Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

A Deep Embedding of μCRL in Lean

Author: Wolf bij 't Vuur (2605356)

1st supervisor: Jasmin Blanchette
daily supervisor: Anne Baanen
2nd reader: Jannis Limperg

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

March 3, 2023

Abstract

In this thesis we cover the partial formalization of the process algebra of μCRL, defined in the Lean Theorem Prover. We use the transition rules of μCRL to define equivalence of processes in Lean, and show that the proof theory of μCRL can be derived from this formalization. Afterwards, we will formalize the Alternating Bit Protocol as a case study, though some complications prevented us from proving correctness.

Contents

List of Figures

LIST OF FIGURES

1

Introduction

Process algebra (6) is a section of theoretical computer science focused on describing computer programs at a low level, in order to more easily verify their correctness. In particular, the focus is on a small set of operators which receive meaning through axioms, expressed as a set of equations. Two process terms that are built from these operators represent equivalent behaviours if they can be equated through these axioms. Process algebra is used for concurrent systems, where it can analyze parallelism. In these systems, most analysis is done on process equations, which can be linearized by a specific procedure. After linearization, the external behaviour can be extracted, from which we can prove correctness. A main part of linearization concerns a specific operator, *hiding*, which is used to abstract away internal actions. This allows the user to focus on the requested external behaviour.

Proof assistants (1) are used to formalize concepts, and verify that proofs are correct. Proof assistants are also called interactive theorem provers. They are interactive, as opposed to automated theorem provers, because they require input from a user in order to find their proofs. Assuming that the theory behind the proof assistant is correct, the user can use the proof assistant to formalize their proofs, and be more certain about the correctness of what they wrote. Many theorems have been formalized in a proof assistant, and sometimes proof assistants were able to find errors in proofs that would otherwise have been overlooked.

Why not, then, verify the verifier? Hence, we set out to formalize a process algebra, called μCRL (9, 10), in a proof assistant, called Lean (2, 8). We take inspiration from an earlier formalization of μCRL by Bezem and Bol in a different theorem prover (7), but aim to formalize in a different way. We wish to make full use of Lean's logic, and hence try to build μCRL from the ground up, deriving the axioms from the base rules. Afterwards, we formalize the same case study as Bezem and Bol, the Alternating Bit Protocol (ABP).

Our question is: can we formalize μCRL into Lean in a natural way, and how intuitive can we make our formalization ?

A full formalization, as well as a full correctness proof for our case study, would have been ideal. In working towards this, we were able to get most of the operators for μCRL into Lean. This was done in an intuitive way, using the transition rules for each operator. Then, we were able to define the notion of bisimilarity of processes in Lean, which is used to prove all the equations that serve as axioms. Thus, these operators were implemented correctly in Lean. However, the hiding operator (see Section 2.1) introduces a lot of complexity, which needs to be formalized in future work. Due to this, we were unable to fully define the ABP. Most of the work has been done, but the hiding operator is crucial for finding the external behaviour of a process, and that is what is used to prove correctness of the process. Hence, we cannot prove correctness of the ABP as of yet. We did implement the rest of the ABP, including the basic actions and their communications, in a Lean-friendly way. The set of equations for the ABP, as well as the encapsulation to force communication were also implemented. What remains is implementing the hiding operator, and using this to linearize the process and show that the external behaviour is as expected.

The source code for this thesis can be found here.

In Chapter 2 we introduce the core concepts of this thesis: μCRL, the ABP, and the Lean Theorem Prover. Afterwards, in Chapter 3 the process of formalizing μCRL into Lean is discussed. Chapter 4 shows how we defined the ABP into Lean, and Chapter 5 covers our thoughts and progress on completing the formalization. Finally, Chapter 6 covers some related work, and 7 is a short conclusionary section.

2

Preliminaries

First we informally define the three important concepts in this thesis.

