair. This is equivalent to using nested `new` operators, i.e., the term `new* [x1 y1 x2 y2 ... xn yn] P` is equivalent to the term

`new [x1 y1] new [x2 y2] ... new [xn yn] P`.

We declare the constructor for the sort `QidSet` with the equational attribute `comm` to impose that the order among the pairs of new co-variables is not distinguished. In this way, whatever they are the process to interact, these will be underneath a scope restriction `new*` and the interaction will be enabled.

```
sorts QidPair QidSet . subsort QidPair < QidSet .
op __ : Qid Qid -> QidPair [ctor] .
op mt : -> QidSet [ctor] .
op __ : QidSet QidSet -> QidSet [ctor comm assoc id: mt] .
op new* [_] _ : QidSet Trm -> Trm [ctor] .
```

Given a process P, let us write $[\![P]\!]$ to denote its representation in Maude. A reduction rule $P \longrightarrow Q$ can be associated to a rewriting rule $l : [\![P]\!]$ `=>` $[\![Q]\!]$. The reduction rules can be stated as follows:

```
crl [FLAT] : P => P' if P' := flatten(P) /\ P =/= P' .
rl [LINCOM] : new* [(x y) nl] x{N} < v > . P | lin y{N}(z) . Q | R =>
               new* [(x y) nl] P | [z := v] Q | R .
rl [UNCOM] : new* [(x y) nl] x{N} < v > . P | un y{N}(z) . Q | R =>
              new* [(x y) nl] P | [z := v] Q | un y{N}(z) . Q | R .
rl [CASE] : new* [(x y) nl] (x{N} << w . P) |
            (y{N} >> { (w : Q) CH }) | R => new* [(x y) nl] P | Q | R .
```

Rule `FLAT` normalizes the whole process. In this sense, additional to the implicit rewriting performed by the equations associated to the congruence rules, the nested `new` declarations are stated as a flat declaration `new*`. We use an auxiliary operation `flatten`, which is defined as follows:

```
op flatten : Trm -> Trm .
eq flatten(new [x y] P) = flatten(new* [x y] P) .
eq flatten(new* [nl] new [x y] P) = flatten(new* [nl x y] P) .
eq flatten(new* [nl] new* [nl'] P) = flatten(new* [nl nl'] P) .
eq flatten(P) = P [owise] .
```

Rules `LINCOM`, `UNCOM` and `CASE` correspond to the specification of the reduction rules related to synchronization in the calculus semantics (see Fig. 2). In these rules, `nl` stands for the additional co-variables being declared. As expected, Rules `LINCOM`, and `UNCOM` perform a substitution of the variable `z` for the value `v`.

We include also some equations which capture natural equivalences for processes involving the auxiliary operator `new*`.

```
eq new* [nl] nil = nil .
eq new* [x y nl] y{N} < v > . P | q x{N}(z) . Q | R =
   new* [y x nl] y{N} < v > . P | q x{N}(z) . Q | R .
eq new* [x y nl] (y{N} << w . P) | (x{N} >> { CH }) | R =
   new* [y x nl] (y{N} << w . P) | (x{N} >> { CH }) | R .
```

Given a pair `x y` of co-variables, we assume that the first action of `x` is an output or a selection and the first action `y` is an input or a branching. The last two equations swap `x` and `y` when this is not the case, to enable the execution of the rewriting rules.

Our rewriting specification enables us to directly execute a possible sequence of reductions over a process using the Maude command '`rew`'. In this way, we can obtain a stable (final) reachable process, which cannot reduce further. Moreover, we can use the reachability command '`search`' to: (i) perform all possible sequence of reductions of a process and obtain every possible stable process and (ii) check whether a process that fits some pattern is reachable or if a specific process is reachable. In Section 4, we leverage commands '`search`' and '`modelCheck`' to detect deadlocked $\mathsf{s}\pi$ processes.

Specification Correctness The transition system associated to our rewrite theory in Maude can be shown to coincide with the reduction semantics in Section 2. This operational correspondence result is detailed in Appendix A.

4 Algorithmic Type Checking for sπ

4.1 Type Syntax

We present a Maude implementation of the algorithmic type checking given in [10]. The type system considers *typing contexts*, denoted Γ, which associate each variable to a specific type, denoted T. Typing contexts and types are defined inductively as follows:

$$\Gamma ::= \emptyset \mid \Gamma, x : T \qquad q ::= \mathsf{lin} \mid \mathsf{un}$$
$$p ::= ?T.T \mid !T.T \mid \&\{l_i : T_i\}_{i\in I} \mid \oplus\{l_i : T_i\}_{i\in I}$$
$$T ::= \mathsf{bool} \mid \mathsf{end} \mid q\ p \mid a \mid \mu a.T$$

where q stands for qualifiers and p stands for pretypes. Moreover, x denotes a variable, each l_i denotes a label and a denotes a general variable. For simplicity, we assume a single basic type for values (bool). Each variable is associated to a (session) type, which represents its intended protocol. In the above grammar, these types correspond to qualified pretypes. The pretype $?T_1.T_2$ (resp. $!T_1.T_2$) is assigned to a variable that first receives (resp. sends) a value of type T_1 and then proceeds to type T_2. The pretype $\&\{l_i : T_i\}_{i\in I}$ (resp. $\oplus\{l_i : T_i\}_{i\in I}$) is assigned to a variable that can offer (resp. select) l_i options and continues with type T_i depending on the label selected. The type end (empty sequence) denotes the type of a variable where no interaction can occur. Recursive types can express infinite sequences of actions; in the type $\mu a.T$, a corresponds to a type variable that must occur guarded in T.

We encode session types in Maude by associating the non-terminals context, qualifiers, pretypes, and types to sorts `Context`, `Qualifier`, `Pretype`, and `Type`.

```
sorts Pretype Type Context ChoiceT ChoiceTset .
subsort ChoiceT < ChoiceTset .
op ?_._ : Type Type -> Pretype .  op !_._ : Type Type -> Pretype .
op +{_} : ChoiceTset -> Pretype . op &{_} : ChoiceTset -> Pretype .
ops bool end : -> Type .         op __ : Qualifier Pretype -> Type .
op u [_] _ : Qid Type -> Type .   op var : Qid -> Type .
ops nil invalid-context : -> Context .
op _:_ : Value Type -> Context .
op _,_ : Context Context -> Context [ctor assoc comm id: nil] .
op _:_ : Qid Type -> ChoiceT . op empty : -> ChoiceTset .
op __ : ChoiceTset ChoiceTset -> ChoiceTset [assoc comm id: empty] .
```

Each production rule is given as a specific constructor. In particular, constructors `+{_}` and `&{_}` represent the pretypes $\oplus\{l_i : T_i\}_{i\in I}$ and $\&\{l_i : T_i\}_{i\in I}$, respectively. The pairs of labels l_i and subtypes T_i are defined as instances of the sort `ChoiceTset`. The recursive type $\mu a.T$ is given as the constructor `u [_] _` and the type variables are given as the constructor `var`. Typing contexts are defined as expected. An empty context is denoted as `nil` whereas a single context is associated to the constructor `_:_`. General contexts are provided by the constructor `_,_`, which is annotated with the equational attributes `assoc`, `comm` and `id` since the order is irrelevant in typing contexts and the construction is associative. Finally, we added a constant `invalid-context` to be used in the type checking to denote a typing error.

4.2 Algorithmic Type Checking

We follow the algorithmic type checking proposed in [10]. This type system enables to type check the $\mathtt{s}\pi$ processes from Section 2, with a minor caveat: algorithmic type checking uses processes in which the restriction operator has a corresponding type annotation, i.e., it uses $(\boldsymbol{\nu}xy : T)P$ instead of $(\boldsymbol{\nu}xy)P$. Consequently, we add a constructor for the sort `Trm` in the Maude specification:

```
op  new[__:_]_ : Qid Qid Type Trm -> Trm [ctor prec 28] .
```

Following [10], we implement the type checking algorithm by relying on some auxiliary functions for type duality (i.e., compatibility), type equality, and context update and difference, among others. They are implemented by means of functions and equations in Maude. Appendix B gives the details of the Maude implementation for type duality (function `dual`), context update (function `+`), and the context difference (function `\`).

Algorithmic type checking is expressed by using sequents of the form $\Gamma_1 \vdash v : T; \Gamma_2$ for values and $\Gamma_1 \vdash P : \Gamma_2; L$ for processes. These two sequents have an input-output reading: sequent $\Gamma_1 \vdash v : T; \Gamma_2$ denotes an algorithm that takes Γ_1

$$\Gamma \vdash \mathsf{true} : \mathsf{bool}; \Gamma \quad [\text{A-True}] \qquad \Gamma_1, x : \mathsf{lin}\ p, \Gamma_2 \vdash x : \mathsf{lin}\ p; (\Gamma_1, \Gamma_2) \quad [\text{A-LinVar}]$$

$$\Gamma \vdash \mathsf{false} : \mathsf{bool}; \Gamma \quad [\text{A-False}] \qquad \frac{\mathsf{un}(T)}{\Gamma_1, x : T, \Gamma_2 \vdash x : T; (\Gamma_1, x : T, \Gamma_2)} \quad [\text{A-UnVar}]$$

Fig. 3. Typing rules for values, $\Gamma \vdash v : T; \Gamma$

and v as input and returns T and Γ_2 as output; similarly, sequent $\Gamma_1 \vdash P : \Gamma_2; L$ denotes an algorithm that takes Γ_1 and P as input and produces Γ_2 and L as output. While Γ_2 is a residual context, the set L collects linear variables occurring in subject position. Intuitively, L tracks the linear variables that are used in P to prevent that they are used again in another process. Both algorithms are given by means of typing rules, which we specify in Maude as an equational theory.

