

An Executable Specification of Asynchronous Pi-Calculus Semantics and May Testing in Maude 2.0

Prasanna Thati Koushik Sen
Department of Computer Science
University of Illinois at Urbana-Champaign
{thati,ksen}@cs.uiuc.edu

Narciso Martí-Oliet
Dpto. de Sistemas Informáticos y Programación
Universidad Complutense de Madrid, Spain
narciso@sip.ucm.es

Abstract

We describe an executable specification of the operational semantics of an asynchronous version of the π -calculus in Maude by means of conditional rewrite rules with rewrites in the conditions. We also present an executable specification of the may testing equivalence on non-recursive asynchronous π -calculus processes, using the Maude meta-level. Specifically, we describe our use of the `metaSearch` operation to both calculate the set of all finite traces of a non-recursive process, and to compare the trace sets of two processes according to a preorder relation that characterizes may testing in asynchronous π -calculus. Thus, in both the specification of the operational semantics and the may testing, we make heavy use of new features introduced in version 2.0 of the Maude language and system.

Key words: π -calculus, asynchrony, may testing, traces, Maude.

1 Introduction

Since its introduction in the seminal paper [11] by Milner, Parrow, and Walker, the π -calculus has become one of the most studied calculus for name-based mobility of processes, where processes are able to exchange names over channels so that the communication topology can change during the computation. The operational semantics of the π -calculus has been defined for several different versions of the calculus following two main styles. The first is the labelled transition system style according to the SOS approach introduced by Plotkin [12]. The second is the reduction style, where first an equivalence is imposed on syntactic processes (typically to make syntax more abstract with respect to properties of associativity and/or commutativity of some operators), and then some reduction or rewrite rules express how the computation proceeds by communication between processes.

The first specification of the π -calculus operational semantics in rewriting logic was developed by Viry in [19], in a reduction style making use of de Bruijn indexes, explicit substitutions, and reduction strategies in Elan [5]. This presentation was later improved by Stehr [14] by making use of a generic calculus for explicit substitutions, known as *CINNI*, which

combines the best of the approaches based on standard variables and de Bruijn indices, and that has been implemented in Maude.

Our work took the work described above as a starting point, together with recent work by Verdejo and Martí-Oliet [18] showing how to use the new features of Maude 2.0 in the implementation of a semantics in the labelled transition system style for CCS. This work makes essential use of conditional rewrite rules with rewrites in the conditions, so that an inference rule in the labelled transition system of the form

$$\frac{P_1 \rightarrow Q_1 \quad \dots \quad P_n \rightarrow Q_n}{P_0 \rightarrow Q_0}$$

becomes a *conditional* rewrite rule of the form

$$P_0 \longrightarrow Q_0 \quad \text{if} \quad P_1 \longrightarrow Q_1 \wedge \dots \wedge P_n \longrightarrow Q_n,$$

where the condition includes rewrites. These rules are executable in version 2.0 of the Maude language and system [6]. However, this is not enough, because it is necessary to have some control on the application of rules. Typically, rewrite rules can be applied anywhere in a term, while the transitions in the operational semantics for CCS or the π -calculus in the SOS style only take place at the top. The new `frozen` attribute available in Maude 2.0 makes this possible, because the declaration of an operator as frozen forbids rewriting its arguments, thus providing another way of controlling the rewriting process. Rewrite conditions when applying conditional rules are solved by means of an implicit search process, which is also available to the user both at the command level and at the metalevel as an operation `metaSearch`.¹

In this way, our first contribution is a fully executable specification of an operational semantics in the labelled transition system style for an asynchronous version of the π -calculus (the semantics for the synchronous case is obtained as a simple modification). This specification uses conditional rewrite rules with rewrites in conditions and the CINNI calculus [14] for managing names and bindings in the π -calculus. However, these two ingredients are not enough to obtain a fully executable specification. A central problem to overcome is that the transitions of a term can be *infinitely branching*. For instance, the term $x(y).P$ can evolve via an input action to one of an infinite family of terms depending on the name received in the input at channel x . Our solution is to define the transitions of a process relative to an execution environment. The environment is represented abstractly as a set of free (global) names that the environment may use while interacting with the process, and transitions are modelled as rewrite rules over a pair consisting of a set of environment names together with a process.

Our next contribution is to implement the verification of the *may-testing preorder* [13] between finitary (non-recursive) asynchronous π -calculus processes, using again ideas from [18] to calculate the set of all finite traces of a process. May-testing is a specific instance of the notion of behavioral equivalence on π -calculus processes. In may testing, two processes are said to be equivalent if they have the same success properties in all experiments. An experiment consists of an observing process that runs in parallel and interacts with the process being tested, and success is defined as the observer signalling a special event. Viewing the occurrence of an event as something bad happening, may testing can be used to reason about safety properties [3].

¹We do not give more details about these features in this paper, since the interested reader can find a detailed explanation about them and their use in the implementation of operational semantics in the paper [18].

Since the definition of may testing involves a universal quantification over all observers, it is difficult to establish process equivalences directly from the definition. As a solution, alternate characterizations of the equivalence that do not resort to quantification over observers have been found. It is known that the trace semantics is an alternate characterization of may testing in (synchronous) π -calculus [8], while a variant of the trace semantics has been shown to characterize may testing in an asynchronous setting [4]. Specifically, in both these cases, comparing two processes according to the may-testing preorder amounts to comparing the set of all finite traces they exhibit. We have implemented for finite asynchronous processes, the comparison of trace sets proposed in [4]. We stress that our choice of specifying an asynchronous version rather than the synchronous π -calculus, is because the characterization of may testing for the asynchronous case is more interesting and difficult. The synchronous version can be specified in an executable way using similar but simpler techniques.

Our first step in obtaining an executable specification of may-testing, is to obtain the set of all finite traces of a given process. This is done at the Maude metalevel by using the `metaSearch` operation to collect all results of rewriting a given term. The second step is to specify a preorder relation between traces that characterizes may-testing. We have represented the trace preorder relation as a rewriting relation, i.e. the rules of inference that define the trace preorder are again modeled as conditional rewrite rules. The final step is to check if two processes are related by the may preorder, i.e. whether a statement of the form $P \sqsubseteq Q$ is true or not. This step involves computing the closure of a trace under the trace-preorder relation, again by the using `metaSearch` operation. Thus, our work demonstrates the utility of the new metalevel facilities available in Maude 2.0.