2.1 μCRL

The μCRL language is an algebra focused on analysing concurrent processes. For a more formal definition please refer to (10) or (9). The main elements of the language are a set of atomic actions and the operators alternative composition $(x + y)$, sequential composition $(x \cdot y)$ and parallel execution $(x \,||\, y)$. The choice of actions is up to the user. For example, an action may send or receive data, perform arithmetic operations or raise and lower a platform. The parallel execution is defined using a notion of communication of these atoms, where it is defined beforehand for every two atoms whether or not they communicate and what atom the result of communication is.

Furthermore, there are also the operators of encapsulation $(\partial_H(x))$, summation $(\Sigma_{d \in D} f(d))$ and hiding $(\tau_I(x))$, which are used to further refine processes.

A semantics of these operators is as follows:

- Atoms are predefined operations such as sending data or updating values.
- Alternative composition $(x + y)$ allows nondeterministic execution of either x or y.
- Sequential composition $(x \cdot y)$ first executes x, then executes y. We often leave out the $\cdot$ and write xy for brevity.
- Parallel execution $(x \,||\, y)$ can execute either of x or y without losing the other option, or if x and y communicate, can execute the communication action and then terminate. Note that the definition of $||$ uses two other operators, $\lfloor\!\lfloor$ and $|$. These are more limited

operators that are combined to form the parallel operator. Left parallel execution $(x \between y)$ works like the parallel execution, but its first action has to be from x. The communication operator $(x \mid y)$ executes the communication action for x and y if they communicate, and deadlocks otherwise.

- Encapsulation $(\partial_H(x))$, given a set H of atomic actions, blocks execution of any of the actions in H by x. This is mainly used to enforce communication actions, by blocking execution of the non-communicating alternatives.

- Summation $(\Sigma_{d \in D} f(d))$, given a possibly infinite set D of data, executes $f(d_1) + f(d_2) + f(d_3) + \ldots$, for all $d_1, d_2, d_3, \ldots \in D$.

- Hiding $(\tau_I(x))$, given a set I of actions, "abstracts" away the actions in I. This is used to hide internal actions, to hone in on the external behaviour, if there is an intended behaviour. It works by replacing any action from I with τ, the hidden action. This τ can be executed, but shows no behaviour.

- Finally, processes can terminate successfully or deadlock. We denote successful termination with $\checkmark$, and deadlock by δ. In both states no further actions can be taken, but termination is intended, whereas we wish to avoid deadlock.

All these operators are subject to two different types of semantics. First there are the transition rules, which allow the user to derive an execution trace from a μCRL process. Similarl to other derivation rules of operational semantics, premises can be used to draw a conclusion. For example, an alternative composition transition rule states that $x \xrightarrow{a} x'$ can be used to infer $x + y \xrightarrow{a} x'$ for all other processes y. All of the relevant μCRL transition rules are depicted below, ordered by which operator they refer to:

- There is a single transition rule for atoms, which states that an atom can execute the action it refers to, then terminate.

$$\frac{}{a \xrightarrow{a} \checkmark}$$

- The alternative composition operator has four rules, allowing execution of either the left or the right process.

$$\frac{x \xrightarrow{a} x'}{x + y \xrightarrow{a} x'} \quad \frac{y \xrightarrow{a} y'}{x + y \xrightarrow{a} y'} \quad \frac{y \xrightarrow{a} \checkmark}{x + y \xrightarrow{a} \checkmark} \quad \frac{x \xrightarrow{a} \checkmark}{x + y \xrightarrow{a} \checkmark}$$

- The sequential composition operator has two rules:

$$\frac{x \xrightarrow{a} x'}{x \cdot y \xrightarrow{a} x' \cdot y} \quad \frac{x \xrightarrow{a} \checkmark}{x \cdot y \xrightarrow{a} y}$$

- All the rules for parallelism are depicted below. Here, whenever c is present it is assumed that $a \mid b = c$.