Fig. 3 shows the typing rules for values, which correspond to the rules in [10]. The rules for boolean values [A-True] and [A-False] produce as results the type bool and the input context Γ remains unaltered. There are two rules for a variable x: if x has a linear type lin p then the entry x : lin p is removed from the returned context (Rule [A-LinVar]); otherwise, if x is unrestricted then the entry $x : T$ is kept in the returned context (Rule [A-UnVar]). The algorithm for type checking of values is then implemented as a function `type-value`, which is defined as follows:

```
op type-value : Context Value -> TupleTypeContext .
eq type-value(C, True) = [C bool] .                         ---[A-TRUE]
eq type-value(C, False) = [C bool] .                        ---[A-FALSE]
ceq type-value(((a : T), C), a) = [((a : T), C) unfold(T)] ---[A-UNVAR]
    if unrestricted(T) .
eq type-value(((a : lin p), C), a) = [C (lin p)] .         ---[A-LINVAR]
eq type-value(((a : u [x] T), C), a) =
   type-value(((a : unfold(u [x] T)), C), a) .              ---[A-LINVAR]
eq type-value(C, v) = ill-typed [owise] .
```

Function `type-value` produces an instance of the sort `TupleTypeContext`. This sort groups a context and a type or a set of variables and it has only one constructor `[_ _]`. The equations related to the typing of boolean values arise as expected, according to the corresponding typing rule. In those cases, a tuple that contains the unmodified context and the type `bool` is produced. For unrestricted variables, given that some types are infinite then, before the update, the unrestricted types are *unfolded* (cf. the `unfold` operation). Unfolding is the mechanism defined in [10] to deal with infinite types: If a type T is a recursive type $\mu a.U$ then the substitution $U[\mu a.U/a]$ is performed. Otherwise, the type T remains unaltered. For linear variables, we also unfold the type when necessary and the linear type is returned and removed from the context.

Fig. 4 shows some of the typing rules for sπ processes; they largely correspond to the rules in [10].

$$\Gamma \vdash \mathbf{0} : \Gamma; \emptyset \qquad \frac{\Gamma_1 \vdash P : \Gamma_2; L_1 \qquad \Gamma_2 \div L_1 \vdash Q : \Gamma_3; L_2}{\Gamma_1 \vdash P \mid Q : \Gamma_3; L_2} \quad \text{[A-INACT] [A-PAR]}$$

$$\frac{\Gamma_1, x : T, y : \overline{T} \vdash P : \Gamma_2; L}{\Gamma_1 \vdash (\boldsymbol{\nu} xy : T)\ P : \Gamma_2 \div \{x, y\}; L \backslash \{x, y\}} \quad \text{[A-RES]}$$

$$\frac{\Gamma_1 \vdash v : q\ \mathsf{bool}; \Gamma_2 \qquad \Gamma_2 \vdash P : \Gamma_3; L \qquad \Gamma_2 \vdash Q : \Gamma_3; L}{\Gamma_1 \vdash \mathsf{if}\ v\ \mathsf{then}\ P\ \mathsf{else}\ Q : \Gamma_3; L} \quad \text{[A-IF]}$$

$$\frac{\Gamma_1 \vdash x : q!T.U; \Gamma_2 \qquad \Gamma 2 \vdash v : T; \Gamma_3 \qquad \Gamma_3 + x : U \vdash P : \Gamma_4; L}{\Gamma_1 \vdash \overline{x}v.P : \Gamma_4; L \cup (\text{if } q = \mathsf{lin} \text{ then } \{x\} \text{ else } \emptyset)} \quad \text{[A-OUT]}$$

$$\frac{\Gamma_1 \vdash x : q_2?T.U; \Gamma_2 \quad (\Gamma_2, y : T) + x : U \vdash P : \Gamma_3; L \quad q_1 = \mathsf{un} \Rightarrow L \backslash \{y\} = \emptyset}{\Gamma_1 \vdash q_1 x(y).P : \Gamma_3 \div \{y\}; L \backslash \{y\} \cup (\text{if } q_2 = \mathsf{lin} \text{ then } \{x\} \text{ else } \emptyset)} \quad \text{[A-IN]}$$

$$\frac{\Gamma_1 \vdash x : q\&\{l_i : T_i\}_{i \in I}; \Gamma_2 \quad \Gamma_2 + x : T_i \vdash P_i : \Gamma_3; L_i \quad \forall_{i \in I, j \in I}\ L_i \backslash \{x\} = L_j \backslash \{x\}}{\Gamma_1 \vdash x \triangleright \{l_i : P_i\}_{i \in I} : \Gamma_3; L \cup (\text{if } q = \mathsf{lin} \text{ then } \{x\} \text{ else } \emptyset)} \quad \text{[A-BRANCH]}$$

$$\frac{\Gamma_1 \vdash x : q \oplus \{l_i : T_i\}_{i \in I}; \Gamma_2 \qquad \Gamma_2 + x : T_j \vdash P : \Gamma_3; L \qquad j \in I}{\Gamma_1 \vdash x \triangleleft l_j.P : \Gamma_3; L \cup (\text{if } q = \mathsf{lin} \text{ then } \{x\} \text{ else } \emptyset)} \quad \text{[A-SEL]}$$

Fig. 4. Typing Rules for Processes, $\Gamma \vdash P : \Gamma; L$

Rule [A-INACT] proceeds as expected. Process $\mathbf{0}$ is well-typed and the typing context Γ remains unaltered and the set of linear variables is empty. Rule [A-PAR] handles parallel composition: to check a process $P \mid Q$ over a context Γ_1, the type of P is checked and the resulting context Γ_2 is used to type-check process Q, making sure that the linear variables used for P are first removed by using the context difference function ($\Gamma_2 \div L_1$). This ensures that free linear variables are used only once. The output of the algorithm for Q (context Γ_3 and set L_2) then corresponds to the ouput of the entire process $P \mid Q$. Rule [A-RES] type-checks a process $(\boldsymbol{\nu} xy : T)P$ in a context Γ_1: it first checks the type of sub-process P in the context Γ_1 extended with the association of variables x, y to the type T and its dual type, denoted $\overline{T}$. It is expected that if type T ($\overline{T}$) is linear then it should not be in the resulting context Γ_2; otherwise, if type T ($\overline{T}$) is unrestricted then it will appear in Γ_2. We require that variables x, y are deleted from the residual context ($\Gamma_2 \div \{x, y\}$) and from the set L of linear variables.

Rule [A-IF] verifies that type of value v is bool in the context Γ_1, and requires that the typecheck of P and Q in the context Γ_2 generate the same residual context Γ_3 and the same set L, since both processes should use the same linear variables. Rule [A-OUT] handles output processes: it uses the incoming context Γ_1 to check the type of x, which should be of the form $q!T.U$. Then, it checks that the type of v in the residual context Γ_2 is T. The type of the continuation P is checked in a new context Γ_3 extended with the association of x and the continuation type U. The rule enforces that types $q!T.U$ and U must be equivalent when x is unrestricted (i.e., $q = \mathsf{un}$). The rule returns a context Γ_4 and a set of variables L joined with x, if linear. Rule [A-In] presents some minor modifications with respect to the one in [10]. We require that in the case of replication there are no (free) subjects on linear variables in process P except possibly the input variable y. Other than this, this rule is similar to Rule [A-Out].

Rule [A-Sel] looks the type of x in the incoming context Γ_1. This type must be of the form $q \oplus \{l_i : T_i\}_{i \in I}$. Subsequently, the continuation P is type-checked under the resulting context Γ_2 updated with a new assumption for x, which is associated to a type T_j. In this way, when $q = \mathsf{un}$ we must have $\oplus\{l_i : T_i\}_{i \in I} = T_j$. This rule produces as result the context Γ_3 and the set of linear variables L is augmented with x if linear. Context Γ_3 and set L also corresponds to the output of the type checking of process P. Finally, we have Rule [A-Branch], which has some minor modifications with respect to the rule in [10]. More precisely, this rule has been changed to require that the sets of (free) subjects on linear variables L_i only differ in the input variable y. The additional details of this rule is quite similar to Rule [A-Sel].

As an example of type checking, if $T = \mathsf{lin}\ !\mathsf{bool}.\mathsf{lin}\ ?\mathsf{bool}.\mathsf{end}$ then we can establish the following sequent:

$$a : \mathsf{bool} \vdash (\boldsymbol{\nu} x_1 y_1 : T)(\mathsf{lin}\ y_1(v).\overline{y_1}v.\mathbf{0} \mid \overline{x_1}a.\mathsf{lin}\ x_1(z).\mathbf{0}) : (a : \mathsf{bool}); \{x_1, y_1\}$$