The structure of the paper follows the steps in the description above. Section 2 describes the syntax of the asynchronous version of the π -calculus that we consider, together with the corresponding CINNI operations we use. Section 3, describes the operational semantics specified by means of conditional rewrite rules. Sections 4 and 5 define traces and the preorder on traces, respectively. Finally, Section 6 contains the specification of the may testing on processes as described above. Section 7 concludes the paper along with a brief discussion of future work.

2 Asynchronous π -Calculus Syntax

Following is a brief and informal review of a version of asynchronous π -calculus that is equipped with a conditional construct for matching names. An infinite set of channel names is assumed, and u, v, w, x, y, z, \dots are assumed to range over it. The set of processes, ranged over by P, Q, R , is defined by the following grammar.

$$P := \bar{x}y \mid \sum_{i \in I} \alpha_i.P_i \mid P_1|P_2 \mid (\nu x)P \mid [x = y](P_1, P_2) \mid !P$$

where α can be $x(y)$ or τ . The output term $\bar{x}y$ denotes an asynchronous message with target x and content y . The summation $\sum_{i \in I} \alpha_i.P_i$ non-deterministically chooses an α_i , and if $\alpha_i = \tau$ it evolves internally to P_i , and if $\alpha_i = x(y)$ it receives an arbitrary name z at channel x and then behaves like $P\{z/y\}$. The process $P\{z/y\}$ is the result of the substitution of free occurrences of y in P by z , with the usual renaming of bound names to avoid accidental captures (thus substitution is defined only modulo α -equivalence). The argument y in $x(y).P$ binds all free occurrences of y in P . The composition $P_1|P_2$ consists of P_1 and P_2 acting in parallel. The components can act independently, and also interact with each other. The

[a := x]	[shiftup a]	[shiftdown a]	[lift a S]
a{0} ↦ x	a{0} ↦ a{1}	a{0} ↦ a{0}	a{0} ↦ [shiftup a] (S a{0})
a{1} ↦ a{0}	a{1} ↦ a{2}	a{1} ↦ a{0}	a{1} ↦ [shiftup a] (S a{1})
...
a{n+1} ↦ a{n}	a{n} ↦ a{n+1}	a{n+1} ↦ a{n}	a{n} ↦ [shiftup a] (S a{n})
b{m} ↦ b{m}	b{m} ↦ b{m}	b{m} ↦ b{m}	b{m} ↦ [shiftup a] (S b{m})

Table 1: The CINNI operations.

restriction $(\nu x)P$ behaves like P except that it can not exchange messages targeted to x , with its environment. The restriction binds free occurrences of x in P . The conditional $[x = y](P_1, P_2)$ behaves like P_1 if x and y are identical, and like P_2 otherwise. The replication $!P$ provides an infinite number of copies of P . The functions for free names $fn(\cdot)$, bound names $bn(\cdot)$ and names $n(\cdot)$, of a process, are defined as expected.

The Maude specification for the π -calculus syntax is given below. The sort **Chan** is used to represent channel names.

```

sort Chan .
sorts Guard GuardedTrm SumTrm Trm .
subsort GuardedTrm < SumTrm .
subsort SumTrm < Trm .

op _(_) : Chan Qid -> Guard .
op tau : -> Guard .
op nil : -> Trm .
op _<_> : Chan Chan -> Trm [frozen] .
op _.._ : Guard Trm -> GuardedTrm [frozen] .
op _+_ : SumTrm SumTrm -> SumTrm [frozen assoc comm] .
op _|_ : Trm Trm -> Trm [frozen assoc comm] .
op new[_]_ : Qid Trm -> Trm [frozen] .
op if=_then_else_fi : Chan Chan Trm Trm -> Trm [frozen] .
op !_ : Trm -> Trm [frozen] .

```

Note that the syntactic form $\sum_{i \in I} \alpha_i.P_i$ has been split into three cases:

1. **nil** represents the case where $I = \emptyset$,
2. a term of sort **GuardedTrm** represents the case where $I = \{1\}$, and
3. a term of sort **SumTrm** represents the case where $I = [1..n]$ for $n > 1$. Since the constructor **_+_** is associative and the sort **GuardedTrm** is a subsort of **SumTrm**, we can represent a finite sum $\sum_{i \in I} \alpha_i.P_i$ as $(\dots (\alpha_1.P_1 + \alpha_2.P_2) + \dots \alpha_n.P_n)$.

To represent substitution on π -calculus processes (and traces, see Section 4) at the language level we use CINNI as a calculus for explicit substitutions [14]. This gives a first-order representation of terms with bindings and capture-free substitutions, instead of going to the metalevel to handle names and bindings. The main idea in such a representation is to keep the bound names inside the binders as it is, but to replace its use by the name followed by an index which is a count of the number of binders with the same name it jumps before it reaches the place of use. Following this idea, we define terms of sort **Chan** as indexed names as follows.

$INP: \sum_{i \in I} \alpha_i . P_i \xrightarrow{x_j z} P_j \{z/y\} \quad j \in I, \alpha_j = x_j(y)$	$OUT: \bar{x}y \xrightarrow{\bar{x}y} 0$
$TAU: \sum_{i \in I} \alpha_i . P_i \xrightarrow{\tau} P_j \quad j \in I, \alpha_j = \tau$	$BINP: \frac{P \xrightarrow{xy} P'}{P \xrightarrow{x(y)} P'} \quad y \notin fn(P)$
$PAR: \frac{P_1 \xrightarrow{\alpha} P'_1}{P_1 P_2 \xrightarrow{\alpha} P'_1 P_2} \quad bn(\alpha) \cap fn(P_2) = \emptyset$	$COM: \frac{P_1 \xrightarrow{\bar{x}y} P'_1 \quad P_2 \xrightarrow{xy} P'_2}{P_1 P_2 \xrightarrow{\tau} P'_1 P'_2}$
$RES: \frac{P \xrightarrow{\alpha} P'}{(\nu y)P \xrightarrow{\alpha} (\nu y)P'} \quad y \notin n(\alpha)$	$OPEN: \frac{P \xrightarrow{\bar{x}y} P'}{(\nu y)P \xrightarrow{\bar{x}(y)} P'} \quad x \neq y$
$CLOSE: \frac{P_1 \xrightarrow{\bar{x}(y)} P'_1 \quad P_2 \xrightarrow{xy} P'_2}{P_1 P_2 \xrightarrow{\tau} (\nu y)(P'_1 P'_2)} \quad y \notin fn(P_2)$	$REP: \frac{P !P \xrightarrow{\alpha} P'}{!P \xrightarrow{\alpha} P'}$
$IF: \frac{P \xrightarrow{\alpha} P'}{[x = x](P, Q) \xrightarrow{\alpha} P'}$	$ELSE: \frac{Q \xrightarrow{\alpha} Q'}{[x = y](P, Q) \xrightarrow{\alpha} Q'} \quad x \neq y$