$$\frac{y \xrightarrow{a} y'}{x \,||\, y \xrightarrow{a} x \,||\, y'} \quad \frac{x \xrightarrow{a} x'}{x \,||\, y \xrightarrow{a} x' \,||\, y} \quad \frac{y \xrightarrow{a} \checkmark}{x \,||\, y \xrightarrow{a} x} \quad \frac{x \xrightarrow{a} \checkmark}{x \,||\, y \xrightarrow{a} y}$$

$$\frac{x \xrightarrow{a} x' \quad y \xrightarrow{b} y'}{x \,||\, y \xrightarrow{c} x' \,||\, y'} \quad \frac{x \xrightarrow{a} \checkmark \quad y \xrightarrow{b} y'}{x \,||\, y \xrightarrow{c} y'} \quad \frac{x \xrightarrow{a} x' \quad y \xrightarrow{b} \checkmark}{x \,||\, y \xrightarrow{c} x'} \quad \frac{x \xrightarrow{a} \checkmark \quad y \xrightarrow{b} \checkmark}{x \,||\, y \xrightarrow{c} \checkmark}$$

$$\frac{x \xrightarrow{a} x'}{x \between y \xrightarrow{a} x' \,||\, y} \quad \frac{x \xrightarrow{a} \checkmark}{x \between y \xrightarrow{a} y}$$

$$\frac{x \xrightarrow{a} x' \quad y \xrightarrow{b} y'}{x \mid y \xrightarrow{c} x' \,||\, y'} \quad \frac{x \xrightarrow{a} \checkmark \quad y \xrightarrow{b} y'}{x \mid y \xrightarrow{c} y'} \quad \frac{x \xrightarrow{a} x' \quad y \xrightarrow{b} \checkmark}{x \mid y \xrightarrow{c} x'} \quad \frac{x \xrightarrow{a} \checkmark \quad y \xrightarrow{b} \checkmark}{x \mid y \xrightarrow{c} \checkmark}$$

- The rules for the hiding operator:

$$\frac{x \xrightarrow{a} x'}{\tau_I(x) \xrightarrow{a} x'} \quad \frac{x \xrightarrow{a} \checkmark}{\tau_I(x) \xrightarrow{a} \checkmark} \text{ if } a \notin I$$

$$\frac{x \xrightarrow{a} x'}{\tau_I(x) \xrightarrow{\tau} x'} \quad \frac{x \xrightarrow{a} \checkmark}{\tau_I(x) \xrightarrow{\tau} \checkmark} \text{ if } a \in I$$

- The rules for the encapsulation operator:

$$\frac{x \xrightarrow{a} x'}{\partial_H(x) \xrightarrow{a} x'} \quad \frac{x \xrightarrow{a} \checkmark}{\partial_H(x) \xrightarrow{a} \checkmark} \text{ if } a \notin H$$

- Finally, the rules for the summation operator:

$$\frac{f(d_i) \xrightarrow{a} f'}{\Sigma_{d \in D} f(d) \xrightarrow{a} f'} \quad \frac{f(d_i) \xrightarrow{a} \checkmark}{\Sigma_{d \in D} f(d) \xrightarrow{a} \checkmark} \text{ for } d_i \in D$$

There are also the axioms of μCRL, which define an equivalence relation over μCRL processes. These consist of equations. An example of these is the axiom $(x + y) \cdot z = x \cdot z + y \cdot z$, left distributivity of the alternative composition operator over the sequential operator. These axioms can be found in Figures 2.1 and 2.2, where a and b are atomic actions, and x, y and z are arbitrary μCRL processes. The axioms in Figure 2.1 are the ones we formalize in this thesis. The ones in 2.2 are left to future work.