The algorithm for type-checking processes is implemented as a function `type-term` that receives an instance of the sort `Context` and an instance of the sort `Trm`. Moreover, it produces an instance of the sort `TupleTypeContext` that groups the resulting typing context and the set L of linear variables that were collected during type-checking. Each rule is implemented by an equation:

```
op type-term : Context Trm -> TupleTypeContext .
eq type-term(C, nil) = [C mt] .                              --- [A-INACT]
ceq type-term(C, P | Q) = [C2 L2]                              --- [A-PAR]
    if [C1 L1] := type-term(C, P) /\
       [C2 L2] := type-term(C1 / L1, Q) .
ceq type-term(C, new [x y : T] P) =                            --- [A-RES]
      [(C1 / (x{0} y{0})) remove(remove(L1, x{0}), y{0})]
  if [C1 L1] := type-term((C, (x{0} : T), (y{0} : dual(T))), P) .
ceq type-term(C, if v then P else Q fi) = [C2 L1]               --- [A-IF]
  if [C1 bool] := type-value(C, v) /\
     [C2 L1] := type-term(C1, P) /\ [C2 L1] := type-term(C1, Q) .
ceq type-term(C, a < v > . P) =                                --- [A-OUT]
         [C3 (if q == lin then (L1 a) else L1 fi)]
    if [C1 (q ! T . U)] := type-value(C, a) /\
       [C2 T’] := type-value(C1, v) /\ /\ equal(T, T’)
       [C3 L1] := type-term((C2 + a : U), P) .
ceq type-term(C, un a(y) . P) = [(C2 / y{0}) mt]                 --- [A-IN]
    if [C1 (un ? T . U)] := type-value(C, a) /\
       [C2 L] := type-term((C1, (y{0} : T)) + a : U, P) /\
       remove(L, y{0}) == mt .
ceq type-term(C, lin a(y) . P) =                                 --- [A-IN]
```

```
       [(C2 / y{0}) (remove(L, y{0})
        (if q == lin then a else mt fi))]
    if [C1 (q ? T . U)] := type-value(C, a) /\
       [C2 L] := type-term((C1, (y{0} : T)) + a : U, P) .
ceq type-term(C, a>>{CH}) = check-branch(C1, a, CH, CHT,q) ---[A-BRANCH]
    if [C1 (q & { CHT })] := type-value(C, a) .
ceq type-term(C, a << x . P) =                                ---[A-SEL]
        [C2 (if q == lin then (L1 a) else L1 fi)]
    if [C1 (q + { (x : T) CHT })] := type-value(C, a) /\
       [C2 L1] := type-term((C1 + a : T), P) .
eq type-term(C, P) = ill-typed [owise] .
```

When type checking is successful, function `type-term` produces an outgoing type context and a set of variables. Those elements are grouped using the constructor `[_,_]`, which is associated to the sort `TupleTypeContext`. We use a Maude comment to annotate each equation with the corespondent typing rule. The correspondence is quite intuitive; we highlight some important details. An empty set of variables is represented with the constant `mt`. We remark that the operator `/` stands for the context difference operation that removes some variables of a type context, whereas operator '`remove`' drops a variable of a variable set. In the equation for Rule [A-Out], we do not use the same variable `T` in the type associated to variable `a` and the type associated to value `v` as it would be expected, since the types are possibly infinite and there are many possible representations for the same infinite type. Instead, we use another variable `T'` and we check that `T` and `T'` are equivalent, using function `equal`.

We divide Rule [A-In] in two different equations for linear and unrestricted inputs. In the linear case, it is possible that the type of the subject `a` is linear or unrestricted; when the variable is linear it must be included in the returned set of linear variables. In the unrestricted case, the type of subject `a` is required to be unrestricted inasmuch as the attempt to use a linear variable in an unrestricted fashion must be rejected. Moreover, we require that the only free linear variable used in process `P` is `y{0}` (condition `remove(L, y{0}) == mt`).

4.3 Type Soundness

Vasconcelos [10] established that the type system for sπ is *sound*: a closed, well-typed process is guaranteed to have a well-defined behavior according to the ascribed protocols and the reduction semantics of the calculus. Also, the algorithmic type checking, as implemented in this section, is proven correct. With these elements in mind, we can integrate both the rewriting specification of the operational semantics and the implementation of the algorithmic type checking. This way, we only execute well-typed processes. For this purpose, we use two auxiliary functions `well-typed` and `erase`. The former checks whether a process does not have typing errors:

```
op well-typed : Trm -> Bool .
eq well-typed(P) = (type-term(nil, P) =/= ill-typed) .
```

Function `well-typed` applies the algorithm for type checking `type-term` over a process `P` and returns `true` when type-checking is successful, i.e. when the result is not `ill-typed`. Function `erase` proceeds inductively on the structure of a process; when it reaches an annotated subprocess '`new [x y : T] P`', it removes the annotation to produce '`new [x y] P`'—see Appendix B for details.

Correspondingly, we extend our specification of the reduction semantics to enable the execution of annotated processes, i.e., processes that use the operator $(\boldsymbol{\nu} xy : T)P$ instead of the operator $(\boldsymbol{\nu} xy)P$:

```
rl [TYPED] : new [x y : T] P => if well-typed(new [x y : T] P)
                  then erase(new [x y] P) else ill-typed-process fi .
```

We check whether process `new [x y : T] P` is well-typed; if so, we rewrite it as an equivalent process in which each occurrence of `new [x y : T]` is replaced by `new [x y]` through the function `erase`. Otherwise, process `new [x y : T] P` is rewritten as `ill-typed-process`, a constant that denotes that the process has a typing error and cannot be executed.

5 Lock and Deadlock Detection in Maude

Although the type system for $\mathsf{s}\pi$ given in [10] enables us to statically detect processes whose variables are used according to their ascribed protocols (expressed as session types), there are processes that are well-typed but that exhibit unwanted behaviors, in particular deadlocks. For example, consider the process

$$P = \overline{x_3}\mathsf{true}.\overline{x_1}\mathsf{true}.\overline{y_2}\mathsf{false}.\mathbf{0} \mid \mathsf{lin}\ y_3(z).\mathsf{lin}\ x_2(w).\mathsf{lin}\ y_1(t).\mathbf{0}$$

Process P is well-typed in a context $x_1 : \mathsf{lin}\ !\mathsf{bool}.\mathsf{end}, y_1 : \mathsf{lin}\ ?\mathsf{bool}.\mathsf{end}$, $x_2 : \mathsf{lin}\ ?\mathsf{bool}.\mathsf{end}, y_2 : \mathsf{lin}\ !\mathsf{bool}.\mathsf{end}, x_3 : \mathsf{lin}\ !\mathsf{bool}.\mathsf{end}, y_3 : \mathsf{lin}\ ?\mathsf{bool}.\mathsf{end}$. Then, process $(\boldsymbol{\nu} x_1 y_1 x_2 y_2 x_3 y_3)P$ can reduce but becomes deadlocked after such a synchronization, due to a circular dependency on variables x_1, y_1, x_2, y_2.

5.1 Definitions

Here we characterize deadlocks in $\mathsf{s}\pi$ and we show how we can use the rewrite specification of the operational semantics and the Maude tools for detecting processes with deadlocks. We follow the formulation of deadlock and lock freedom given by Padovani [3], which uses the notion of *pending communication.* We start by defining the reduction contexts $\mathcal{C}$:

$$\mathcal{C} ::= [\,] \quad \mid \quad (\mathcal{C} \mid P) \quad \mid \quad (\boldsymbol{\nu} xy)\mathcal{C}$$

The notion of pending communication in a process P with respect to variables x, y is defined with the following auxiliary predicates:

$$\begin{aligned}
\mathsf{in}(x,P) &\overset{\text{def}}{\iff} P \equiv \mathcal{C}[\mathsf{lin}\ x(y).Q] \ \wedge\ x \notin \mathtt{bn}(\mathcal{C}) \\
\mathsf{in}^*(x,P) &\overset{\text{def}}{\iff} P \equiv \mathcal{C}[\mathsf{un}\ x(y).Q] \ \wedge\ x \notin \mathtt{bn}(\mathcal{C}) \\
\mathsf{out}(x,P) &\overset{\text{def}}{\iff} P \equiv \mathcal{C}[\overline{x}v.Q] \ \wedge\ x \notin \mathtt{bn}(\mathcal{C}) \\
\mathsf{sync}(x,y,P) &\overset{\text{def}}{\iff} (\mathsf{in}(x,P) \ \vee\ \mathsf{in}^*(x,P)) \ \wedge\ \mathsf{out}(y,P) \\
\mathsf{wait}(x,y,P) &\overset{\text{def}}{\iff} (\mathsf{in}(x,P) \ \vee\ \mathsf{out}(y,P)) \ \wedge\ \neg\mathsf{sync}(x,y,P)
\end{aligned}$$

There, we assume the extension of function $\mathtt{bn}(.)$ to reduction contexts. Intuitively, the first three predicates express the existence of a pending communication on a variable x. More in details:

- Predicate $\mathsf{in}(x,P)$ holds if x is free in P and there is a subprocess of P that is able to make a linear input on x. Predicate $\mathsf{in}^*(x,P)$ is its analog for unrestricted inputs.
- Predicate $\mathsf{out}(x,P)$ holds if x is free in P and a subprocess of P is waiting to send a value v.
- Predicate $\mathsf{sync}(x,y,P)$ denotes a pending input on x for which a synchronization on y is immediately possible.
- Predicate $\mathsf{wait}(x,y,P)$ denotes a pending input/output for which a synchronization on x, y is not immediately possible.

Let us write $\longrightarrow^*$ to denote the reflexive, transitive closure of $\longrightarrow$. Also, write $P \not\rightarrow$ if there is no Q such that $P \longrightarrow Q$. With these elements, we now proceed to characterize the deadlock and lock freedom properties. We say process P is

- *deadlock free* if for every Q such that $P \longrightarrow^* (\boldsymbol{\nu} x_1 y_1)(\boldsymbol{\nu} x_2 y_2)\dots(\boldsymbol{\nu} x_n y_n)Q \not\rightarrow$ it holds that $\neg\mathsf{wait}(x_i,y_i,Q)$ for every x_i.
- *lock free* if for every Q such that $P \longrightarrow^* (\boldsymbol{\nu} x_1 y_1)(\boldsymbol{\nu} x_2 y_2)\dots(\boldsymbol{\nu} x_n y_n)Q$ and $\mathsf{wait}(x_i,y_i,Q)$ there exists R such that $Q \longrightarrow^* R$ and $\mathsf{sync}(x_i,y_i,R)$ hold.

This way, a process is deadlock free if there are not stable states with pending inputs or outputs; a process is lock free if it is able to eventually perform a synchronization in any pending input or output.