Table 2: A labelled transition system for asynchronous π -calculus.

```
sort Chan .
op _'{'_}' : Qid Nat -> Chan [prec 1] .
```

We introduce a sort of substitutions **Subst** together with the following operations:

```
op '['_':=' ] : Qid Chan -> Subst .
op '['shiftp_' ] : Qid -> Subst .
op '['shiftdown_' ] : Qid -> Subst .
op '['lift_-' ] : Qid Subst -> Subst .
```

The first two substitutions are basic substitutions representing *simple* and *shiftp* substitutions; the third substitution is a special case of *simple* substitution; the last one represents complex substitution where a substitution can be lifted using the operator **lift**. The intuitive meaning of these operations is described in Table 1. Using these, explicit substitutions for π -calculus processes are defined equationally. Following are a few interesting equations:

```
eq S (P + Q) = (S P) + (S Q) .
eq S (CX(Y) . P) = (S CX)(Y) . ([lift Y S] P) .
eq S (new [X] P) = new [X] ([lift X S] P) .
```

3 Operational Semantics

A labelled transition system (see Table 2) is used to give an operational semantics for the calculus as in [4]. The transition system is defined modulo α -equivalence on processes in that α -equivalent processes have the same transitions. The rules *COM*, *CLOSE*, and *PAR* have symmetric versions that are not shown.

Transition labels, which are also called actions, can be of five forms: τ (a silent action), $\bar{x}y$ (free output of a message with target x and content y), $\bar{x}(y)$ (bound output), xy (free input of a message), and $x(y)$ (bound input). The functions $fn(\cdot)$, $bn(\cdot)$ and $n(\cdot)$ are defined on actions as expected. The set of all visible (non- τ) actions is denoted by \mathcal{L} , and α is assumed

to range over \mathcal{L} . As a uniform notation for free and bound actions the following notational convention is adopted: $(\emptyset)\bar{x}y = \bar{x}y$, $(\{y\})\bar{x}y = \bar{x}(y)$, and similarly for input actions. The variable \hat{z} is assumed to range over $\{\emptyset, \{z\}\}$. The term $(\nu\hat{z})P$ is $(\nu z)P$ if $\hat{z} = \{z\}$, and P otherwise.

We define the sort `Action` and the corresponding operations as follows:

```

sorts Action ActionType .
ops i o : -> ActionType .
op f : ActionType Chan Chan -> Action .
op b : ActionType Chan Qid -> Action .
op tauAct : -> Action .

```

The operators `f` and `b` are used to construct free and bound actions respectively. Name substitution on actions is defined equationally in the obvious way.

The inference rules in Table 2 are modelled as conditional rewrite rules with the premises as conditions of the rule.² Since rewrites do not have labels unlike the labelled transitions, we make the label a part of the resulting term; thus rewrites corresponding to transitions in the operational semantics are of the form $P \Rightarrow \{\alpha\}Q$.

Because of the *INP* and *OPEN* rules, the transitions of a term can be infinitely branching. Specifically, in case of the *INP* rule there is one branch for every possible name that can be received in the input. In case of the *OPEN* rule, there is one branch for every name that is chosen to denote the private channel that is being emitted (note that the transition rules are defined only modulo α -equivalence). To overcome this problem, we define transitions over pairs of the form $[[\mathbf{CS}] P]$, where \mathbf{CS} is a set of channel names containing all the names that the environment with which the process interacts, knows about. The set \mathbf{CS} expands during bound input and output interactions when private names are exchanged between the process and its environment.

The infinite branching due to the *INP* rule is avoided by allowing only the names in the environment set \mathbf{CS} to be received in free inputs. Since \mathbf{CS} is assumed to contain all the free names in the environment, an input argument that is not in \mathbf{CS} would be a private name of the environment. Now, since the identifier chosen to denote the fresh name is irrelevant, all bound input transitions can be identified to a single input. With these simplifications, the number of input transitions of a term become finite. Similarly, in *OPEN* rule, since the identifier chosen to denote the private name emitted is irrelevant, instances of the rule that differ only in the chosen name are not distinguished.

We discuss in detail the implementation of only a few of the inference rules; the reader is referred to the appendix for a complete list of all the rewrite rules for Table 2.

```

sorts EnvTrm TraceTrm .
subsort EnvTrm < TraceTrm .
op [_]_ : Chanset Trm -> EnvTrm [frozen] .
op {[_]}_ : Action TraceTrm -> TraceTrm [frozen] .

```

The following rule is for free inputs.

$$\text{r1 [Inp]} : [\mathbf{CY} \ \mathbf{CS}] ((\mathbf{CX}(X) . P) + \text{SUM}) \Rightarrow \{f(i, \mathbf{CX}, \mathbf{CY})\} ([\mathbf{CY} \ \mathbf{CS}] ([X := \mathbf{CY}] P)) .$$

²The symmetric versions missing in the table need not be implemented because the process constructors `+_` and `_|_` have been declared as commutative.

The next rule we consider is the one for bound inputs. Since the identifier chosen to denote the bound argument is irrelevant, we use the constant 'U for all bound inputs, and thus 'U{0} denotes the fresh channel received. Note that in contrast to the *BINP* rule of Table 2, we do not check if 'U{0} is in the free names of the process performing the input, and instead we shift up the channel indices appropriately, in both the set of environment names *CS* and the process *P* in the righthand side and condition of the rule. This is justified because the transition target is within the scope of the bound name in the input action. Note also that the channel *CX* in the action is not shifted down because it is out of the scope of the bound argument. Finally, the set of environment names is expanded by adding the received channel 'U{0} to it.

```

crl [BInp] : [CS] P => {b(i,CX,'U)} ['U{0} [shiftup 'U] CS] P1
             if ['U{0} [shiftup 'U] CS] [shiftup 'U] P =>
               {f(i,CX,'U{0})} ['U{0} [shiftup 'U] CS] P1 .