$$
\begin{aligned}
x + y &= y + x \\
x + (y + z) &= (x + y) + z \\
x + x &= x \\
(x + y) \cdot z &= x \cdot z + y \cdot z \\
(x \cdot y) \cdot z &= x \cdot (y \cdot z) \\
x \,||\, y &= x \between y + y \between x + x \mid y \\
a \between x &= a \cdot x \\
a \cdot x \between y &= a \cdot (x \,||\, y) \\
(x + y) \between z &= x \between z + y \between z \\
a \mid b &= c \text{ if } a \text{ and } b \text{ communicate to } c \\
a \mid b &= \delta \text{ if } a \text{ and } b \text{ do not communicate} \\
a \cdot x \mid b &= (a \mid b) \cdot x \\
a \mid b \cdot x &= (a \mid b) \cdot x \\
a \cdot x \mid b \cdot y &= (a \mid b) \cdot (x \,||\, y) \\
(x + y) \mid z &= x \mid z + y \mid z \\
x \mid (y + z) &= x \mid y + x \mid z
\end{aligned}
\qquad
\begin{aligned}
x + \delta &= \delta \\
\delta \cdot x &= \delta \\
\partial_H(\delta) &= \delta \\
\partial_H(a) &= a \text{ if } a \notin H \\
\partial_H(a) &= \delta \text{ if } a \in H \\
\partial_H(x + y) &= \partial_H(x) + \partial_H(y) \\
\partial_H(x \cdot y) &= \partial_H(x) \cdot \partial_H(y) \\
\delta \between x &= \delta \\
\delta \mid x &= \delta \\
x \mid \delta &= \delta
\end{aligned}
$$

$$
\begin{aligned}
\Sigma_{d \in D} x &= x \\
\Sigma_{d \in D} P(d) &= \Sigma_{d \in D} P(d) + P(d_0) \text{ for } d_0 \in D \\
\Sigma_{d \in D}(P(d) + Q(d)) &= \Sigma_{d \in D} P(d) + \Sigma_{d \in D} Q(d) \\
(\Sigma_{d \in D} P(d)) \cdot x &= \Sigma_{d \in D}(P(d) \cdot x) \\
(\Sigma_{d \in D} P(d)) \between x &= \Sigma_{d \in D}(P(d) \between x) \\
(\Sigma_{d \in D} P(d)) \mid x &= \Sigma_{d \in D}(P(d) \mid x) \\
x \mid (\Sigma_{d \in D} P(d)) &= \Sigma_{d \in D}(x \mid P(d)) \\
\partial_H(\Sigma_{d \in D} P(d)) &= \Sigma_{d \in D}(\partial_H(P(d))) \\
(\forall e \in D, P(e) = Q(e)) &\implies \Sigma_{d \in D} P(d) = \Sigma_{d \in D} Q(d)
\end{aligned}
$$

Figure 2.1: μCRL axioms for all the operators formalized.

$$
\begin{aligned}
x \cdot \tau &= x \\
x \cdot (\tau \cdot (y + z) + y) &= x \cdot (y + z) \\
\tau_I(\delta) &= \delta \\
\tau_I(a) &= a \text{ if } a \notin I \\
\tau_I(a) &= \tau \text{ if } a \in I \\
\tau_I(x + y) &= \tau_I(x) + \tau_I(y) \\
\tau_I(x \cdot y) &= \tau_I(x) \cdot \tau_I(y) \\
\tau_I(\Sigma_{d \in D} P(D)) &= \Sigma_{d \in D}(\tau_I(P(d)))
\end{aligned}
$$

Figure 2.2: μCRL axioms for the hiding operator and the hidden action.

The transition rules and the schema of axioms should be compatible, in the sense that the axioms should be derivable from the transition rules. As we will see, for most of them this is easily true, but others are more difficult to show.

Example 2.1.1. Consider the actions a and b, and define a μCRL process $a \cdot b + b \cdot a$. Its state space is pictured in Figure 2.3. The starting state two outgoing transitions, one a-transition to the state b, and one b-transition the the state a. These correspond to the nondeterministic choice given by the process. In one case, we choose the left process, in the other, the right. The top left transition, $ab \xrightarrow{a} b$, can be derived using $a \xrightarrow{a} \checkmark$ and the transition rules for the sequential composition operator.

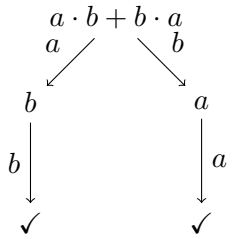

Figure 2.3: The execution trace of the process $a \cdot b + b \cdot a$.