We can use Maude to verify deadlock freedom and lock freedom for typed processes. Indeed, we can use the reachability tool `search` and the LTL model checker `modelCheck`. We first represent the previous predicates over process terms as functions in Maude over instances of the sorts `Trm` and `Chan`:

```
ops in out in* : Chan Trm -> Bool .
ops sync wait : Chan Chan Trm -> Bool .
op wait-aux : QidSet Trm -> Bool .
eq in(a, lin a(x) . Q | R) = true .
eq in(a, P) = false [owise] .
eq in*(a, un a(x) . Q | R) = true .
eq in*(a, P) = false [owise] .
```

```
eq out(a, a < v > . Q | R) = true .
eq out(a, P) = false [owise] .
eq sync(a, b, P) = (in(a, P) or in*(a, P)) and out(b, P) .
eq wait(a, b, P) = (in(a, P) or out(b, P)) and not sync(a, b, P) .
eq wait-aux(mt, P) = false .
eq wait-aux((x y) nl, P) = wait(x{0}, y{0}, P) or
   wait(y{0}, x{0}, P) or wait-aux(nl, P) .
```

Above, we use function `wait-aux` to determine if a group of pairs of co-variables contains a pair for which there is a pending communication.

The deadlock freedom property imposes that there should be no stable states in which there are pending communications. Consequently, we can use the Maude command `search` as follows to determine whether a process is deadlock free:

```
search init =>!
         new* [nl:QidSet] P:Trm such that wait-aux(nl:QidSet, P:Trm) .
```

where `init` denotes for the process to be checked. We recall that the `search` command with the arrow `=>!` looks for final (stable) states. In this way, `init` is deadlock free if the search returns no solution.

For the lock freedom property, we can not use the reachability tool since this property requires the checking some intermediate states. Consequently, we represent the lock freedom property as an LTL formula and use the built-in LTL model checker in Maude. Below, we define the Maude predicates `psync` and `pwait` that we will use in the LTL model checker:

```
ops pwait psync : Chan Chan -> Prop [ctor] .
eq new* [(x y) nl] P |= pwait(x{0}, y{0}) =
                        wait(x{0}, y{0}, P) or wait(y{0}, x{0}, P) .
eq new* [(x y) nl] P |= psync(x{0}, y{0}) =
                        sync(x{0}, y{0}, P) or sync(y{0}, x{0}, P) .
```

In the predicates `psync` and `pwait`, we use normalized processes, i.e., processes where the nested scope restrictions are flattened in an equivalent process that uses the operator `new*`. This assumption simplifies the definitions. Both `psync` and `pwait` predicates use the functions `in`, `in*`, `out`, `sync` and `wait` as expected according to the definition.

The Kripke structure that is generated for Maude will use such normalized process term as states. The Maude predicates `pwait` and `psync` hold with respect to a pair of dual variables if there is a pending communication and there is a synchronization in the process associated to a state. The lock freedom property imposes for each variable that if in any state there is a pending communication then eventually there will be a synchronization. Formalizing the lock freedom property requires to check each possible subject. For that reason, the LTL formula associated to this property depends on the variables being used in the process. We define a function `build-lock-formula` that takes the used variables and builds the corresponding LTL formula as follows:

```
ops P1 P2 P3 P4 P5 : -> Trm .
eq P1 = new* [('y1' 'x1')('y2' 'x2')('y3' 'x3')]
      ('x3'{0} < True > . 'x1'{0} < True > . 'y1'{0} < False > . nil |
       lin 'y3'{0}('z') . lin 'y2'{0}('x') . lin 'x2'{0}('w') . nil) .
eq P2 = new* [('x1' 'y1')('x2' 'y2')('a' 'b')]
             ('x1'{0} < 'b'{0} > . nil | 'a'{0} < True > . nil |
              un 'y1'{0}('z') . 'x2'{0} < 'z'{0} > . nil |
              un 'y2'{0}('w') . 'x1'{0} < 'w'{0} > . nil ) .
```

Fig. 5. Processes in Maude

```
op build-lock-formula : QidSet -> Formula .
eq build-lock-formula(mt) = True .
eq build-lock-formula((x y) nl) =
     [] (<> pwait(x{0}, y{0}) -> <> psync(x{0}, y{0})) /\
        build-lock-formula(nl) .
```

This way, the resulting LTL formula corresponds to the conjunction of subformulas associated to each dual variable. The model checker can be used as follows:

```
red modelCheck(init, build-lock-formula(vars)) .
```

where `init` stands for the process term and `vars` stands for a set of pairs of co-variables. If the `init` is lock-free then the invocation of `modelCheck` will produce `true`. Otherwise, the invocation will show a counterexample with a sequence of rules that produces a state where the formula is not fulfilled.

5.2 Examples

We give a couple of examples of well-typed processes in sπ, with different lock- and deadlock-freedom properties. (Appendix C presents additional examples.)

$$P_1 = (\boldsymbol{\nu} x_1 y_1)(\boldsymbol{\nu} x_2 y_2)(\boldsymbol{\nu} x_3 y_3)(\overline{x_3}\mathsf{true}.\overline{x_1}\mathsf{true}.\overline{y_1}\mathsf{false}.\mathbf{0} \mid \mathsf{lin}\ y_3(z).\mathsf{lin} y_2(x).\mathsf{lin} x_2(w).\mathbf{0})$$
$$P_2 = (\boldsymbol{\nu} x_1 y_1)(\boldsymbol{\nu} x_2 y_2)(\boldsymbol{\nu} ab)(\overline{x_1}b.\mathbf{0} \mid \overline{a}\mathsf{true}.\mathbf{0} \mid \mathsf{un}\ y_1(z).\overline{x_2}z.\mathbf{0} \mid \mathsf{un}\ y_2(w).\overline{x_1}w.\mathbf{0})$$

Process P_1 is a simple process that reduces to a deadlock immediately after a synchronization on the co-variables x_3, y_3. Process P_2 represents an infinite process where the variable b is repeatedly shared through communications on x_1, y_1, x_2, y_2. The process is a not lock-free: b is never used to synchronize with its co-variable a. Fig. 5 gives the Maude terms associated to these processes.

We analyze `P1` using Maude by executing:

```
search P1 =>! new* [nl:QidSet] P:Trm
                    such that wait-aux(nl:QidSet, P:Trm) .
red modelCheck(P1,
        build-lock-formula(('y1' 'x1')('y2' 'x2')('y3' 'x3'))) .
```

We obtain the following results, which confirm that `P1` is not deadlock free and not lock free:

```
search in TEST : P1 =>! new*[nl:QidSet]P:Trm
                      such that wait-aux(nl:QidSet, P:Trm) = true .
Solution 1 (state 1)
nl:QidSet --> ('x3' 'y3') ('y1' 'x1') 'y2' 'x2'
P:Trm --> 'x1'{0} < True > . 'y1'{0} < False > . nil |
           lin 'y2'{0}('x') . lin 'x2'{0}('w') . nil

No more solutions.

result ModelCheckResult: counterexample(
   {new*[('x3' 'y3') ('y1' 'x1') 'y2' 'x2']
      'x3'{0} < True > . 'x1'{0} < True > . 'y1'{0} < False > . nil |
       lin 'y3'{0}('z') . lin 'y2'{0}('x') . lin 'x2'{0}('w') . nil,
    'LINCOM},
   {new*[('x3' 'y3') ('y1' 'x1') 'y2' 'x2']
      'x1'{0} < True > . 'y1'{0} < False > . nil |
       lin 'y2'{0}('x') . lin 'x2'{0}('w') . nil,
    deadlock})
```

Consider now a similar execution for process P2:

```
search P2 =>! new* [nl:QidSet] P:Trm
                      such that wait-aux(nl:QidSet, P:Trm) .
red modelCheck(P2, build-lock-formula(('x1' 'y1')('x2' 'y2')('a' 'b'))) .
```

We obtain the following results, which confirm that P2 is an infinite process that is deadlock free but not lock free:

```
search in TEST : P2 =>! new*[nl:QidSet]P:Trm
                       such that wait-aux(nl:QidSet, P:Trm) = true .

No solution.

result ModelCheckResult: counterexample(nil,
   {new*[('a' 'b') ('x1' 'y1') 'x2' 'y2']
       'a'{0} < True > . nil | 'x1'{0} < 'b'{0} > . nil |
       un 'y1'{0}('z') . 'x2'{0} < 'z'{0} > . nil |
       un 'y2'{0}('w') . 'x1'{0} < 'w'{0} > . nil, 'UNCOM}
   {new*[('a' 'b') ('x1' 'y1') 'x2' 'y2']
       'a'{0} < True > . nil | 'x2'{0} < 'b'{0} > . nil |
       un 'y1'{0}('z') . 'x2'{0} < 'z'{0} > . nil |
       un 'y2'{0}('w') . 'x1'{0} < 'w'{0} > . nil, 'UNCOM})
```

6 Closing Remarks

In this paper, we have reported on an executable specification in Maude of the operational semantics and the associated algorithmic type-checking of $\mathsf{s}\pi$, a session-typed π-calculus proposed by Vasconcelos in [10]. We integrated both specifications closely following his formulation. To our knowledge, ours is the first Maude implementation of a session-typed process language. Because typing in [10] does not exclude deadlocks, we leverage built-in tools in Maude and executable specifications to detect well-typed dead-locked processes. In our view, these developments establish a promising starting point to the automated analysis of message-passing concurrency specifications.

As future work, we intend to adapt our equational theories to leverage the confluence checker tool available in Maude. Additionally, we expect to extend our executable specifications to perform behavioral analysis of the processes that implement *multiparty session types*, in the spirit of [7]. Likewise, we aim to explore the automated analysis of communication correctness of an extension of $\mathsf{s}\pi$ with *higher-order* process communication, in which values can be abstractions (functions from names to processes).

Acknowledgements This work has been partially supported by the Dutch Research Council (NWO) under project No. 016.Vidi.189.046 (Unifying Correctness for Communicating Software).