```

The following rule treats the case of bound outputs.

```

crl [Open] : [CS] (new [X] P) => {[shiftdown X] b(o,CY,X)} [X{0} [shiftup X] CS] P1
             if [[shiftup X] CS] P => {f(o,CY,X{0})} [[shiftup X] CS] P1 /\ X{0} /= CY .

```

Like in the case of bound inputs, we identify all bound outputs to a single instance in which the identifier *X* that appears in the restriction is chosen as the bound argument name. Note that in both the righthand side of the rule and in the condition, the indices of the channels in *CS* are shifted up, because they are effectively moved across the restriction. Similarly, the channel indices in the action in the righthand side of the rule are shifted down since the action is now moved out of the restriction. Note also that the exported name is added to the set of environment names, because the environment that receives this exported name can use it in subsequent interactions.

The *PAR* inference rule is implemented by two rewrite rules, one for the case where the performed action is free, and the other where the action is bound. Following is the rewrite rule for the latter. The rewrite rule for the former case is simpler and appears in the appendix.

```

var IO : ActionType
crl [Par] : [CS] (P | Q) => {b(IO,CX,Y)} [Y{0} ([shiftup Y] CS)] (P1 | [shiftup Y] Q1)
             if [CS] P => {b(IO,CX,Y)} ([Y{0} ([shiftup Y] CS)] P1).

```

Note that the side condition of the *PAR* rule in Table 2, which avoids confusion of the emitted bound name with free names in *Q*, is achieved by shifting up channel indices in *Q*. This is justified because the righthand side of the rule is under the scope of the bound output action. Similarly, the channel indices in the environment are also shifted up. Further, the set of environment names is expanded by adding the exported channel *Y{0}*.

Finally, we consider the rewrite rule for *CLOSE*. The process *P* emits a bound name *Y*, which is received by process *Q*. Since the scope of *Y* after the transition includes *Q*, the rewrite involving *Q* in the second condition of the rule is carried out within the scope of the bound name that is emitted. This is achieved by adding the channel *Y{0}* to the set of environment names and shifting up the channel indices in both *CS* and *Q* in the rewrite. Note that since the private name being exchanged is not emitted to the environment, we neither expand the set *CS* in the right hand side of the rule, nor shift up the channel indices in it.

```

crl [Close] : [CS] (P | Q) => {tauAct} [CS] new [Y] (P1 | Q1)
             if [CS] P => {b(o,CX,Y)} [Y{0} [shiftup Y] CS] P1 /\
               [Y{0} [shiftup Y] CS] [shiftup Y] Q => {f(i,CX,Y{0})} [Y{0} [shiftup Y] CS] Q1 .

```

We conclude this section with the following note. The operator $\{-\}_-$ is declared **frozen** because further rewrites of the process term encapsulated in a term of sort `TraceTrm` are useless. This is because all the conditions of the transition rules only involve one step rewrites (the righthand side of these rewrites can only match a term of sort `TraceTrm` with a single action prefix). Further note that, to prevent rewrites of a term to a non well-formed term all the constructors for π -calculus terms (Section 2) have been declared **frozen**. For instance in the absence of this declaration we would have rewrites of the form $P \mid Q \Rightarrow \{A\}.P1 \mid Q$ to a non well-formed term.

4 Trace Semantics

The set \mathcal{L}^* is the set of traces. The functions $fn(\cdot)$, $bn(\cdot)$ and $n(\cdot)$ are extended to \mathcal{L}^* in the obvious way. The α -equivalence on traces is defined as expected, and α -equivalent traces are not distinguished. The relation \Longrightarrow denotes the reflexive transitive closure of $\xrightarrow{\tau}$, and $\xRightarrow{\beta}$ denotes $\Longrightarrow \xrightarrow{\beta} \Longrightarrow$. For $s = l.s'$, we compactly write $P \xRightarrow{l} \xRightarrow{s'} P'$ as $P \xRightarrow{s} P'$. We use $P \xRightarrow{s}$ as an abbreviation for, $P \xRightarrow{s} P'$ for some P' . The set of traces that a process exhibits is then $\llbracket P \rrbracket = \{s \mid P \xRightarrow{s}\}$.

In the implementation, we introduce a sort `Trace` as supersort of `Action` to specify traces.

```
subsort Action < Trace .
op epsilon : -> Trace .
op _._ : Trace Trace -> Trace [assoc id: epsilon] .
```

We define the operator $\text{' }[_\text{'}$ to represent a complete trace. The motivation for doing so is to restrict the equations and rewrite rules defined over traces to operate only on a complete trace instead of a part of it. The following equation defines α -equivalence on traces.

```
op '[_]' : Trace -> TTrace .
ceq [TR1 . b(IO,CX,Y) . TR2] = [TR1 . b(IO,CX,'U) . [Y := 'U{0}] [shiftup 'U] TR2]
    if Y /= 'U .
```

Note that, in a trace $\text{TR1}.b(\text{IO},\text{CX},\text{Y}).\text{TR2}$ the action $b(\text{IO},\text{CX},\text{Y})$ binds the identifier Y in TR2 .

Because the operator $\text{op } \{-\}_-$: `Action TraceTrm -> TraceTrm` is declared as **frozen**, a term of sort `EnvTrm` can rewrite only once, and so we cannot obtain the set of finite traces of a process by simply rewriting it multiple times in all possible ways. The problem is solved as in [18], by specifying the trace semantics using rules that generate the transitive closure of one step transitions as follows:

```
sort TTrm .
op [_] : EnvTrm -> TTrm [frozen] .
var TT : TraceTrm .

crl [reflx] : [ P ] => {A} Q if P => {A} Q .
crl [trans] : [ P ] => {A} TT if P => {A} Q /\ [ Q ] => TT /\ [ Q ] /= TT .
```

We use the operator $[_]$ to prevent infinite loops while evaluating the conditions of the rules above. If this operator is not used then the lefthand side of the rewrite in the condition would match the lefthand side of the rule itself, and so the rule itself could be used in order

<i>(Drop)</i>	$s_1.(\hat{y})s_2$	\prec	$s_1.(\hat{y})xy.s_2$	if $(\hat{y})s_2 \neq \perp$
<i>(Delay)</i>	$s_1.(\hat{y})(\alpha.xy.s_2)$	\prec	$s_1.(\hat{y})xy.\alpha.s_2$	if $(\hat{y})(\alpha.xy.s_2) \neq \perp$
<i>(Annihilate)</i>	$s_1.(\hat{y})s_2$	\prec	$s_1.(\hat{y})xy.\bar{x}y.s_2$	if $(\hat{y})s_2 \neq \perp$

Table 3: A preorder relation on traces.

to solve its condition. This operator is also declared as **frozen** to prevent useless rewrites inside `[_]`.