2.1.1 Bisimilarity

Two μCRL processes are said to be bisimilar if they can execute the same trace of actions, and have the same branching structure in their execution trace. This can be formalized as follows:

Definition 2.1.1. A symmetric binary relation on μCRL processes R is called a *bisimulation* if the following holds:

- If Rvw and v has a transition $v \xrightarrow{a} v'$, then there is a w' such that $w \xrightarrow{a} w'$ and $Rv'w'$.
- If Rvw and v has terminated, then w must also have terminated.

Given this definition we have:

Definition 2.1.2. Two μCRL processes x and y are *bisimilar* if there is a bisimulation R such that Rxy holds. In that case, R is the bisimilarity relation for x and y.

The following examples show the concept of bisimulation in more detail.

Example 2.1.2. Consider the two processes $a(b+c)$ and $ab+ac$. They are not bisimilar, because for the second process, the choice between b and c is lost after the first transition. See Figure 2.4 for a representation of their state spaces. If we look at the states after the first transition, note how, in order for the two processes to be bisimilar, the state on the left would need to be bisimilar to both states on the right. This means that the states on the right would need to have both a b and a c transition. As this is not the case, these processes cannot be bisimilar.

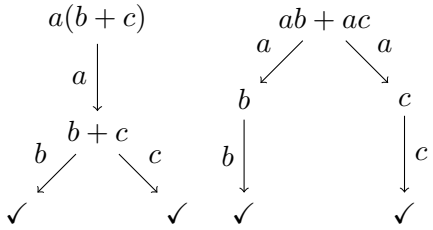

Figure 2.4: The processes $a(b+c)$ and $ab+ac$ are not bisimilar.

Example 2.1.3. Consider the two processes $(a+b)b$ and $ab+bb$. They are bisimilar because they execute the same actions and they never lose possible actions during execution. See Figure 2.5 for a representation of the state spaces and the bisimilarity relation linking them.

Standard bisimulation is not enough to capture the complexity of the hiding operator and the hidden action. It would require hidden actions to be equivalent as well, even though they are hidden, and therefore supposed to show no behaviour. We would like to have a way for processes to be equivalent, even with non-matching amounts of hidden actions. This is why we introduce branching bisimulation.

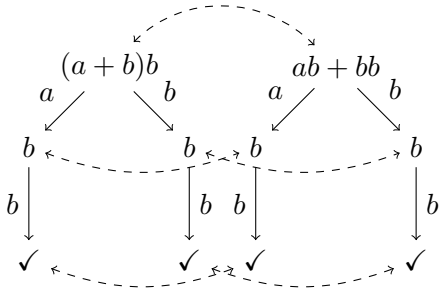

Figure 2.5: The processes $(a+b)(b+a)$ and $ab+bb+ba+aa$ are bisimilar.

Definition 2.1.3. A symmetric binary relation R is a *branching bisimulation* if the following holds:

- If Rvw and $v \xrightarrow{a} v'$, then either $a = \tau$ and $Rv'w$, or there is a sequence of (zero or more) τ-transitions from w to a point w' such that Rvw', $w' \xrightarrow{a} z$ and $Rv'z$.
- If Rvw and v has terminated, then there is a sequence of (zero or more) τ-transitions from w to a point w' such that Rvw' and w' has terminated.

Note that the second condition implies that a terminated process can be related to non-terminated processes.

This definition is better, but it still has a problem: it is not a congruence. For example, a and $\tau \cdot a$ are branching bisimilar, but $a+b$ and $\tau \cdot a + b$ are not branching bisimilar. To create a congruence, another definition is required: rooted branching bisimulation, which requires that the first actions executed are not the hidden action.

Definition 2.1.4. A symmetric binary relation R is a *rooted branching bisimulation* if the following holds:

- If Rvw and $v \xrightarrow{a} v'$, then there is a w' with $w \xrightarrow{a} w'$ and v' and w' are branching bisimilar.
- If Rvw and v has terminated, then w must also have terminated.

Note that once again terminated processes can only be bisimilar to terminated processes.

Some small examples follow:

- The processes a and $a \cdot \tau$ are rooted branching bisimilar and branching bisimilar.
- The processes $\tau \cdot (b+a) + \tau \cdot (a+b)$ and $a+b$ are branching bisimilar but not rooted branching bisimilar.