We are grateful to Camilo Rocha for his useful suggestions on this work, and to the anonymous reviewers for their careful reading and constructive remarks.

References

1. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All about Maude - a High-Performance Logical Framework: How to Specify, Program and Verify Systems in Rewriting Logic. Springer-Verlag, Berlin, Heidelberg (2007), `https://doi.org/10.1007/978-3-540-71999-1`
2. Franco, J., Vasconcelos, V.T.: A concurrent programming language with refined session types. In: Counsell, S., Núñez, M. (eds.) Software Engineering and Formal Methods - SEFM 2013 Collocated Workshops: BEAT2, WS-FMDS, FM-RAIL-Bok, MoKMaSD, and OpenCert, Madrid, Spain, September 23-24, 2013, Revised Selected Papers. Lecture Notes in Computer Science, vol. 8368, pp. 15–28. Springer (2013), `https://doi.org/10.1007/978-3-319-05032-4_2`
3. Padovani, L.: Deadlock and lock freedom in the linear π-calculus. In: Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS). CSL-LICS '14, Association for Computing Machinery, New York, NY, USA (2014), `https://doi.org/10.1145/2603088.2603116`
4. Pitsiladis, G., Stefaneas, P.: Implementation of Privacy Calculus and Its Type Checking in Maude: 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part II, pp. 477–493 (11 2018), `https://doi.org/10.1007/978-3-030-03421-4_30`
5. Ramírez Restrepo, C.A., Pérez, J.A.: Executable rewriting semantics for session-based concurrency (Extended Version). Tech. rep., Universidad Javeriana / University of Groningen (2022), `www.jperez.nl`
6. Sangiorgi, D., Walker, D.: The Pi-Calculus: A Theory of Mobile Processes. Cambridge University Press (2003), `http://books.google.com/books?id=QkBL_7VtiPgC`

7. Scalas, A., Yoshida, N.: Less is more: multiparty session types revisited. Proc. ACM Program. Lang. **3**(POPL), 30:1–30:29 (2019), `https://doi.org/10.1145/3290343`
8. Stehr, M.: CINNI - A generic calculus of explicit substitutions and its application to λ-, ς- and π- calculi. In: Futatsugi, K. (ed.) The 3rd International Workshop on Rewriting Logic and its Applications, WRLA 2000, Kanzawa, Japan, September 18-20, 2000. Electronic Notes in Theoretical Computer Science, vol. 36, pp. 70–92. Elsevier (2000), `https://doi.org/10.1016/S1571-0661(05)80125-2`
9. Thati, P., Sen, K., Martí-Oliet, N.: An executable specification of asynchronous pi-calculus semantics and may testing in Maude 2.0. Electron. Notes Theor. Comput. Sci. **71**, 261–281 (2002), `https://doi.org/10.1016/S1571-0661(05)82539-3`
10. Vasconcelos, V.T.: Fundamentals of session types. Inf. Comput. **217**, 52–70 (2012), `https://doi.org/10.1016/j.ic.2012.05.002`
11. Verdejo, A., Martí-Oliet, N.: Implementing CCS in Maude 2. In: Gadducci, F., Montanari, U. (eds.) Proceedings Fourth International Workshop on Rewriting Logic and its Applications, WRLA 2002, Pisa, Italy, September 19–21, 2002. Electronic Notes in Theoretical Computer Science, vol. 71, pp. 239–257. Elsevier (2002), `https://doi.org/10.1016/S1571-0661(05)82540-X`
12. Viry, P.: Input/output for ELAN. Electronic Notes in Theoretical Computer Science **4**, 51–64 (1996), `https://doi.org/10.1016/S1571-0661(04)00033-7`, first International Workshop on Rewriting Logic and its Applications, WRLA 1996, Asilomar, USA, 1996
13. Viry, P.: A rewriting implementation of pi-calculus. Tech. rep. (1996)

A Operational Correspondence

Here we prove that the transition system associated to the rewrite theory in our Maude specification coincides with the reduction semantics for $\mathsf{s}\pi$. Given an $\mathsf{s}\pi$ process P, we use the notation $[\![P]\!]$ to denote its representation in Maude. We state the operational correspondence between them with two theorems, completeness and soundness:

Theorem 1 (Completeness). *Let P an $\mathsf{s}\pi$ process and let $[\![P]\!]$ be its corresponding representation in the rewrite theory (Σ, E, ϕ, R) of Section 3. Then, if $P \longrightarrow P'$ then there is a rewriting rule $l : [\![P]\!] \to [\![P']\!]$ that can be applied.*

Proof. By induction on the reduction $P \longrightarrow P'$, with a case analysis on the last rule being applied. We have three base cases, corresponding to the forms of direct communication in the calculus (Rules [R-LinCom], [R-UnCom], and [R-Case]) and three inductive cases (Rules [R-Par], [R-Res] and [R-Struct]). For each case, we deepen in the correspondence with a rewriting rule $l : [\![P]\!] \to [\![P']\!]$.

1. Rule [R-LinCom]: This rule in the operational semantics of the calculus (see Fig. 2) is stated as:

$$\frac{}{(\boldsymbol{\nu} xy)(\overline{x}v.P \mid \mathsf{lin}\ y(z).Q \mid R) \longrightarrow (\boldsymbol{\nu} xy)(P \mid Q[v/z] \mid R)}$$

   Then, we must show that there is a rewrite rule that corresponds to this rule, i.e., we need to determine a rewrite rule in our rewrite theory such that:

$$[\![(\boldsymbol{\nu} xy)(\overline{x}v.P \mid \mathsf{lin}\ y(z).Q \mid R)]\!] \longrightarrow [\![(\boldsymbol{\nu} xy)(P \mid Q[v/z] \mid R)]\!]$$

   The correspondence with the rewriting rule labeled `LINCOM` is quite intuitive:

```
new* [(x y) nl] x{N} < v > . P | lin y{N}(z) . Q | R =>
                              new* [(x y) nl] P | [z := v] Q | R .
```

   Clearly, it holds that

   $[\![(\boldsymbol{\nu} xy)(\overline{x}v.P \mid \mathsf{lin}\ y(z).Q \mid R)]\!] =$
   `new* [(x y) nl] x{N} < v > . P | lin y{N}(z) . Q | R`

   and

   $[\![(\boldsymbol{\nu} xy)(P \mid Q[v/z] \mid R)]\!] =$ `new* [(x y) nl] P | [z := v] Q | R`

   where `nl` = `mt`. It is easy to check that in virtue of the mathematical induction, any possible reduction involving subprocess R corresponds to a different application of some reduction rule.
2. Rule [R-UnCom]: Then the reduction proceeds as follows:

$$\frac{}{(\boldsymbol{\nu} xy)(\overline{x}v.P \mid \mathsf{un}\ y(z).Q \mid R) \longrightarrow (\boldsymbol{\nu} xy)(P \mid Q[v/z] \mid \mathsf{un}\ x(y).Q \mid R)}$$

   Again, we must show a corresponding rule in our rewrite theory such that:

$$[\![(\boldsymbol{\nu} xy)(\overline{x}v.P \mid \mathsf{un}\ y(z).Q \mid R)]\!] \longrightarrow [\![(\boldsymbol{\nu} xy)(P \mid Q[v/z] \mid \mathsf{un}\ x(y).Q \mid R)]\!]$$

   The correspondence with the rewrite rule `UNCOM` arises immediately:

```
new* [(x y) nl] x{N} < v > . P | un y{N}(z) . Q | R =>
          new* [(x y) nl] P | [z := v] Q | un y{N}(z) . Q | R .
```

Then, it is evident that

$[\![(\boldsymbol{\nu} xy)(\overline{x}v.P \mid \mathsf{un}\ y(z).Q \mid R)]\!] =$
`new* [(x y) nl] xN < v > . P | un yN(z) . Q | R`

and

$[\![(\boldsymbol{\nu} xy)(P \mid Q[v/z] \mid \mathsf{un}\ x(y).Q \mid R)]\!] =$
`new* [(x y) nl] P | [z := v] Q | un yN(z) . Q | R`

where `nl` = `mt`. It is easy to check that in virtue of the mathematical induction, any possible reduction involving subprocess R corresponds to a different application of some reduction rule.

3. Rule [R-Case] Then the reduction proceeds as follows:

$$\frac{j \in I}{(\boldsymbol{\nu} xy)(x \triangleleft l_j.P \mid y \triangleright \{l_i : Q_i\}_{i\in I} \mid R) \longrightarrow (\boldsymbol{\nu} xy)(P \mid Q_j \mid R)}$$

As in the previous cases, there is also a quite intuitive correspondence with the rule `CASE` in the rewrite theory:

```
new* [(x y) nl] (x{N} << w . P) | (y{N} >> { (w : Q) CH }) | R =>
             new* [(x y) nl] P | Q | R .
```

As in the other cases, any possible reduction involving subprocess R corresponds to a different application of some reduction rule.

4. Rules [R-Par], [R-Res]: These rules capture the compositionality of the operational semantics but in themselves they do not express any additional alternative of reduction. The effect of these rules is obtained for free in Maude as an effect of the equational theory underlying the rewrite theory and due to the rewriting rules in a system module of Maude allow to rewrite specific subterms of a term.
   - Rule [R-Par]: This rule in the operational semantics of the calculus is stated as:

$$\frac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q}$$

     We assume as inductive hypothesis that for a process $P \longrightarrow P'$ it holds that there is a rewriting rule $l : [\![P]\!] \rightarrow [\![P']\!]$ that can be applied. We must show that there is a rewrite rule such that:

$$[\![P \mid Q]\!] \longrightarrow [\![P' \mid Q]\!]$$

     Now, given that the rewriting rules in a system module of Maude allow to rewrite specific subterms of a term, due to the Maude term $[\![P \mid Q]\!]$ contains the subterm $[\![P]\!]$, then the term $[\![P \mid Q]\!]$ will be rewritten to $[\![P' \mid Q]\!]$ by means the application of the same rewrite rule l.