We can now use the **search** command of Maude 2.0 to find all possible traces of a process. The traces appear as prefix of the one-step successor of a process. Thus, we have obtained an executable specification of the trace semantics of asynchronous π -calculus.

5 A Trace Based Characterization of May Testing

The may testing framework [13] is instantiated on asynchronous π -calculus as follows. Observers are processes that can emit a special message $\bar{\mu}\mu$. We say that an observer O accepts a trace s if $O \xrightarrow{\bar{s}\bar{\mu}\mu}$, where \bar{s} is the trace obtained by complementing the actions in s , i.e. converting input actions to output actions and vice versa. The may preorder \sqsubseteq over processes is defined as: $P \sqsubseteq Q$ if for every observer O , $P|O \xrightarrow{\bar{\mu}\mu}$ implies $Q|O \xrightarrow{\bar{\mu}\mu}$. We say P and Q are may-equivalent, i.e. $P = Q$ if $P \sqsubseteq Q$ and $Q \sqsubseteq P$. The universal quantification on contexts in this definition makes it very hard to prove equalities directly from the definition, and makes mechanical checking impossible. To circumvent this problem, a trace based alternate characterization of the equivalence is proposed in [4]. We now summarize this characterization and discuss our implementation of it.

The preorder \preceq on traces is defined as the reflexive transitive closure of the laws shown in Table 3, where the notation $(\hat{y})\cdot$ is extended to traces as follows.

$$(\hat{y})s = \begin{cases} s & \text{if } \hat{y} = \emptyset \text{ or } b \notin fn(s) \\ s_1.x(y).s_2 & \text{if } \hat{y} = \{y\} \text{ and there are } s_1, s_2, x \text{ s.t.} \\ & s = s_1.xy.s_2 \text{ and } y \notin n(s_1) \cup \{x\} \\ \perp & \text{otherwise} \end{cases}$$

For sets of traces R and S , we define $R \lesssim S$, if for every $s \in S$ there is an $r \in R$ such that $r \preceq s$. The may preorder is then characterized in [4] as: $P \sqsubseteq Q$ if and only if $[Q] \lesssim [P]$.

The main intuition behind the preorder \preceq is that if an observer accepts a trace s , then it also accepts any trace $r \preceq s$. The first two laws state that an observer cannot force inputs on the process being tested. Since outputs are asynchronous, the actions following an output in a trace exhibited by the observer need not causally depend on the output. Hence the observer's output can be delayed until a causally dependent action, or dropped if there are no such actions. The annihilation law states that an observer can consume its own outputs unless there are subsequent actions that depend on the output. The reader is referred to [4] for further details on this characterization.

We encode the trace preorder as rewrite rules on terms of the sort `TTrace`; specifically, the relation $r \prec s$ if *cond*, is encoded as `s => r if cond`. The reason for this form of representation will be justified in Section 6. The function $(\{y\})\cdot$ on traces is defined equationally by the operation `bind`. The constant `bot` of sort `Trace` is used by the `bind` operation to signal error.

```

op bind : Qid Trace -> Trace .
op bot  :   -> Trace .

var TR  : Trace .
var IO  : ActionType.

ceq TR . bot = bot  if t /= epsilon .
ceq bot . TR = bot  if t /= epsilon .

eq bind(X , epsilon) = epsilon .

eq bind(X , f(i,CX,CY) . TR ) = if CX /= X{0} then
    if CY == X{0} then ([shiftdown X] b(i, CX , X)) . TR
    else ([shiftdown X] f(i, CX , CY)) . bind(X , TR) fi
    else bot fi .

eq bind(X , b(IO,CX,Y) . TR) = if CX /= X{0} then
    if X /= Y then ([shiftdown X] b(i, CX , Y)) . bind(X , TR)
    else ([shiftdown X] b(IO, CX , Y)) . bind(X , swap(X,TR)) fi
    else bot fi .

```

The equation for the case where the second argument to `bind` begins with a free output is not shown as it is similar. Note that the channel indices in actions until the first occurrence of `X{0}` as the argument of a free input are shifted down as these move out of the scope of the binder `X`. Further, when a bound action with `X` as the bound argument is encountered, the `swap` operation is applied to the remaining suffix of the trace. The swap operation simply changes the channel indices in the suffix so that the binding relation is unchanged even as the binder `X` is moved across the bound action. This is accomplished by simultaneously substituting `X{0}` with `X{1}`, and `X{1}` with `X{0}`. Finally, note that when `X{0}` is encountered as the argument of a free input, the input is converted to a bound input. If `X{0}` is first encountered at any other place, an error is signalled by returning the constant `bot`.

The encoding of the preorder relation on traces is now straightforward.

```

crl [Drop] : [ TR1 . b(i,CX,Y) . TR2 ] => [ TR1 . bind(Y , TR2) ]
           if bind(Y , TR2) /= bot .

r1 [Delay] : [ ( TR1 . f(i,CX,CY) . b(IO,CU,V) . TR2 ) ] =>
           [ ( TR1 . b(IO,CU,V) . ([shiftup V] f(i, CX , CY)) . TR2 ) ] .

crl [Delay] : [ ( TR1 . b(i,CX,Y) . f(IO,CU,CV) . TR2 ) ] =>
           [ ( TR1 . bind(Y , f(IO,CU,CV) . f(i,CX,Y{0})) . TR2 ) ]
           if bind(Y , f(IO,CU,CV) . f(i,CX,Y{0})) . TR2 /= bot .

crl [Annihilate] : [ ( TR1 . b(i,CX,Y) . f(o,CX,Y{0}) . TR2 ) ] =>
           [ TR1 . bind(Y , TR2) ]
           if bind(Y , TR2) /= bot .

```

Note that in the first `Delay` rule, the channel indices of the free input action are shifted up when it is delayed across a bound action, since it gets into the scope of the bound argument. Similarly, in the second `Delay` rule, when the bound input action is delayed across a free input/output action, the channel indices of the free action are shifted down by the `bind` operation. The other two subcases of the `Delay` rule, namely, where a free input is to be

delayed across a free input or output, and where a bound input is to be delayed across a bound input or output, are not shown as they are similar. Similarly, for `Annihilate`, the case where a free input is to be annihilated with a free output is not shown.

