Executable Specifications of Message-based Concurrency in Maude

Bachelor's Project Thesis

Bastiaan Haaksema, b.haaksema@student.rug.nl
Supervisors: Prof. Dr. Jorge Pérez & Dr. Dan Frumin

Abstract: The π-calculus is a theoretical model of interacting processes, useful to express and reason about message-passing concurrency. It includes formal ways of expressing programs and formal rules for executing their behavior. Executable specifications of process calculi bridge the gap between theory and practice, while their implementations are helpful as tools for reasoning about the trustworthiness of software. By creating an executable specification for π-calculus with reduction semantics, we discover that its design and implementation are influenced by the π-calculus syntax and axioms of structural congruence.

1 Introduction

In the field of fundamental computing, there is interest in the formal modeling and reasoning of concurrent systems. Much research on this topic has been conducted by Milner, who introduced the Calculus of Communicating Systems (CCS) [3]. He then worked with Parrow and Walker to extend the CCS by allowing processes to communicate the names of communication channels themselves, which resulted in the π-calculus [4]. There are many variants of the π-calculus, which can differ in their operational semantics. The operational semantics describe transitions in the calculus as individual computational steps and can be classified as either semantics based labeled transition system (LTS) or reduction semantics. A LTS describes the behavior of an open system, where the labels describe the interaction of the system with its environment. In contrast, reduction semantics describe the behavior of a closed system, where no external interactions are possible, thus removing the need for labels. Both of these approaches have their applications, though the key benefit of reduction semantics is the simplified notation.

Shifting from paper-based specifications of π-calculus processes to specifications executable by machines is an important step to increase our confidence in formal specifications and to streamline reasoning, thus bridging the gap between theory and practice. The many variations of the π-calculus result in numerous distinct executable specification implementations. The first specification of the π-calculus in reduction style was introduced by Viry [13]. It made use of De Bruijn indexes and explicit substitutions in the language ELAN. Stehr suggested improving this specification with the use of his Generic Calculus of Explicit Substitutions (CINNI) [9], which combines the usage of standard variables and De Bruijn indices. Stehr used the rewrite engine Maude to make this calculus executable. Thati et al. observed all the preceding work, and created an executable specification of asynchronous π-calculus with a LTS in Maude using rewriting logic [10]. This was soon followed by the implementation of typed mobile ambients [7], and more recently, with an implementation of the π-calculus variant privacy calculus [6], both of which also utilized CINNI in Maude. A more general overview of rewriting logic is given by Meseguer [2].

We are particularly interested in π-calculus with reduction semantics because of its simple representation, and would like to use it to practically analyze and reason about message-based concurrency. Therefore, we have created an executable specification for π-calculus with reduction semantics in Maude. This variant of π-calculus has a different syntax than the original π-calculus for which Viry created their executable specification [13]. Besides a difference in the syntax, this executable specification features an alternative substitution calculus, based on CINNI [9]. The Maude implementation of asynchronous π-calculus and similar work have been used as examples of operational semantics implementations [10, 11]. Because of the development of our executable specification, we will be able to answer the question: What are the design choices when creating and implementing an executable specification of π-calculus with reduction semantics?

Before the executable specification can be presented, the variant of the π-calculus and its properties have to be considered. Section 2 will describe a π-calculus variant with reduction semantics derived from Sangiorgi's [8]. Afterwards, Section 3 will present the corresponding executable specification formal notation, its analysis and comments on the implementation in Maude. Readers only interested in the practical application of the implemented executable specification can skip to Section 4, after confirming the presented π-calculus variant suits their needs. Finally, Section 5 will conclude by answering the research question.

2 The π-calculus variant

2.1 Syntax and substitution

In their introduction to the π-calculus, Parrow stated that process terms should satisfy the following three conditions when working with reduction semantics [5]:

1. All sums are guarded sums, i.e., of kind $\alpha_1.P_1 + \cdots + \alpha_n.P_n$ (where α are prefixes).
2. There are no unguarded match-operators.
3. There are no τ-prefixes.

The syntax in Table 2.1 neatly satisfies the first condition by only allowing summations as the operands of sums. For the sake of simplifying the coming executable specification, the non-essential match operator has been omitted, automatically satisfying the second condition. Without the match operator there is a loss of expressiveness, though a similar effect can be achieved by exploiting the parallel composition operator [5]. Another difference with respect to the original syntax definition by Sangiorgi is the replacement of replication by the input-guarded replication. This change has implications which will be elaborated on later. This syntax definition deviates from the third condition, by allowing the appearance of τ-prefixes. This is compensated for by the addition of the `R-TAU` reduction rule, which does not appear in Parrow's explanation of reduction semantics.