– Similarly, in the case of the Rule [R-Res], which is stated as:

$$\frac{P \longrightarrow P'}{(\boldsymbol{\nu} xy)P \longrightarrow (\boldsymbol{\nu} xy)P'}$$

We also assume as inductive hypothesis that for a process $P \longrightarrow P'$ we have a rewriting rule $l : [\![P]\!] \rightarrow [\![P']\!]$ that can be applied. We must show that there is a rewrite rule such that:

$$[\![(\boldsymbol{\nu} xy)P]\!] \longrightarrow [\![(\boldsymbol{\nu} xy)P']\!]$$

Again, the Maude term $[\![(\boldsymbol{\nu} xy)P]\!]$ includes the subterm $[\![P]\!]$ and consequently, the term $[\![(\boldsymbol{\nu} xy)P]\!]$ will be rewritten to $[\![(\boldsymbol{\nu} xy)P']\!]$ by using the rewrite rule l.

5. Rule [R-Struct]: This rule captures the effect of the operational semantics modulo the structural congruence relation. In this way, any possible reduction of a process P such that $P \equiv Q$ is also possible for process Q. This rule is clearly correspondent to the rewrite rule `FLAT`:

```
P => P' if P' := flatten(P) /\ P =/= P' .
```

We recall that the function `flatten` produces an equivalent process where the occurrences of the scope restriction operator are taken to the top-level.

□

Theorem 2 (Soundness). *Let* `T`, `T'` *be instances of the sort* `Trm` *in the rewrite theory* (Σ, E, ϕ, R) *of Section 3 and* $\mathtt{T}^{C}$ *be the canonical form of* `T`*. Then, if there is a rewrite rule* $l : \mathtt{T}^{C} \rightarrow \mathtt{T}'$ *that can be applied then there exist processes* P*,* Q *such that* $\mathtt{T} = [\![P]\!]$*,* $\mathtt{T}' = [\![Q]\!]$*, and:*

– $P \equiv Q$ *or*
– $P \longrightarrow Q$

The rewriting rule $l : \mathtt{T} \rightarrow \mathtt{T'}$ *can be preceded for a sequence of applications of some equations in the equational theory underlying the rewrite theory.*

Proof. By a case analysis on the rewriting rule being applied over the Maude term `T` and the correspondence with an $\mathsf{s}\pi$ reduction. We have four cases corresponding to the rewrite rules `FLAT`, `LINCOM`, `UNCOM`, and `CASE` in the rewrite theory. For each case, we deepen in the correspondence with a specific reduction over processes P, Q related to the Maude term being rewritten. We remark that there is a one-to-one correspondence between the process terms (see Section 2) and the instances of the Maude sort `Trm` (see Section 3). For this reason, for each Maude term `P` there is a process P such that $\mathtt{P} = [\![P]\!]$.

1. Rule `FLAT`: This rule is stated in Maude (see Section 3) as:

```
crl [FLAT] : P => P' if P' := flatten(P) /\ P =/= P' .
```

Here, the Maude term `P` is rewritten as the term `P'`, which is obtained by the function `flatten`. As already mentioned, the function `flatten` produces an equivalent process where the occurrences of the scope restriction operator are taken to the top-level. Consequently, the Maude term `P'` is the Maude representation of a process Q such that $Q \equiv P$, as expected.

2. Rule `LINCOM`: This rule is stated in Maude (see Section 3) as:

```
new* [(x y) nl] x{N} < v > . P | lin y{N}(z) . Q | R =>
                           new* [(x y) nl] P | [z := v] Q | R .
```

Again, in virtue of the one-to-one correspondence among the process terms syntax and the sort and constructors in the equational theory, we have that there exist processes P and Q such that:

$$[\![P]\!] = \texttt{new* [(x y) nl] x\{N\} < v > . P | lin y\{N\}(z) . Q | R}$$
$$[\![Q]\!] = \texttt{new* [(x y) nl] P | [z := v] Q | R}$$

Clearly, process P must be equivalent to a process matching a pattern $(\boldsymbol{\nu}xy)\ldots\overline{x}v.P' \mid \mathsf{lin}\ y(z).Q' \mid R$ and process Q must be equivalent to a process matching a pattern $(\boldsymbol{\nu}xy)\ldots P' \mid Q' \mid R$. In consequence, it is easy to check that $P \longrightarrow Q$, as expected.

3. Rule `UNCOM`: These rule is stated as follows (see Section 3):

```
new* [(x y) nl] x{N} < v > . P | un y{N}(z) . Q | R =>
           new* [(x y) nl] P | [z := v] Q | un y{N}(z) . Q | R .
```

As before, it is easy to check that there exist processes P and Q for which holds:

$$[\![P]\!] = \texttt{new* [(x y) nl] x\{N\} < v > . P | un y\{N\}(z) . Q | R}$$
$$[\![Q]\!] = \texttt{new* [(x y) nl] P | [z := v] Q | un y\{N\}(z) . Q | R}$$

Then, process P is of the form $(\boldsymbol{\nu}xy)\ldots\overline{x}v.P' \mid \mathsf{un}\ y(z).Q' \mid R$ and process Q is of the form $(\boldsymbol{\nu}xy)\ldots P' \mid Q' \mid \mathsf{un}\ y(z).Q' \mid R$ and $P \longrightarrow Q$, as expected.

4. Rule `CASE`: This rule is stated in the rewrite theory as:

```
new* [(x y) nl] (x{N} << w . P) | (y{N} >> { (w : Q) CH }) | R =>
             new* [(x y) nl] P | Q | R .
```

Once again, in virtue of the one-to-one correspondence among the process term syntax and the Maude sorts and constructors, we have that there exist processes P and Q such that:

$$[\![P]\!] = \texttt{new* [(x y) nl] (x\{N\} << w . P) | (y\{N\} >> (w : Q) CH) | R}$$
$$[\![Q]\!] = \texttt{new* [(x y) nl] P | Q | R}$$

Process P is equivalent to a process matching the pattern $(\boldsymbol{\nu}xy)\ldots x \triangleleft l_j.P' \mid y \triangleright \{l_i : Q_i\}_{i \in I} \mid R$ and process Q is equivalent to a process matching the pattern $(\boldsymbol{\nu}xy)\ldots P' \mid Q_j \mid R$. Thereafter, we have that $P \longrightarrow Q$, as expected.

$\square$

$$\overline{\mathsf{end}} = \mathsf{end} \qquad \overline{a} = a \qquad \overline{\mu a.T} = \mu a.\overline{T}$$

$$\overline{q?T.U} = q!T.\overline{U} \qquad \overline{q!T.U} = q?T.\overline{U}$$

$$\overline{q \oplus \{l_i : T_i\}_{i \in I}} = q\&\{l_i : \overline{T_i}\}_{i \in I} \qquad \overline{q\&\{l_i : T_i\}_{i \in I}} = q \oplus \{l_i : \overline{T_i}\}_{i \in I}$$

Fig. 6. Definition of dual function

$$\frac{x : U \notin \Gamma}{\Gamma + x : U = \Gamma, x : U} \qquad \frac{\mathsf{un}(T)}{(\Gamma, x : T) + x : T = (\Gamma, x : T)}$$

$$\Gamma \div \emptyset = \Gamma \qquad \frac{\Gamma_1 \div L = \Gamma_2, x : T \qquad \mathsf{un}(T)}{\Gamma_1 \div (L, x) = \Gamma_2} \qquad \frac{\Gamma_1 \div L = \Gamma_2 \qquad x \notin \mathtt{dom}(\Gamma_2)}{\Gamma_1 \div (L, x) = \Gamma_2}$$

Fig. 7. Definition of context update (up) and context difference (down)

B Omitted Material on Algorithmic Type Checking

The duality operation on session types is essential to enforce communication correctness; it is defined in Fig. 6. In Maude, the operation `dual`, given next, enables us to obtain the dual of a type. It takes an instance of the sort `Type` and produces an instance of the same sort:

```
op dual : Type -> Type . op dual-aux : ChoiceTset -> ChoiceTset .
eq dual(end) = end . eq dual(var(x)) = var(x) .
eq dual(q ? T1 . T2) = q ! T1 . dual(T2) .
eq dual(q ! T1 . T2) = q ? T1 . dual(T2) .
eq dual(u [x] T) = u [x] dual(T) .
eq dual(q +{ CHT }) = q &{ dual-aux(CHT) } .
eq dual(q &{ CHT }) = q +{ dual-aux(CHT) } .
eq dual-aux(empty) = empty .
eq dual-aux((x : T) CHT) = (x : dual(T)) dual-aux(CHT) .
```

Each one the previous equations stands for a specific case in the definition of the dual function in Fig. 6. The auxiliary operator `dual-aux` allows to apply the dual function to each subtype in branching and selection types.

Another important operation in the algorithmic type checking is *context update*, denoted '+'. This operation enables us to extend a type context with a new type for a variable; it is used when checking the type of input, output, branching, and selection processes. The formal definition is shown in Fig. 7 (top). There are two rules: the first one requires linear variables not to be in the context and the second one imposes updating an unrestricted variable is only possible if its type is unchanged.

The context update operation is implemented in Maude by means of the operator `_+_:_`, which takes a type context, a value and a type and produces a new context.

```
op _+_:_ : Context Value Type -> Context .
ceq C + v : T = (C, (v : T)) if not v in C .
ceq (C, (v : T1)) + v : T2 = (C, (v : T2’))
   if unrestricted(T1) /\ T1’ := unfold(T1) /\
      T2’ := unfold(T2) /\ equal(T1’, T2’) .
eq C + v : T = invalid-context [owise] .
```

Each rule is associated to a previous equation and it proceeds as expected. Particularly, for the case of unrestricted variables, given that some types are infinite then before the update the unrestricted types are unfolded (cf. operation `unfold`). Type unfolding is the mechanism that we use to deal with infinite types. Given a type T, it is defined as follows: if T is a recursive type $\mu a.U$ then unfolding means performing the substitution $U[\mu a.U/a]$. Otherwise, T remains unaltered. The types `T1` and `T2` involved in the update operation are unfolded and the resulting types must be equal. Lastly, we add an additional equation to produce the constant `invalid-context` when the context update can not be performed.