6 Verifying May Preorder between Finite Processes

We now describe our implementation of verification of may preorder between finite processes, i.e. processes without replication, by exploiting the trace-based characterization of may testing discussed in Section 5. The finiteness of a process P only implies that the length of traces in $[P]$ is bounded. But the number of traces in $[P]$ can be infinite (even modulo α -equivalence) because the *INP* rule is infinitely branching. To avoid the problem of having to compare infinite sets, we observe that

$$[Q] \preceq [P] \quad \text{if and only if} \quad [Q]_{fn(P,Q)} \preceq [P]_{fn(P,Q)},$$

where for a set of traces S and a set of names ρ we define $S_\rho = \{s \in S \mid fn(s) \subseteq \rho\}$. Now, since the traces in $[P]$ and $[Q]$ are finite in length, it follows that $[P]_{fn(P,Q)}, [Q]_{fn(P,Q)}$ are finite modulo α -equivalence. In fact, the set of traces generated for $[[fn(P,Q)] P]$ by our implementation described in Section 3, contains exactly one representative from each α -equivalence class of $[P]_{fn(P,Q)}$.

Given processes P and Q , we generate the set of all traces (modulo α -equivalence) of $[[fn(P,Q)] P]$ and $[[fn(P,Q)] Q]$ using the metalevel facilities of Maude 2.0. As mentioned in Section 4, these terms, which are of sort `TTrm`, can be rewritten only once. The term of sort `TraceTrm` obtained by rewriting contains a finite trace as a prefix. To create the set of all traces, we compute all possible one-step rewrites. This computation is done by the function `TTrmtoNormalTraceSet` at the metalevel using the operation `metaSearch`. This function uses two auxiliary functions `TTrmtoTraceSet` and `TraceSettoNormalTraceSet`.

```
op TTrmtoTraceSet : Term -> TermSet .
op TraceSettoNormalTraceSet : TermSet -> TermSet .
op TTrmtoNormalTraceSet : Term -> TermSet .

eq TTrmtoNormalTraceSet(T) = TraceSettoNormalTraceSet(TTrmtoTraceSet(T)) .
```

The function `TTrmTraceSet` uses the function `allOneStepAux(T,N)` that returns the set of all one-step rewrites (according to the rules given in Sections 3 and 4, which are defined in modules named `APISEMANTICS` and `APITRACE`) of the term T which is the meta-representation of a term of sort `TTrm`, skipping the first N solutions. In the following equations, the operator `u` stands for set union.

```
op APITRACE-MOD : -> Module .
eq APITRACE-MOD = ['APITRACE] .

var N : MachineInt .
vars T X : Term .

op allOneStepAux : Term MachineInt Term -> TermSet .
eq TTrmtoTraceSet(T) = allOneStepAux(T,0,'X:TraceTrm) .
eq allOneStepAux(T,N,X) =
  if metaSearch(APITRACE-MOD,T,X,nil,'+',1,N) == failure then 'epsilon.Trace
  else TraceTermToTrace(getTerm(metaSearch(APITRACE-MOD,T,X,nil,'+',1,N))) u
    allOneStepAux(T,N + 1,X) fi .
```

The function `TraceTermToTrace`, used in `allOneStepAux`, extracts the trace out of a meta-representation of a term of sort `TraceTerm`. The function `TraceSettoNormalTraceSet` uses `metaReduce` to convert each trace in a trace set to its α -normal form.

```

op TraceTermToTrace : Term -> Term .

eq TraceSettoNormalTraceSet(mt) = mt .
eq TraceSettoNormalTraceSet(T u TS) =
  getTerm(metaReduce(TRACE-MOD, '['[_'] [ T ])) u TraceSettoNormalTraceSet(TS) .

```

We implement the relation \lesssim on sets defined in Section 5 as the predicate `<<`. We check if $P \sqsubseteq Q$ by computing this predicate on the meta-represented trace sets $[P]_{fn(P,Q)}$ and $[Q]_{fn(P,Q)}$ as follows. For each (meta-represented) trace `T` in $[P]_{fn(P,Q)}$, we compute the reflexive transitive closure of `T` with respect to the laws shown in Table 3. The laws are implemented as rewrite rules in the module `TRACE-PREORDER`. We then use the fact that $[Q]_{fn(P,Q)} \lesssim [P]_{fn(P,Q)}$ if and only if for every trace `T` in $[P]_{fn(P,Q)}$, the closure of `T` and $[Q]_{fn(P,Q)}$ have a common element.

```

op TRACE-PREORDER-MOD : -> Module .
eq TRACE-PREORDER-MOD = ['TRACE-PREORDER] .

var N : MachineInt .
vars T T1 T2 X : Term .
var TS TS1 TS2 : TermSet .

op _<<_ : TermSet TermSet -> Bool .
op _<<<_ : TermSet Term -> Bool .
op TTraceClosure : Term -> TermSet .
op _maypre_ : Term Term -> Bool .

eq TS2 << mt = true .
eq TS2 << (T1 u TS1) = TS2 <<< T1 and TS2 << TS1 .
eq TS2 <<< T1 = not disjoint?(TS2 , TTraceClosure(T1)) .
eq T1 maypre T2 = TTrmtoNormalTraceSet(T2) << TTrmtoNormalTraceSet(T1) .

```

The computation of the closure of `T` is done by the function `TTraceClosure`. It uses `TTraceClosureAux` to compute all possible (multi-step) rewrites of the term `T` using the rules defined in the module `TRACE-PREORDER`.

```

op TRACE-PREORDER-MOD : -> Module .
eq TRACE-PREORDER-MOD = ['TRACE-PREORDER] .
op TTraceClosureAux : Term Term MachineInt -> TermSet .

eq TTraceClosure(T) = TTraceClosureAux(T, 'TT:TTrace,0) .
eq TTraceClosureAux(T,X,N) =
  if metaSearch(TRACE-PREORDER-MOD,T,X,nil,'*,maxMachineInt,N) == failure then mt
  else getTerm(metaSearch(TRACE-PREORDER-MOD,T,X,nil,'*,maxMachineInt,N)) u
    TTraceClosureAux(T,X,N + 1) fi .