`GUARD`	α	$::=$	τ	$\mid$	$\overline{a}b$	$\mid$	$a(x)$		
`TERM`	P	$::=$	M	$\mid$	$P \mid P'$	$\mid$	$\nu x\ P$	$\mid$	$!a(x).P$
`SUMTERM`	M	$::=$	0	$\mid$	$\alpha.P$	$\mid$	$M + M'$		

Table 2.1: Syntax definition

Here is a compact summary describing the syntax. 0 is the process that cannot perform any actions. The output-prefixed process $\overline{a}x.P$ sends the name x via a and continues as P. The input-prefixed process $a(x).P$ receives a name along a and continues as $P\{b/x\}$, meaning all free occurrences of x in P are substituted by the received name b. The silent-prefixed process $\tau.P$ can evolve to P without any external interaction. Also known as the choice operator, the sum $P + P'$

represents a process that can enact either P or P'. The composition $P \mid P'$ represents P and P' acting in parallel; they can proceed independently and can also interact with each other. The restriction $\nu x\ P$ binds x in scope P, meaning it cannot be used for communication between P and other processes. Finally, a process with input-guarded replication $!a(x).P$ provides a copy of P for every received name along a. To avoid ambiguity when writing processes as linear expressions, the prefixing, restriction and input-guarded replication operators enjoy a higher precedence than choice and parallel composition.

The restriction $\nu x\ P$ and the input-prefixed process $a(x).P$, and by extension the input-guarded replication, are able to bind the name x in the scope P. An occurrence of a name in a process is free if it is not bound in its scope. As seen while describing the input prefix, there is an implicitly defined substitution operator $P\{b/x\}$, with the important aspect that this operator replaces all free occurrences of x with b. To avoid unintended capture of names by binders, it is possible to change bound names in a process by α-conversion (an example will be given in Section 2.3). Substitutions $P\sigma$ are applied according to the substitution applications of Table 2.2, where $=$ denotes definitional equality.

0σ	$=$	0	$(\overline{a}b.P)\sigma$	$=$	$\overline{a\sigma}b\sigma.P\sigma$
$(\tau.P)\sigma$	$=$	$\tau.P\sigma$	$(a(x).P)\sigma$	$=$	$a\sigma(x).P\sigma$
$(P \mid P')\sigma$	$=$	$P\sigma \mid P'\sigma$	$(!a(x).P)\sigma$	$=$	$!a\sigma(x).P\sigma$
$(P + P')\sigma$	$=$	$P\sigma + P'\sigma$	$(\nu x\ P)\sigma$	$=$	$\nu x\ P\sigma$

Table 2.2: Substitution application

2.2 Structural congruence

Before being able to discuss the reduction rules, we have to take a look at the relation of structural congruence. This relation, denoted by $\equiv$, identifies processes which intuitively represent the same thing because of their structure. This property is formalized by the axioms of structural congruence in Table 2.3. Readers familiar with the work of Sangiorgi, might be missing the axioms of structural congruence involving the match and replication operator. These two axioms have been removed for our variant because of the omission of the match operator and the replacement of replication by input-guarded replication. The first six axioms show that the sum and parallel composition operators are associative and communicative with identity element 0 while the remaining three axioms represent the laws of scope extension.

SC-SUM-ASSOC	$M_1 + (M_2 + M_3)$	$\equiv$	$(M_1 + M_2) + M_3$	
SC-SUM-COMM	$M_1 + M_2$	$\equiv$	$M_2 + M_1$	
SC-SUM-INACT	$M + 0$	$\equiv$	M	
SC-COMP-ASSOC	$P_1 \mid (P_2 \mid P_3)$	$\equiv$	$(P_1 \mid P_2) \mid P_3$	
SC-COMP-COMM	$P_1 \mid P_2$	$\equiv$	$P_2 \mid P_1$	
SC-COMP-INACT	$P \mid 0$	$\equiv$	P	
SC-RES	$\nu x\ \nu y\ P$	$\equiv$	$\nu y\ \nu x\ P$	
SC-RES-INACT	$\nu x\ 0$	$\equiv$	0	
SC-RES-COMP	$\nu x\ (P_1 \mid P_2)$	$\equiv$	$P_1 \mid \nu x\ P_2$	if $x \notin \mathtt{fn}(P_1)$

Table 2.3: Structural congruence

The SC-RES-COMP axiom shows that the scope of the restriction $\nu x\ P_2$ can be extended to P_1 if x is not a free name in P_1. Conversely, in the case where x is a free name in P_1, the scope can still be extended by applying α-conversion such that x in P_2 is changed to a name which is not free in P_1. The axioms of structural congruence, together rules of equational reasoning of Table 2.4 define the relation of structural congruence. The rules of equational reasoning show the reflexive, symmetric, transitive and context properties of the relation. The CONG equational reasoning rule uses the definition of contexts by Sangiorgi [8].