- The processes $c \cdot (\tau \cdot (b+a) + \tau \cdot (a+b))$ and $c \cdot (a+b)$ are rooted branching bisimilar but not bisimilar.

Rooted branching bisimilarity is the equivalence relation that bridges the gap between transition rules and the axioms: processes that are equivalent according to the axioms are also rooted branching bisimilar, and vice versa.

2.2 Alternating Bit Protocol

The Alternating Bit Protocol (ABP) was defined by Bartlett, Scantlebury and Wilkinsonin in (4) to help guarantee delivery of messages in communication protocols, specifically over phone lines. Afterwards it began being used as a small process that is shown to be correct, useful in case studies. It consists of four components and a set of channels between those components, as pictured in Figure 2.6. The A and D channels are called the external channels, and B, C, E and F are the internal channels.

A quick overview is as follows: The sender receives data from the outside, and wishes to send this to the receiver. The receiver then sends it on. However, messages in the internal channels can be corrupted, which does not prevent delivery, but invalidates the data. To still guarantee delivery, the sender adds a bit to the data, and alternates this after each successful message. The receiver also sends back acknowledgments for each message received, with the bit attached. In case of a corrupted or wrong bit acknowledgment, the sender resends the data.

In μCRL, the ABP is represented as as a set of recursive equations that expand out into the correct atoms. They are then combined into a starting process, and encapsulated to force communication and abstract away the internal actions.

Definition 2.2.1. The Alternating Bit Protocol uses the following set of recursive equations, where b is a bit, and d is an element of the data we wish to communicate:

$$\begin{aligned}
S(b) &= \Sigma_d r_A(d) \cdot s_B(d,b) \cdot T(d,b) \\
T(d,b) &= r_F(b) \cdot S(1-b) \\
&\quad + (r_F(1-b) + r_F(\bot)) \cdot s_B(d,b) \cdot T(d,b) \\
R(b) &= \Sigma_d r_C(d,b) \cdot s_D(d) \cdot s_E(b) \cdot R(1-b) \\
&\quad + \Sigma_d (r_C(d,1-b) + r_C(\bot)) \cdot s_E(1-b) \cdot R(b) \\
K &= \Sigma_d \Sigma_b r_B(d,b) \cdot (j \cdot s_C(d,b) + j \cdot s_C(\bot)) \cdot K \\
L &= \Sigma_b r_E(b) \cdot (j \cdot s_F(b) + j \cdot s_F(\bot)) \cdot L
\end{aligned}$$

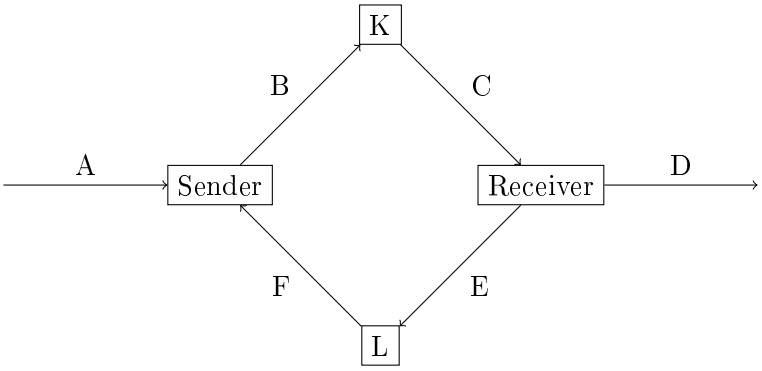

Figure 2.6: An overview of the Alternating Bit Protocol.

Think of S and T as governing the sender, R as governing the receiver, and K and L as handling the corruption of messages. These equations are then combined into the final ABP process $(S(0) \,||\, R(0) \,||\, K \,||\, L)$. In order to show the correct behaviour, it is then further refined by first encapsulating with the set $H = \{s_I(x), r_I(x) \mid I \text{ is internal}\}$ to force communication, and then abstraction with the set $I = \{c_I(x) \mid I \text{ is internal}\}$ to abstract away the internal actions.