The *context difference* function, denoted ‘$\div$’, is the mechanism that prevents the use of linear variables in several threads. This operation removes the variables in a set from a typing context. This function is defined inductively as shown in Fig. 7 (bottom). It is implemented in Maude as follows:

```
op _/_ : Context Chanset -> Context .
eq C / mt = C .
eq ((a : T), C) / (a L1) = C / L1 .
eq C / L1 = C [owise] .
```

On the other hand, the function `erase` (used in Section 4.3) inductively analyzes a process; when it reaches an annotated subprocess (i.e. a process `new [x y : T] P`), it returns a non-annotated process (i.e., a process `new [x y] P`). This function is defined in Maude as follows:

```
op  erase : Trm -> Trm .
op  erase-aux : Choiceset -> Choiceset .
eq  erase(nil) = nil .
eq  erase(a < v > . P) = a < v > . erase(P) .
eq  erase(q a(x) . P) = q a(x) . erase(P) .
eq  erase(if v then P else Q fi) = if v then erase(P) else erase(Q) fi .
ceq erase(P | Q) = erase(P) | erase(Q) if P =/= nil and Q =/= nil .
eq  erase(a >> {CH}) = a >> { erase-aux(CH) } .
eq  erase(a << x . P) = a << x . erase(P) .
eq  erase-aux(empty) = empty .
eq  erase-aux((x : P) CH) = (x : erase(P)) erase-aux(CH) .
```

```
eq  erase(new [x y] P) = new [x y] erase(P) .
eq  erase(new [x y : T] P) = new [x y] erase(P) .
```

C Additional Examples of Lock and Deadlock Detection

C.1 Examples

We now present a few examples of well-typed processes in $\mathsf{s}\pi$, with different lock- and deadlock-freedom properties:

$$P_1 = (\boldsymbol{\nu} x_1y_1)(\boldsymbol{\nu} x_2y_2)(\boldsymbol{\nu} x_3y_3)(\overline{x_3}\mathsf{true}.\overline{x_1}\mathsf{true}.\overline{y_1}\mathsf{false}.\mathbf{0} \mid \mathsf{lin}\ y_3(z).\mathsf{lin}y_2(x).\mathsf{lin}x_2(w).\mathbf{0})$$
$$P_2 = (\boldsymbol{\nu} x_1y_1)(\boldsymbol{\nu} x_2y_2)(\boldsymbol{\nu} ab)(\overline{x_1}b.\mathbf{0} \mid \overline{a}\mathsf{true}.\mathbf{0} \mid \mathsf{un}\ y_1(z).\overline{x_2}z.\mathbf{0} \mid \mathsf{un}\ y_2(w).\overline{x_1}w.\mathbf{0})$$
$$P_3 = (\boldsymbol{\nu} zw)(\boldsymbol{\nu} xy)(\overline{z}x.\mathsf{lin}\ y(v).\mathbf{0} \mid \mathsf{lin}\ w(t).\overline{t}\mathsf{true}.\mathbf{0})$$
$$P_4 = (\boldsymbol{\nu} x_1y_1)(\boldsymbol{\nu} x_2y_2)(\boldsymbol{\nu} x_3y_3)(\boldsymbol{\nu} x_4y_4)(\boldsymbol{\nu} x_5y_5)(\boldsymbol{\nu} x_6y_6)$$
$$\big(\overline{x_2}\mathsf{true}.\mathsf{un}\ x_1(w).\overline{w}x_4.\overline{w}y_4.\mathbf{0} \mid \mathsf{lin}\ y_2(b).y_4(z).\overline{x_4}a.\mathbf{0} \mid \mathsf{lin}\ y_6(a).\mathsf{lin}\ y_5('b').\mathbf{0} \mid$$
$$\overline{y_1}x_3.\mathsf{lin}\ y_3(z).y_3(t).\overline{z}a.t(c).\overline{x_5}\mathsf{true}.\overline{x_6}\mathsf{false}.\mathbf{0}\big)$$
$$P_5 = (\boldsymbol{\nu} x_1y_1)(\boldsymbol{\nu} x_2y_2)(\boldsymbol{\nu} x_3y_3)(\boldsymbol{\nu} x_4y_4)(\boldsymbol{\nu} x_5y_5)$$
$$\big(\overline{x_1}x_2.y_2 \triangleright \{a : \overline{x_3}\mathsf{true}.\overline{x_4}\mathsf{false}.\mathsf{lin}\ y_4(c).\mathbf{0}\ \ b : \overline{x_3}\mathsf{false}.\overline{y_5}x_4.\mathsf{lin}\ y_4(c).\mathbf{0}\} \mid$$
$$\mathsf{un}\ y_1(z).z \triangleleft a.\mathsf{lin}\ y_3(c).\mathbf{0} \mid \mathsf{lin}\ y_1(w).w \triangleleft b.\mathsf{lin}\ y_3(c).\mathsf{lin}\ x_5(t).\overline{t}\mathsf{true}.\mathbf{0}\big)$$

Processes P_1 and P_2 were discussed in the main text. Some intuitions for the other processes follow:

- Process P_3 is a simple lock and deadlock free process.
- Process P_4 is not deadlock free and not lock free. The deadlock is reached after some synchronizations in the pairs of co-variables x_1, y_1, x_2, y_2, x_3, y_3 and x_4, y_4.
- Process P_5 has a branching subprocess where a deadlock can be reached by selecting the branch with label a.

Fig. 8 gives the Maude terms associated to these processes.

C.2 Detecting (dead)locks in Maude

We give the execution of both commands with respect to process P3, which is deadlock and lock free:

```
search P3 =>! new* [nl:QidSet] P:Trm
                      such that wait-aux(nl:QidSet, P:Trm) .
red modelCheck(P3, build-lock-formula(('z' 'w') ('x' 'y'))) .
```

We obtain:

```
search in TEST : P3 =>! new*[nl:QidSet]P:Trm
                       such that wait-aux(nl:QidSet, P:Trm) = true .

No solution.

reduce in TEST : modelCheck(P3, build-lock-formula(('x' 'y') 'z' 'w')) .
result Bool: true
```

```
ops P1 P2 P3 P4 P5 : -> Trm .
eq P1 = new* [('y1' 'x1')('y2' 'x2')('y3' 'x3')]
      ('x3'{0} < True > . 'x1'{0} < True > . 'y1'{0} < False > . nil |
       lin 'y3'{0}('z') . lin 'y2'{0}('x') . lin 'x2'{0}('w') . nil) .
eq P2 = new* [('x1' 'y1')('x2' 'y2')('a' 'b')]
             ('x1'{0} < 'b'{0} > . nil | 'a'{0} < True > . nil |
              un 'y1'{0}('z') . 'x2'{0} < 'z'{0} > . nil |
              un 'y2'{0}('w') . 'x1'{0} < 'w'{0} > . nil ) .
eq P3 = new* [('z' 'w')('x' 'y')]
             ('z'{0}< 'x'{0} > . lin 'y'{0}('v') . nil) |
              lin 'w'{0}('t') . 't'{0} < True > . nil) .
eq P4 = new*[('x1' 'y1')('x2' 'y2')('x3' 'y3')
             ('x4' 'y4')('x5' 'y5')('x6' 'y6')]
       ('x2'{0} < True > . un 'x1'{0}('w') . 'w'{0} < 'x4'{0} > .
                                             'w'{0} < 'y4'{0} > . nil |
        lin 'y2'{0}('b') . lin 'y4'{0}('z') . 'x4'{0} < 'a'{0} > . nil |
        'y1'{0} < 'x3'{0} > . lin 'y3'{0}('z') . lin 'y3'{0}('t') .
             'z'{0} < 'a'{0} > . lin 't'{0}('c') . 'x5'{0} < True > .
                                               'x6'{0} < False > . nil |
          lin 'y6'{0}('a') . lin 'y5'{0}('b'). nil) .
eq P5 = new*[('x1' 'y1')('x2' 'y2')('x3' 'y3')('x4' 'y4')('x5' 'y5')]
       ('x1'{0} < 'x2'{0} > .
           ('y2'{0} >> {('a' : 'x3'{0} < True > . 'x4'{0} < False > .
                                            lin 'y4'{0} ('c') . nil)
                        ('b' : 'x3'{0} < False > . 'y5'{0} < 'x4'{0} > .
                                            lin 'y4'{0}('c')  . nil)}) |
        un 'y1'{0}('z') . ('z'{0} << 'a' . lin 'y3'{0}('c') . nil) |
        lin 'y1'{0}('w') . ('w'{0} << 'b' . lin 'y3'{0}('c') .
                               lin 'x5'{0}('t') . 't'{0} < True > . nil)) .
```

Fig. 8. Processes in Maude

We perform a similar analysis for the process P4 by using the commands:

```
search P4 =>! new* [nl:QidSet] P:Trm
                       such that wait-aux(nl:QidSet, P:Trm) .
red modelCheck(P4, build-lock-formula(('x1' 'y1')('x2' 'y2')('x3' 'y3')
                                     ('x4' 'y4')('x5' 'y5')('x6' 'y6'))) .
```