```

This computation is terminating as the number of traces to which a trace can rewrite using the trace preorder laws is finite modulo α -equivalence. This follows from the fact that the length of a trace is non-increasing across rewrites, and the free names in the target of a rewrite are also free names in the source. Since the closure of a trace is finite, `metaSearch` can

be used to enumerate all the traces in the closure. Note that although the closure of a trace is finite, it is possible to have an infinite rewrite that loops within a subset of the closure. Further, since T is a meta-representation of a trace, `metaSearch` can be applied directly to T inside the function `TTraceClosureAux(T,X,N)`.

7 Conclusions and Future Work

In this paper, we have described an executable specification in Maude of the operational semantics of an asynchronous version of the π -calculus using conditional rewrite rules with rewrites in the conditions as proposed by Verdejo and Martí-Oliet in [18], and the CINNI calculus proposed by Stehr in [14] for managing names and their binding. In addition, we also implemented the may testing relation for π -calculus processes using the Maude metalevel, where we use the `metaSearch` operation to calculate the set of all traces for a process and then compare two sets of traces according to a preorder relation between traces. As emphasized throughout the paper, the new features introduced in Maude 2.0 have been essential for the development of this executable specification, including rewrites in conditions, the `frozen` attribute, and the `metaSearch` operation.

An interesting direction of further work is to extend our implementation to the various typed variants of π -calculus. Two specific typed asynchronous π -calculi for which the work is under way are the local π -calculus ($L\pi$) [10] and the Actor model [1, 15]. Both of these formal systems have been used extensively in formal specification and analysis of concurrent object-oriented languages [2, 7], and open distributed and mobile systems [9]. The alternate characterization of may testing for both of these typed calculi was recently published [16, 17]. We are extending the work presented here to account for the type systems for these calculi, and modifications to the trace based characterization of may testing. We plan to include a brief report on these extensions in the final version of this paper.

Acknowledgments

This research has been supported in part by the Defense Advanced Research Projects Agency (Contract numbers: F30602-00-2-0586 and F33615-01-C-1907), the ONR MURI Project *A Logical Framework for Adaptive System Interoperability*, and the Spanish CICYT project *Desarrollo Formal de Sistemas Basados en Agentes Móviles* (TIC2000-0701-C02-01). This work was done while the last author was visiting the Department of Computer Science in the University of Illinois at Urbana-Champaign, for whose hospitality he is very grateful. We would like to thank José Meseguer for encouraging us to put together several complementary lines of work in order to get the results described in this paper.

References

- [1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.
- [2] G. Agha. Concurrent Object-Oriented Programming. *Communications of the ACM*, 33(9):125–141, September 1990.
- [3] M. Boreale, R. De Nicola, and R. Pugliese. Proof techniques for cryptographic protocols. In *Proceedings of 14th Symposium on Logic in Computer Science*. IEEE Computer Science Press, 1999.
- [4] M. Boreale, R. De Nicola, and R. Pugliese. Trace and testing equivalence on asynchronous processes. *Information and Computation*, 172(2):139–164, 2002.

- [5] Peter Borovanský, Claude Kirchner, H el ene Kirchner, Pierre-Etienne Moreau, and Marian Vittek. ELAN: A logical framework based on computational systems. In Jos e Meseguer, editor, *Proceedings First International Workshop on Rewriting Logic and its Applications, WRLA '96, Asilomar, California, September 3–6, 1996*, volume 4 of *Electronic Notes in Theoretical Computer Science*, pages 35–50. Elsevier, 1996. <http://www.elsevier.nl/locate/entcs/volume4.html>.
- [6] Manuel Clavel, Francisco Dur an, Steven Eker, Patrick Lincoln, Narciso Mart ı-Oliet, Jos e Meseguer, and Jos e F. Quesada. Towards Maude 2.0. In Kokichi Futatsugi, editor, *Proceedings Third International Workshop on Rewriting Logic and its Applications, WRLA 2000, Kanazawa, Japan, September 18–20, 2000*, volume 36 of *Electronic Notes in Theoretical Computer Science*, pages 297–318. Elsevier, 2000. <http://www.elsevier.nl/locate/entcs/volume36.html>.
- [7] I.A. Mason and C.Talcott. A semantically sound actor translation. In *ICALP 97*, pages 369–378, 1997. LNCS 1256.
- [8] M.Boreale and R. De Nicola. Testing equivalence for mobile systems. In *Information and Computation*, volume 120, pages 279–302, 1995.
- [9] M. Merro, J. Kleist, and U. Nestmann. Local π -calculus at work: Mobile objects as mobile processes. In *Proceedings of IFIP TCS2000*, 2000. LNCS 1872.
- [10] M. Merro and D. Sangiorgi. On Asynchrony in Name-Passing Calculi. In *Proceeding of ICALP '98*. Springer-Verlag, 1998. LNCS 1443.
- [11] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes (Parts I and II). *Information and Computation*, 100:1–77, 1992.
- [12] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Dept., Aarhus University, September 1981.
- [13] M. Hennesy R. De Nicola. Testing equivalence for processes. *Theoretical Computer Science*, 34:83–133, 1984.
- [14] Mark-Oliver Stehr. CINNI — A generic calculus of explicit substitutions and its application to λ -, ζ - and π -calculi. In Kokichi Futatsugi, editor, *Proceedings Third International Workshop on Rewriting Logic and its Applications, WRLA 2000, Kanazawa, Japan, September 18–20, 2000*, volume 36 of *Electronic Notes in Theoretical Computer Science*, pages 71–92. Elsevier, 2000. <http://www.elsevier.nl/locate/entcs/volume36.html>.
- [15] C. Talcott. An Actor Rewriting Theory. In *Electronic Notes in Theoretical Computer Science 5*, 1996.
- [16] Prasanna Thati, Reza Ziaei, and Gul Agha. A theory of may testing for actors. In *Formal Methods for Open Object-based Distributed Systems*, March 2002.
- [17] Prasanna Thati, Reza Ziaei, and Gul Agha. A theory of may testing for asynchronous calculi with locality and no name matching. In *Proceedings of the 9th International Conference on Algebraic Methodology and Software Technology*. Springer Verlag, September 2002. LNCS.
- [18] Alberto Verdejo and Narciso Mart ı-Oliet. Implementing CCS in Maude 2. Manuscript, Universidad Complutense de Madrid, Spain, submitted to WRLA 2002, 2002.
- [19] Patrick Viry. Input/output for ELAN. In Jos e Meseguer, editor, *Proceedings First International Workshop on Rewriting Logic and its Applications, WRLA '96, Asilomar, California, September 3–6, 1996*, volume 4 of *Electronic Notes in Theoretical Computer Science*, pages 51–64. Elsevier, 1996. <http://www.elsevier.nl/locate/entcs/volume4.html>.