2.3 Lean

For a complete overview of the Lean Theorem Prover, see (2) or (8). Lean is an open source interactive theorem prover, based on a dependent type theory with inductive families. In Lean, we can define inductive types, and reason on lemmas and possibly inductive propositions about them. Using these methods we can formalise mathematical concepts and state and prove theorems about various subjects.

For example, an inductive type is created by the use of the `inductive` keyword, followed by a name for the type and `: Type`, to show we are defining a type. Then we can define the constructors of that type: functions and constants. This is done by using the `| name : type` syntax. This allows us to define our constructors and then specify the type they should have, be that a constant (just one type) or a function (two or more types connected by $\rightarrow$ operators).

After defining an inductive type, functions and predicates can be defined about it. This can also be done inductively, but does not have to be. To avoid inconsistencies, Lean requires recursive definitions to be decreasing. This means that there is a metric that decreases with each recursive call of the definition. Lean can usually prove automatically that a function decreases, but it sometimes needs some help. Predicates can also be

defined inductively, however the syntax differs a little from defining types. Instead of defining constructors, we need to specify rules for when the predicate is made true. These can have various premises. Finally, lemmas can be made about functions and predicates, and subsequently proved.

A major part of the proof structure of Lean is the use of tactics and tactics mode. This is a backwards-style proof, working on the goal to try and prove it equal to one or more assumptions. Various tactics exist, such as `intro`, `cases`, and `apply`.

Of particular note is the `simp` tactic, which attempts to rewrite the goal according to various pre-defined lemmas. It can also be given lemmas to use via the `simp [lemma_1, lemma_2, ...]` construction.

To end this section, let us show a simple example of a Lean formalization. The following is an inductive definition of the natural numbers in Lean. Note the two standard constructors, and how they are written.

```
inductive natural : Type
| zero : natural
| succ : natural → natural
```

Now we can open the `natural` namespace, to avoid having to write `natural.zero` and `natural.succ`, and then we can define the `even` inductive predicate.

```
namespace natural

inductive even : natural → Prop
| zero : even zero
| add_two (k) : even k → even (succ (succ k))
```

As mentioned above, an inductive predicate is defined by specifying rules for when it is true. In this case, `even` is true either when the argument is zero, or when the argument is an even number plus 2.

We can prove lemmas about the even predicate:

```
lemma not_even_one :
¬even (succ (zero)) :=
begin
  intro h,
  cases h
end

lemma even_four :
```

```
even (succ (succ (succ (succ zero)))) :=
begin
  apply even.add_two,
  apply even.add_two,
  exact even.zero
end

end natural
```

In this case we showed that one is not even, and four is even. In the `one` case you can see that we assume that one is even (`intro h`), and then we can appeal to Lean using `cases`. This will refer to the definition of `even`, and because none of the cases apply, Lean knows that this situation is impossible, which proves the lemma.

The second lemma uses the inductive predicate's rules to prove the goal.

3

Translating μCRL into Lean

Bezem, Bol and Groote embed μCRL in the proof assistant Coq by postulating μCRL's datatypes and axioms as extra axioms in Coq's logic (7). Our goal is similar. However, in Lean we make an attempt to embed μCRL even more into the language, compared to what is done in the paper. We create a model of μCRL, by defining the syntax of μCRL and then creating a quotient type that includes all the axioms of the language. Our aim is to add operators, starting with just the atom, alternative composition and sequential composition operators. First we discuss two different approaches, and which one we eventually use. Then we describe the implementation of the basic language using this approach, and how we create a quotient on it. Finally we will discuss how we added all the other required operators, which will give us the required quotient at the end. As a first step, below we depict the main μCRL syntax. For the first sections, only the `atom`, `seq`, and `alt` operators are relevant.

```
inductive  mcrl2  (α : Type) : Type 1
| atom : α → mcrl2
| seq : mcrl2 → mcrl2 → mcrl2
| alt : mcrl2 → mcrl2 → mcrl2
| par : mcrl2 → mcrl2 → mcrl2
| parl : mcrl2 → mcrl2 → mcrl2
| comm : mcrl2 → mcrl2 → mcrl2
| encap : set α → mcrl2 → mcrl2
| sum {β : Type} : set β → (β → mcrl2) → mcrl2
| deadlock : mcrl2
```

We will come back to the definition of `sum` in Section 3.4.