REFL	$P_1 \equiv P_1$		
SYMM	$P_1 \equiv P_2$	implies	$P_2 \equiv P_1$
TRANS	$P_1 \equiv P_2$ and $P_2 \equiv P_3$	implies	$P_1 \equiv P_3$
CONG	$P_1 \equiv P_2$	implies	$C[P_1] \equiv C[P_2]$

Table 2.4: Equational reasoning

2.3 Operational semantics

Now that the syntax and structural congruence of our π-calculus variant have been introduced, we can move on to analyze its reduction relation, $\longrightarrow$, on processes. Intuitively, the statement $P \longrightarrow P'$ expresses that process P reduces to process P' by means of an action within P. Considering the process P which has two components, with one of these components having the capability to send along a while the other can receive along a. The inference rule of communication reduction (R-COM) shows that this P can have a reduction because of the interaction between its components. Since replication is not represented as an axiom of structural congruence, a second variant of the reduction communication rule is introduced for the input-guarded replication. In this rule of input-guarded replication reduction (R-REP), the receiving component $!a(x).P$ is split into the two components $P\{b/x\}$ (where b is the received name) and $!a(x).P$ after reduction.

$$\texttt{R-COM}\quad \frac{}{(\overline{a}b.P_1 + M_1) \mid (a(x).P_2 + M_2) \longrightarrow P_1 \mid P_2\{b/x\}}$$

$$\texttt{R-REP}\quad \frac{}{(\overline{a}b.P_1 + M_1) \mid (!a(x).P_2 + M_2) \longrightarrow P_1 \mid P_2\{b/x\} \mid !a(x).P_2}$$

$$\texttt{R-TAU}\quad \frac{}{\tau.P + M \longrightarrow P}$$

$$\texttt{R-PAR}\quad \frac{P_1 \longrightarrow P_1'}{P_1 \mid P_2 \longrightarrow P_1' \mid P_2} \qquad \texttt{R-RES}\quad \frac{P \longrightarrow P'}{\nu x\ P \longrightarrow \nu x\ P'}$$

$$\texttt{R-STRUCT}\quad \frac{P_1 \equiv P_2,\ P_2 \longrightarrow P_2',\ P_1' \equiv P_2'}{P_1 \longrightarrow P_1'}$$

Table 2.5: Reduction Rules

As promised in Section 2.1 we also introduce another rule R-TAU as counterpart to the communication rules to manage τ-prefixes that represent an internal action within a process. Additionally, the R-PAR and R-RES rules allow reductions under parallel composition and restriction. Without these two rules, a process P_1 would not be able to reduce to P_1' if composed in parallel with another process or if it lives under a restriction.

Finally, the structural congruence reduction rule is required to allow applications of the axioms of structural congruence when working with reductions. One might consider the process term $\tau.P$ and its intuitive reduction to P, and notice how this reduction is depended on an application of the structural congruence axiom `SC-SUM-INACT`. Because the `R-STRUCT` rule freely allows these applications, we don't have to define new rules that could be created by the axioms of structural congruence.

3 The executable specification

3.1 Extending CINNI$_\pi$

When working on the π-calculus with pen and paper, α-conversion is an intuitive a convenient way of avoiding capture when name clashes occur. Unfortunately, this exact process is quite difficult for a machine to replicate. The idea behind CINNI is to solve this problem by adding indices to the representation of names and explicitly defining the substitution operator, hence why CINNI is called an explicit substitution calculus. There are other explicit substitution calculi based on De Bruijn indices that replace variables with indices. The benefit of CINNI over these other explicit substitution calculi is the smaller difference between our π-calculus and the executable specification. The idea is to eliminate the need for α-conversion by keeping the bound names inside the binders as they are, but to replace their 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. In other words, occurrences of the names in the π-calculus: a, $b\ldots$ are replaced by combinations of a name and an index: $a\{i\}$, $b\{j\}\ldots$, while the binders remain unchanged.