Then, we obtain:

```
search in TEST : P4 =>! new*[nl:QidSet]P:Trm
                           such that wait-aux(nl:QidSet, P:Trm) = true .
Solution 1 (state 6)
nl:QidSet --> ('x2' 'y2') ('x3' 'y3') ('x4' 'y4')('x5' 'y5')
              ('x6' 'y6') 'y1' 'x1' P:Trm -->
        'x5'{0} < True > . 'x6'{0} < False > . nil |
```

```
       lin 'y6'{0}('a') . lin 'y5'{0}( 'b') . nil |
       un 'x1'{0}('w') . 'w'{0} < 'x4'{0} > . 'w'{0} < 'y4'{0} > . nil

No more solutions.

reduce in TEST : modelCheck(P4, build-lock-formula(('x1' 'y1')
           ('x2' 'y2') ('x3' 'y3') ('x4' 'y4') ('x5' 'y5') 'x6' 'y6')) .
result ModelCheckResult: counterexample(
   {new*[('x1' 'y1') ('x2' 'y2') ('x3' 'y3') ('x4' 'y4')
                                  ('x5' 'y5') 'x6' 'y6']
     'x2'{0} < True > . un 'x1'{0}('w') . 'w'{0} < 'x4'{0} > .
                                         'w'{0} < 'y4'{0} > . nil |
     'y1'{0} < 'x3'{0} > . lin 'y3'{0}('z') . lin 'y3'{0}('t') .
     'z'{0} < 'a'{0} > . lin 't'{0}('c') .  'x5'{0} < True > .
     'x6'{0} < False > . nil | lin 'y2'{0}('b') . lin 'y4'{0}('z') .
     'x4'{0} < 'a'{0} > . nil | lin 'y6'{0}('a') . lin 'y5'{0}('b') .
     nil,'LINCOM}
   {new*[('x2' 'y2') ('x3' 'y3') ('x4' 'y4') ('x5' 'y5')
                                 ('x6' 'y6') 'y1' 'x1']
     'y1'{0} < 'x3'{0} > . lin 'y3'{0}('z') . lin 'y3'{0}('t') .
     'z'{0} < 'a'{0} > . lin 't'{0}('c') . 'x5'{0} < True > .
     'x6'{0} < False > . nil | lin 'y4'{0}('z') . 'x4'{0} < 'a'{0} > .
     nil | lin 'y6'{0}('a') . lin 'y5'{0}('b') . nil |  un 'x1'{0}('w') .
     'w'{0} < 'x4'{0} > . 'w'{0} < 'y4'{0} > . nil,'UNCOM}
   {new*[('x2' 'y2') ('x3' 'y3') ('x4' 'y4') ('x5' 'y5')
                                 ('x6' 'y6') 'y1' 'x1']
     'x3'{0} < 'x4'{0} > . 'x3'{0} < 'y4'{0} > . nil | lin 'y3'{0}('z') .
     lin 'y3'{0}('t') . 'z'{0} < 'a'{0} > . lin 't'{0}('c') .
     'x5'{0} < True > . 'x6'{0} < False > . nil | lin 'y4'{0}('z') .
     'x4'{0} < 'a'{0} > . nil | lin 'y6'{0}('a') . lin 'y5'{0}('b') .
     nil | un 'x1'{0}('w') . 'w'{0} < 'x4'{0} > . 'w'{0} < 'y4'{0} > .
     nil,'LINCOM}
   {nw*[('x2' 'y2') ('x3' 'y3') ('x4' 'y4') ('x5' 'y5')
                               ('x6' 'y6') 'y1' 'x1']
     x3'{0} < 'y4'{0} > . nil | lin 'y3'{0}('t') . 'x4'{0} < 'a'{0} > .
      lin 't'{0}('c') . 'x5'{0} < True > . 'x6'{0} < False > . nil |
      lin 'y4'{0}('z') . 'x4'{0} < 'a'{0} > . nil | lin 'y6'{0}('a') .
      lin 'y5'{0}('b') . nil | un 'x1'{0}('w') . 'w'{0} < 'x4'{0} > .
      'w'{0} < 'y4'{0} > . nil,'LINCOM}
   {new*[('x2' 'y2') ('x3' 'y3') ('x4' 'y4') ('x5' 'y5')
                                 ('x6' 'y6') 'y1' 'x1']
      'x4'{0} < 'a'{0} > . lin 'y4'{0}('c') . 'x5'{0} < True > .
      'x6'{0} < False > . nil | lin 'y4'{0}('z') . 'x4'{0} < 'a'{0} > .
      un 'x1'{0}('w') . 'w'{0} < 'x4'{0} > . 'w'{0} < 'y4'{0} > .
      nil, 'LINCOM}
   {new*[('x2' 'y2') ('x3' 'y3') ('x4' 'y4') ('x5' 'y5')
                                  ('x6' 'y6') 'y1' 'x1']
      'x4'{0} < 'a'{0} > . nil | lin 'y4'{0}('c') . 'x5'{0} < True > .
```

```
     'x6'{0} < False > . nil | lin 'y6'{0}('a') . lin 'y5'{0}('b') .
     nil | un 'x1'{0}('w') . 'w'{0} < 'x4'{0} > . 'w'{0} < 'y4'{0} > .
     nil,'LINCOM},
   {new*[('x2' 'y2') ('x3' 'y3') ('x4' 'y4') ('x5' 'y5')
                               ('x6' 'y6') 'y1' 'x1']
     'x5'{0} < True > . 'x6'{0} < False > . nil |
     lin 'y6'{0}('a') . lin 'y5'{0}('b') . nil |
     un 'x1'{0}('w') . 'w'{0} < 'x4'{0} > . 'w'{0} < 'y4'{0} > .
     nil,deadlock})
```

Process P4 performs some reductions on variables x1, y1, x2, y2, x3, y3, x4, y4 before reaching a deadlock involving variables x5, y5, x6, y6.

Finally, Consider a similar execution for the process P5:

```
search P5 =>! new* [nl:QidSet] P:Trm
                      such that wait-aux(nl:QidSet, P:Trm) .
red modelCheck(P5, build-lock-formula(('x1' 'y1')('x2' 'y2')('x3' 'y3')
                                       ('x4' 'y4')('x5' 'y5'))) .
```

We obtain the following results:

```
search in TEST : P5 =>! new*[nl:QidSet]P:Trm
                  such that wait-aux(nl:QidSet,P:Trm) = true .
Solution 1 (state 6)
nl:QidSet --> ('x1' 'y1') ('x2' 'y2') ('x3' 'y3') ('x4' 'y4') 'x5' 'y5'
P:Trm --> 'x4'{0} < False > . lin 'y4'{0}('c') . nil |
        lin 'y1'{0}('w') . ('w'{0} << 'b' . lin 'y3'{0}('c') .
                          lin 'x5'{0}('t') . 't'{0} < True > . nil) |
        un 'y1'{0}('z') . ('z'{0} << 'a' . lin 'y3'{0}('c') . nil)

No more solutions.

reduce in TEST : modelCheck(P5, build-lock-formula(('x1' 'y1')
                      ('x2' 'y2') ('x3' 'y3') ('x4' 'y4') 'x5' 'y5')) .
result ModelCheckResult: counterexample(
   {new*[('x1' 'y1') ('x2' 'y2') ('x3' 'y3') ('x4' 'y4') 'x5' 'y5']
     'x1'{0} < 'x2'{0} > . ('y2'{0} >>{('a' : 'x3'{0} < True > .
        'x4'{0} < False > . lin 'y4'{0}('c') . nil)
         'b' : 'x3'{0} < False > .  'y5'{0} < 'x4'{0} > .
                                       lin 'y4'{0}('c') . nil}) |
     lin 'y1'{0}('w') . ('w'{0} << 'b' . lin 'y3'{0}('c') .
                         lin 'x5'{0}('t') . 't'{0} < True > . nil) |
     un 'y1'{0}('z') . ('z'{0} << 'a' . lin 'y3'{0}('c') . nil),'UNCOM}
   {new*[('x1' 'y1') ('x2' 'y2') ('x3' 'y3') ('x4' 'y4') 'x5' 'y5']
     ('y2'{0} >>{('a' : 'x3'{0} < True > . 'x4'{0} < False > .
                 lin 'y4'{0}('c') . nil) 'b' : 'x3'{0} < False > .
                 'y5'{0} < 'x4'{0} > . lin 'y4'{0}('c') . nil}) |
     lin 'y1'{0}('w') . ('w'{0} << 'b' . lin 'y3'{0}('c') .
                        lin 'x5'{0}('t') . 't'{0} < True > . nil) |
     un 'y1'{0}('z') . ('z'{0} << 'a' . lin 'y3'{0}('c') . nil) |
```

```
    ('x2'{0} << 'a' . lin 'y3'{0}('c') . nil),'CASE}
 {new*[('x1' 'y1') ('x2' 'y2') ('x3' 'y3') ('x4' 'y4') 'x5' 'y5']
    'x3'{0} < True > . 'x4'{0} < False > . lin 'y4'{0}('c') . nil |
    lin 'y1'{0}('w') . ('w'{0} << 'b' . lin 'y3'{0}('c') .
       lin 'x5'{0}('t') . 't'{0} < True > . nil) |
    lin 'y3'{0}('c') . nil |
    un 'y1'{0}('z').('z'{0} << 'a' . lin 'y3'{0}('c') . nil), 'LINCOM},
 {new*[('x1' 'y1') ('x2' 'y2') ('x3' 'y3') ('x4' 'y4') 'x5' 'y5']
    'x4'{0} < False > . lin 'y4'{0}('c') . nil |
    lin 'y1'{0}('w') . ('w'{0} << 'b' . lin 'y3'{0}('c') .
                       lin 'x5'{0}('t') . 't'{0} < True > . nil) |
    un 'y1'{0}('z').('z'{0} << 'a' . lin 'y3'{0}('c') . nil),deadlock})
```

These results are consistent being as process P5 is not deadlock free and not lock free.