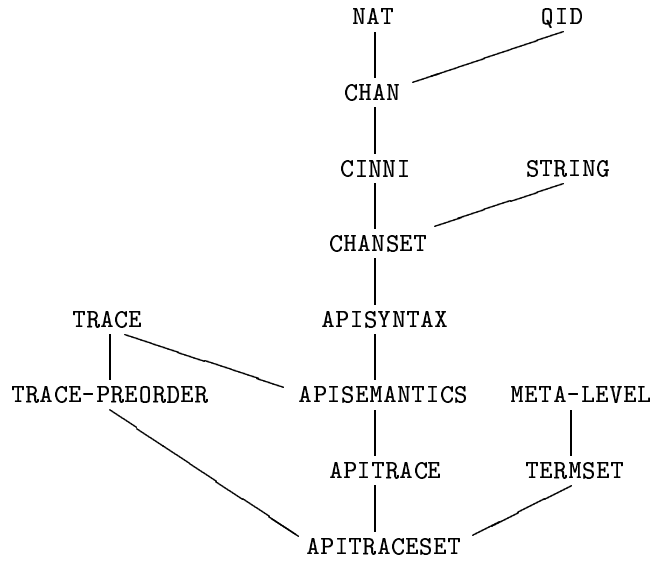


Figure 1: The graph of module importation in our implementation.

A Appendix

The diagram in Figure 1 illustrates the graph of module importation in our implementation that closely follows the structure of the paper. The complete code is available at <http://osl.cs.uiuc.edu/~ksen/api/>. Here we only show the module that contains the rewrite rules for the operational semantics of asynchronous π -calculus (Table 2). The function `genQid` used in the side condition of one of the `RES` rules, generates an identifier that is fresh, i.e. an identifier not used to construct channel names in the set passed as the argument to the function.

```

mod APISEMANTICS is
  inc APISYNTAX .
  inc CHANSET .
  inc TRACE .
  sorts EnvTrm TraceTrm .
  subsort EnvTrm < TraceTrm .

  op [_]_ : Chanset Trm -> EnvTrm [frozen] .
  op {[_]}_ : Action TraceTrm -> TraceTrm [frozen] .
  op notinfn : Qid Trm -> Bool .

  vars N : Nat .
  vars X Y Z : Qid .
  vars CX CY : Chan .
  var CS : Chanset .
  vars A : Action .
  vars P1 Q1 P Q : Trm .
  var SUM : SumTrm .
  var IO : ActionType .

```

```

eq  notinfn(X,P) = not X{0} in freenames(P) .

rl [Inp] : [CY CS] (CX(X) . P) => {f(i,CX,CY)} ([CY CS] ([X := CY] P)) .

rl [Inp] : [CY CS] ((CX(X) . P) + SUM) => {f(i,CX,CY)} ([CY CS] ([X := CY] P)) .

rl [Tau] : [CS] (tau . P) => { tauAct } ([CS] P) .

rl [Tau] : [CS] ((tau . P) + SUM) => { tauAct } ([CS] P) .

crl [BInp] : [CS] P => {b(i,CX,'U')} ['U{0} [shiftup 'U] CS] P1
             if ['U{0} [shiftup 'U] CS] [shiftup 'U] P =>
                {f(i,CX,'U{0})} ['U{0} [shiftup 'U] CS] P1 .

rl [Out] : [CS] CX < CY > => { f(o,CX,CY) } ([CS] nil) .

crl [Par] : [CS] (P | Q) => {f(I0,CX,CY)} ([CS] (P1 | Q))
             if [CS] P => {f(I0,CX,CY)} ([CS] P1) .

crl [Par] : [CS] (P | Q) =>
             {b(I0,CX,Y)} [Y{0} ([shiftup Y] CS)] (P1 | [shiftup Y] Q)
             if [CS] P => {b(I0,CX,Y)} ([Y{0} ([shiftup Y] CS)] P1) .

crl [Com] : [CS] (P | Q) => {tauAct} ([CS] (P1 | Q1))
             if [CS] P => {f(o,CX,CY)} ([CS] P1) /\ [CS] Q => {f(i,CX,CY)} ([CS] Q1) .

crl [Close] : [CS] (P | Q) => {tauAct} [CS] new [Y] ( P1 | Q1)
             if [CS] P => {b(o,CX,Y)} [Y{0} [shiftup Y] CS] P1 /\
                [Y{0} [shiftup Y] CS] [shiftup Y] Q =>
                {f(i,CX,Y{0})} [Y{0} [shiftup Y] CS] Q1 .

crl [Res] : [CS] (new [X] P) => {[shiftdown X] f(I0,CX,CY)} [CS] (new [X] P1)
             if not X{0} in (CX CY) /\ [[shiftup X] CS] P =>
                {f(I0,CX,CY)} [[shiftup X] CS] P1 .

crl [Res] : [CS] (new [X] P) =>
             {[shiftdown X] b(I0,CX,Z)} [Z{0} CS] new [X] ([ Y := Z{0} ] P1)
             if X{0} /= CX /\ Z := genQid(X{0} CS freenames(P)) /\
                [[shiftup X] CS] P =>
                {b(I0,CX,Y)} [Y{0} [shiftup Y] [shiftup X] CS] P1 .

crl [Open] : [CS] (new [X] P) =>
             {[shiftdown X] b(o,CY,X)} [X{0} [shiftup X] CS] P1
             if [[shiftup X] CS] P => {f(o,CY,X{0})} [[shiftup X] CS] P1 /\
                X{0} /= CY .

crl [If] : [CS] (if CX = CY then P else Q fi) => {A} [CS] P1
             if [CS] P => {A} [CS] P1 .

crl [Else] : [CS] (if CX = CY then P else Q fi) => {A} [CS] Q1
             if CX /= CY /\ [CS] Q => {A} [CS] Q1 .

crl [Rep] : [CS] (! P) => {A} [CS] P1 if [CS] (P | (! P)) => {A} [CS] P1 .
endm

```