The CINNI substitution calculus contains the two kinds of basic substitutions $[a := x]$ (simple) and $\uparrow a$ (shift). In addition, substitutions can be lifted using $\Uparrow a(\sigma)$ (lift), where σ ranges over all three kinds of CINNI substitutions. To differentiate the explicit CINNI substitutions from the implicit π-calculus substitutions, we write them before instead of after the terms. All future mentioned substitutions can be assumed to be explicit CINNI substitutions. In Table 3.1, the equations of simple substitution on the left show that the name a is substituted by x whenever the name a is accompanied by index 0 while decreasing all other indices with a higher value. Next, the equations on the top right of the table show that the shift operation increments the indices corresponding to the specified name. Finally, the lift operation is used to adjust basic substitutions such that they skip one index (we will see a use for this shortly). The operator equations also show that all the substitutions involving a have no effect on all other names b, where $a \neq b$.

$$
\begin{array}{rcl}
[a := x]\ a\{0\} & \xrightarrow{=} & x \\
[a := x]\ a\{n+1\} & \xrightarrow{=} & a\{n\} \\
[a := x]\ b\{n\} & \xrightarrow{=} & b\{n\}
\end{array}
\qquad
\begin{array}{rcl}
\uparrow a\ a\{n\} & \xrightarrow{=} & a\{n+1\} \\
\uparrow a\ b\{n\} & \xrightarrow{=} & b\{n\} \\
\Uparrow a(\sigma)\ a\{0\} & \xrightarrow{=} & a\{0\} \\
\Uparrow a(\sigma)\ a\{n+1\} & \xrightarrow{=} & \uparrow a\ (\sigma a\{n\}) \\
\Uparrow a(\sigma)\ b\{n\} & \xrightarrow{=} & \uparrow a\ (\sigma b\{n\})
\end{array}
$$

Table 3.1: Explicit substitution on variables

The original instantiation of CINNI for π-calculus (CINNI$_\pi$) also contains equations depended on the π-calculus syntax to define applications of substitution on process terms [9]. For our purposes, CINNI$_\pi$ is extended according to the syntax of Table 2.1, such that it includes the τ-prefix, choice operator and input-guarded replication. Of this extension, called CINNI$_\pi^+$, the syntax-depended equations are shown in Table 3.2. These equations show applications of substitutions in the π-calculus from Table 2.2 adapted to the explicit CINNI substitutions. The lift operation is used to adjust substitutions when they move into a new scope. For the equation involving the parallel composition operator, it is assumed that both P and P' are not 0.

$$
\begin{array}{rclrcl}
\sigma 0 & \xrightarrow{=} & 0 & \sigma(\overline{a}b.P) & \xrightarrow{=} & \overline{\sigma a}\sigma b.\sigma P \\
\sigma(\tau.P) & \xrightarrow{=} & \tau.\sigma P & \sigma(a(x).P) & \xrightarrow{=} & \sigma a(x).(\Uparrow x(\sigma)P) \\
\sigma(P \mid P') & \xrightarrow{=} & \sigma P \mid \sigma P' & \sigma(!a(x).P) & \xrightarrow{=} & !\sigma a(x).(\Uparrow x(\sigma)P) \\
\sigma(P + P') & \xrightarrow{=} & \sigma P + \sigma P' & \sigma(\nu x\ P) & \xrightarrow{=} & \nu x\ (\Uparrow x(\sigma)P)
\end{array}
$$

Table 3.2: Explicit substitution on terms

Additionally, process congruence is generated by the equations of associativity, commutativity and identity for the choice and parallel composition operators in combination with the equations of scope extension in Table 3.3. When extending the scope of a restriction to a process P_1, the shift operator is used to increment the index of all occurrences of the restricted name in P_1. The conditions of the scope extension equations avoid non-termination and presuppose a total order on names.

$$
\begin{array}{rcll}
\nu x\ 0 & \xrightarrow{=} & 0 & \\
\nu x\ \nu y\ P & \xrightarrow{=} & \nu y\ \nu x\ P & \text{if } y < x \\
P_1 \mid \nu x\ P_2 & \xrightarrow{=} & \nu x\ (\uparrow x\ P_1 \mid P_2) & \text{if } P_1 \neq 0 \neq P_2
\end{array}
$$

Table 3.3: Scope extension equations

Up until now, $a \xrightarrow{=} b$ could be simply interpreted as an equation $a = b$. However, by regarding these equations as rewrite rules, they equip the CINNI$_\pi^+$ explicit substitution calculus with operational semantics to obtain a rewrite system. Just like Stehr showed that the rewrite relations induced by the equations of CINNI$_\pi$ are convergent (confluent and terminating) [9], the same can be shown for CINNI$_\pi^+$ because of the almost identical design of the executable specification so far. This property of convergence will be affected by the following introduction of the reduction relation to the executable specification as shown by the analysis of the executable specification in Section 3.2.

$$
\begin{array}{llcl}
\texttt{R-COM} & (\overline{a}b.P_1 + M_1) \mid (a(x).P_2 + M_2) & \longrightarrow & P_1 \mid [x := b]\,P_2 \\
\texttt{R-REP} & (\overline{a}b.P_1 + M_1) \mid (!a(x).P_2 + M_2) & \longrightarrow & P_1 \mid [x := b]\,P_2 \mid !a(x).P_2 \\
\texttt{R-TAU} & \tau.P + M & \longrightarrow & P
\end{array}
$$

Table 3.4: Reduction rewrite rules

The reduction relation in the executable specification is defined by the rewrite rules of Table 3.4. These rules are an adaptation of the three inference rules `R-COM`, `R-REP` and `R-TAU` from Section 2.3 to the explicit substitution calculus. In the executable specification it is ensured that, in conformance with the π-calculus, reduction never takes place inside terms of the form $\overline{a}b.P_1$ or $a(x).P_2$ while still allowing for reductions under parallel composition and restriction without needing the `R-PAR` and `R-RES` rules. This can be implemented with a suitable strategy for when the rewrite rules should be executed. Additionally, the characteristics of the `R-STRUCT` rule are provided by the equational representation of structural congruence from before, since $P_1 \longrightarrow P_1'$ can be achieved by $P_1 \xrightarrow{=} P_2 \longrightarrow P_2' \xrightarrow{=} P_1'$.

3.2 Analysis

In the π-calculus as introduced in Section 2, processes are considered modulo the set of structural equations consisting of α-conversion and the axioms of structural congruence in Table 2.3. Let's call the equational theory entailed by these axioms SN', for Structural equivalence of processes with Names. Reduction of processes modulo SN' according to the inference rules of Section 2.3 is denoted by $P_1/_{\mathrm{SN'}} \longrightarrow P_2/_{\mathrm{SN'}}$. Then, in Section 3, processes modulo CINNI$_\pi^+$ were said to be considered modulo the equations of associativity, commutativity and identity for the choice and parallel composition operators in combination with the equations of scope extension in Table 3.3. The reduction of processes modulo CINNI$_\pi^+$ enabled by the rewrite rules of Table 3.4 is designed such that $P_1/_{\mathrm{CINNI}_\pi^+} \longrightarrow P_2/_{\mathrm{CINNI}_\pi^+}$ iff $P_1/_{\mathrm{SN'}} \longrightarrow P_2/_{\mathrm{SN'}}$, meaning that a process P_1 expressed in SN' or CINNI$_\pi^+$ reduces to the same process P_2 expressed in that respective theory.

Both Stehr and Viry devote their attention to the convergent property of their obtained rewrite systems [9, 12]. Viry ensures convergence of their rewrite system by introducing a rewrite strategy to restricting applications of the expansion and replication theorems to be bounded by the size of the input term. Stehr did not need such a rewrite system, since they used a smaller π-calculus without the choice and replication operators that are essential to those theorems.

The divergence caused by the expansion theorem is demonstrated by this process term with the choice operator:

$$(\overline{a}b.0 + \tau.0) \mid a(x).0 \quad \longrightarrow \quad 0 \mid [x := b]\,0 \quad \vee \quad 0 \mid a(x).0$$

In our Maude implementation of the executable specification, it is the user's responsibility to ensure convergence. This is made possible by the powerful `search` command and the possibility to assign bounds for termination, as will be explained in Section 4. Using this method for π-calculus with conventional replication would require special attention to termination for every occurrence of a process term involving replication. This was the motivation behind the replacement of replication by input-guarded replication discussed in Section 2.1. Since an input-guarded process term is only able to replicate itself when receiving a name, instead of at all times, it allows the use of some controlled form of replication without worrying about termination as much. Non-termination is not entirely eliminated, as demonstrated by this process term that can be rewritten to itself:

$$\overline{a}b.0 \mid !a(x).\overline{a}x.0 \quad \longrightarrow \quad 0 \mid (\overline{a}x.0)\{b/x\} \mid !a(x).\overline{a}x.0 \quad \overset{=}{\longrightarrow} \quad \overline{a}b.0 \mid !a(x).\overline{a}x.0$$

Though, these kinds of situations can be avoided by carefully considering the input process term.

3.3 Maude implementation

In our implementation, `sorts` and `subsorts` are used to mirror the three layers of the syntax (Guards, Terms, and Sums). The sorts `Term` and `SumTerm` are connected with the subsort relation to represent $P ::= M$, while the `Guard` sort is used in the prefix operator as expected. Another sort `Input` is introduced as a subsort of `Guard` which is used to construct the input-guarded replication operator $!a(x).P$. Built-in Maude operator attributes were used for the representation of precedence, associativity, commutativity and identity properties, which are easily added to π-calculus operators as needed. Binders of the input prefix and restriction are represented by a quoted identifier, whereas a channel name (sort `Chan`) is represented by the combination of a quoted identifier (the name) and a natural number (the index). Since substitutions occur on both process terms and channels, the operator for substitution in Maude is overloaded for both of these sorts.

All the equations and rewrite rules introduced in Section 3.1 can then be expressed using the already implemented syntax. Though, for one of the conditions of the scope extension laws, we need to be able to compare names for one of the restrictions. Since binders are represented by quoted identifiers and every quoted identifier has a unique obtainable corresponding string, a string comparison operation can be made. This comparison uses the lexicographic order, where characters are compared going through their ASCII codes. Some aspects of the executable specification are implicitly implemented by Maude behavior, such as equational reasoning and the execution strategy. The Maude implementation described in this section can be found in its entirety in Appendix A.

4 Executable specification usage and Maude examples

As mentioned in the previous section, to make our specification executable, it has been implemented in the programming language Maude. For our purposes, the Maude environment revolves around the commands: `rewrite` and `search`. These commands can be used after loading the executable specification modules and setting the user preferred display options. For repeated use of the executable specification it is recommended to save these steps in a Maude script, similar to `interpreter.maude` in Appendix A. The following description and examples of the `rewrite` and `search` commands are influenced by their description in the Maude Manual [1], just tuned to our use case.

The `rewrite` command, causes the specified term to be rewritten using the rules and equations of the currently loaded modules (unless otherwise specified by the user) and has the following structure:

```
rewrite [depth] term .
```

The optional depth bound restricts the number of rewrite rule applications, it can be omitted if the user is certain the process term only has a finite number of reductions. The term has to be entered according to the syntax implementation of the executable specification, where binders are represented by quoted identifiers, while names also have an index. For example, $a(x).\overline{x}b.0 \mid \nu b\ \overline{a}b.\tau.0$ can be rewritten by Maude with the command:

```
rewrite 'a{0}('x) . 'x{0} < 'b{0} > . nil | ['b] 'a{0} < 'b{0} > . tau . nil .
```

To which the Maude environment will respond with: `['b]'b{0} < 'b{1} > . nil`. An interpretation from this result is that the term has been rewritten to $\nu b'\ \overline{b'}b.0$ (where b' is the result of α-conversion on b), exactly as expected after applications of the rewrite rules and structural congruence axioms. The process term from Section 3.2, that was used to demonstrate non-termination in the executable specification, $\overline{a}b.0 \mid !a(x).\overline{a}x.0$, can be rewritten using the optional depth bound as follows:

```
rewrite [128] 'a{0} < 'b{0} > . nil | !  'a{0}('x) . 'a{0} < 'x{0} > . nil .
```

Maude has no problem with this input, as after 128 rewrite rule applications it will respond with a solution identical to the input term. While quite simple to use, this `rewrite` command can only find the first solution that cannot be further rewritten. The `search` command transcends these limitations by also being able to find intermediate solutions and solutions of divergent process terms. The command has the structure:

```
search [solutions, depth] term searchtype pattern
```

The four searchtypes and the solutions they find are described by the following list, where one step is one rewrite rule application:

`=>1`	one-step solutions
`=>+`	one or more steps solutions
`=>*`	zero or more steps solutions
`=>!`	only final solutions, that cannot be further rewritten

Similar to the `rewrite` command, the `search` command has an optional bound `depth` for restricting the number of rewrite rule applications. Another optional bound `solutions` restricts the number of solutions that can be found. The search pattern describes what kind of terms the solutions should consist of. The interesting search pattern for our use case is `X:Term`, which searches for some solution `X` of the sort `Term` (including `SumTerm` because of the subsort relation). To demonstrate the power of the `search` command, we take a look at one final example. Solutions to the process term from Section 3.2, that was used to demonstrate divergence in the executable specification, $(\overline{a}b.0 + \tau.0) \mid a(x).0$, can be found with the following `search` command:

```
search 'a{0} < 'b{0} > . nil + tau . nil | 'a{0}('x) . nil =>+ X:Term .
```

Where only one solution would have been found with the `rewrite` command, this `search` command will find the two solutions that explore both of the capabilities of the sum term.

5 Conclusions

Creating and implementing an executable specification of π-calculus with reduction semantics allowed for the discovery of design choices and their effect. The first and most important consideration is the syntax of the desired π-calculus variant. For this project, the syntax had to be compatible with reduction semantics and contain all operators we deemed essential, including some kind of replication. The replacement of replication by input-guarded replication has led to a functioning form of replication in the executable specification without a special strategy for replication theorem applications, though, with the unfortunate effect of non-termination. Another challenging aspect of the executable specification was representing structural congruence, and specifically the scope extension laws. Their implementation using directional equations is made possible by the choice to only allow scopes to expand when rewriting process terms. Finally, the execution strategy mentioned towards the end of Section 2.3 should only allow application of the rewrite rules under appropriate contexts. This should be taken into account when deciding the programming language for the implementation of the executable specification. While Maude was very suitable for this project, one might also consider the other popular rewriting logic languages CafeOBJ and ELAN [2].

Of course, the presented executable specification and its implementation have room for improvement. An example would be the automatic translation of process terms in formal syntax to and from their representation in Maude syntax, this could be implemented utilizing Maude's `META-LEVEL`. The executable specification itself could be improved by enforcing termination with respect to replication or the addition of the match operator, the latter of which could be implemented by a strategy that only evaluates the match operators not under an input prefix.

Acknowledgements

This bachelor's thesis was written at the conclusion of my time in the Computing Science programme at the University of Groningen. I want to thank my peer Gökçe Ece Aşırdizer for proofreading this document and my supervisors Prof. Dr. Jorge Pérez and Dr. Dan Frumin for their support, guidance and feedback during the entirety of the project.

References

[1] Manuel Clavel et al. Maude Manual (Version 3.1). SRI International, Menlo Park, CA 94025, USA, 2020. URL `http://maude.cs.illinois.edu/w/images/6/62/Maude-3.1-manual.pdf`.

[2] José Meseguer. Twenty years of rewriting logic. The Journal of Logic and Algebraic Programming, 81(7):721–781, 2012. ISSN 1567-8326. doi:https://doi.org/10.1016/j.jlap.2012.06.003. Rewriting Logic and its Applications.

[3] Robin Milner. A Calculus of Communicating Systems, volume 92 of Lecture Notes in Computer Science. Springer, first edition, 1980.

[4] Robin Milner et al. A calculus of mobile processes, i/ii. Information and Computation, 100 (1):1–77, 1992. ISSN 0890-5401.

[5] Joachim Parrow. Chapter 8 - An Introduction to the π-Calculus. In Handbook of Process Algebra, pages 479–543. Elsevier Science, Amsterdam, 2001. ISBN 978-0-444-82830-9. doi:https://doi.org/10.1016/B978-044482830-9/50026-6.

[6] Georgios Pitsiladis and Petros Stefaneas. Implementation of Privacy Calculus and Its Type Checking in Maude. In Leveraging Applications of Formal Methods, Verification and Validation. Verification, pages 477–493, Cham, 2018. Springer International Publishing. ISBN 978-3-030-03421-4.

[7] Fernando Rosa-Velardo et al. Typed Mobile Ambients in Maude. Electronic Notes in Theoretical Computer Science, 147(1):135–161, 2006. ISSN 1571-0661. doi:https://doi.org/10.1016/j.entcs.2005.06.041. Proceedings of the 6th International Workshop on Rule-Based Programming (RULE 2005).

[8] Davide Sangiorgi and David Walker. The Pi-Calculus - a theory of mobile processes. Cambridge University Press, 2001. ISBN 978-0-521-78177-0.

[9] Mark-Oliver Stehr. CINNI - A Generic Calculus of Explicit Substitutions and its Application to λ- σ- and π-Calculi. Electronic Notes in Theoretical Computer Science, 36:70–92, 2000. ISSN 1571-0661. doi:https://doi.org/10.1016/S1571-0661(05)80125-2. The 3rd International Workshop on Rewriting Logic and its Applications.

[10] Prasanna Thati et al. An Executable Specification of Asynchronous Pi-Calculus Semantics and May Testing in Maude 2.0. Electronic Notes in Theoretical Computer Science, 71: 261–281, 2004. ISSN 1571-0661. doi:https://doi.org/10.1016/S1571-0661(05)82539-3. WRLA 2002, Rewriting Logic and Its Applications.

[11] Alberto Verdejo and Narciso Martí-Oliet. Implementing CCS in Maude 2. Electronic Notes in Theoretical Computer Science, 71:282–300, 2004. ISSN 1571-0661. doi:https://doi.org/10.1016/S1571-0661(05)82540-X. URL `https://www.sciencedirect.com/science/article/pii/S157106610582540X`. WRLA 2002, Rewriting Logic and Its Applications.

[12] Patrick Viry. A Rewriting Implementation of pi-calculus. Technical Report 96-30, Università di Pisa, 1996. URL `http://eprints.adm.unipi.it/1952/`.

[13] Patrick Viry. Input/Output for ELAN. Electronic Notes in Theoretical Computer Science, 4: 51–64, 1996. ISSN 1571-0661. doi:https://doi.org/10.1016/S1571-0661(04)00033-7. RWLW96, First International Workshop on Rewriting Logic and its Applications.

A Appendix: Executable specification of π-calculus with reduction semantics

cinni.maude

```
fmod CINNI is
  protecting QID .

  *** Channel definition
  sort Chan .
  op _{_} : Qid Nat -> Chan [ctor prec 1] .

  *** Calculus independent equations of CINNI
  sort Subst .
  op [_:=_] : Qid Chan -> Subst .
  op [shift_] : Qid -> Subst .
  op [lift__] : Qid Subst -> Subst .
  op __ : Subst Chan -> Chan .

  vars a b : Qid .
  var n : Nat .
  var x : Chan .
  var S : Subst .

  eq [a := x] a{0} = x .
  eq [a := x] a{s(n)} = a{n} .
  ceq [a := x] b{n} = b{n} if a =/= b .

  eq [shift a] a{n} = a{s(n)} .
  ceq [shift a] b{n} = b{n} if a =/= b .

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

syntax.maude

```
fmod SYNTAX is
  protecting CINNI .

  *** Syntax definition
  sorts Guard Input .
  subsort Input < Guard .
  op tau : -> Guard [ctor] .
  op _<_> : Chan Chan -> Guard [ctor prec 4] .
  op _(_) : Chan Qid -> Input [ctor prec 3] .

  sort Term .
  op _|_ : Term Term -> Term [ctor assoc comm prec 9 id: nil] .
  op [_]_ : Qid Term -> Term [ctor prec 7] .
  op !_._ : Input Term -> Term [ctor prec 6] .

  sort SumTerm .
  subsort SumTerm < Term .
  op nil : -> SumTerm [ctor] .
  op _._ : Guard Term -> SumTerm [ctor prec 5] .
  op _+_ : SumTerm SumTerm -> SumTerm [ctor assoc comm prec 8 id: nil] .

  *** Scope extension
  vars P P1 P2 : Term .
  vars x y : Qid .

  eq [x] nil = nil .
  ceq [x] P1 | P2 = [x] (P1 | [shift x] P2) if P1 =/= nil and P2 =/= nil
    .
  ceq [x] [y] P = [y] [x] P if string(y) < string(x) .

  *** Substitution rules
  op __ : Subst Term -> Term [ctor prec 2] .

  var S : Subst .
  vars a b : Chan .

  eq S (tau . P) = tau . (S P) .
  eq S (a < b > . P) = (S a) < (S b) > . (S P) .
  eq S (a(x) . P) = (S a)(x) . ([lift x S] P) .
  ceq S (P1 | P2) = (S P1) | (S P2) if P1 =/= nil and P2 =/= nil .
  eq S ([x] P) = [x] ([lift x S] P) .
  eq S (! a(x) . P) = ! (S a)(x) . ([lift x S] P) .
  eq S nil = nil .
  eq S (P1 + P2) = (S P1) + (S P2) .
endfm
```

semantics.maude

```
mod SEMANTICS is
  protecting SYNTAX .

  vars P1 P1' P2 : Term .
  vars M1 M2 : SumTerm .
  var x : Qid .
  vars a b : Chan .

  *** Reduction rules without alpha equivalence
  rl [R-COM] : a < b > . P1 + M1 | a(x) . P2 + M2 => P1 | [x := b] P2 .
  rl [R-REP] : a < b > . P1 + M1 | ! a(x) . P2 + M2 => P1 | [x := b] P2 |
     ! a(x) . P2 .
  rl [R-TAU] : tau . P1 + M1 => P1 .
endm
```

interpreter.maude

```
load cinni .
load syntax .
load semantics .

set show advisories off .
set show command off .
```