3.1 Inductive or Bisimulation

Our first attempt used an inductive type to define all the axioms explicitly. This type is shown below:

```
inductive mcrl2_equiv {α : Type} : mcrl2 α → mcrl2 α → Prop
| alt_comm {x y} : mcrl2_equiv (x + y) (y + x)
| alt_ideml {x} : mcrl2_equiv (x + x) x
| alt_idemr {x} : mcrl2_equiv x (x + x)
| alt_assocl {x y z} : mcrl2_equiv (x + (y + z)) ((x + y) + z)
| alt_assocr {x y z} : mcrl2_equiv ((x + y) + z) (x + (y + z))
| seq_rdistl {x y z} : mcrl2_equiv ((x + y) · z) (x · z + y · z)
| seq_rdistr {x y z} : mcrl2_equiv (x · z + y · z) ((x + y) · z)
| seq_assocl {x y z} : mcrl2_equiv (x · (y · z)) ((x · y) · z)
| seq_assocr {x y z} : mcrl2_equiv ((x · y) · z) (x · (y · z))
```

This worked well for the basic version of the language, though we needed to add some extra axioms in order to make it a proper equivalence relations. These axioms, in the form of congruence rules and a transitivity rule, are shown below.

```
inductive mcrl2_equiv {α : Type} : mcrl2 α → mcrl2 α → Prop
...
| atom {a} : mcrl2_equiv (atom a) (atom a)
| alt {x y x' y'}
(h₁ : mcrl2_equiv x x') (h₂ : mcrl2_equiv y y') :
mcrl2_equiv (x + y) (x' + y')
| seq {x y x' y'}
(h₁ : mcrl2_equiv x x') (h₂ : mcrl2_equiv y y') :
mcrl2_equiv (x · y) (x' · y')
| trans {x y z}
(h₁ : mcrl2_equiv x y) (h₂ : mcrl2_equiv y z) :
mcrl2_equiv x z
```

After getting started on adding support for the parallel operator, we noticed that the definition would become bloated, as we would need to define 3 axioms for the new operators as well as add 11 new axioms. Hence, we decided to try a different way.

Our new idea is to define transition rules for each μCRL operator first, and then use these transitions to define a notion of bisimilarity. This bisimilarity is then used as the relation for the quotient.

3.2 Transitions

Once again, starting out with just the basic μCRL operators (atoms, alternative composition and sequential composition), their rules for transitions were defined. We wish for any μCRL process to have two options: it can continue, with transitions to other processes, or it can deadlock, if it has no transitions to take. A μCRL process should also be able to terminate, after which no more transitions should be taken. So we would want a μCRL transition to be represented by an inductive predicate that should take a process, an action and a term that can either be termination or another process. Termination was a bit of an obstacle at first: if we made it a part of the μCRL type, then we would have to handle cases such as $x \cdot \checkmark$ or $\checkmark \,||\, y$, presumably by explicitly excluding these cases in the rules. These are not valid μCRL processes, but would be definable in the Lean model. Hence, we decided to model termination differently. It is modelled by the `option` type, where `none` means termination, and `some x` means a next process. So, a transition is a term of type `mcrl2` $\alpha \to \alpha \to$ `option (mcrl2` α`)` $\to$ `Prop`. This way we can have processes terminate, without having to make sure termination never appears on the left side of a transition. The initial set of transitions is shown below.

```
inductive transition : mcrl2 α → α → option (mcrl2 α) → Prop
| atom {a} : transition (atom a) a none
| altl {x y z a} (h : transition x a z) :
transition (x + y) a z
| altr {x y z a} (h : transition y a z) :
transition (x + y) a z
| seq {x y z a} (h : transition x a z) :
transition (x · y) a (seq' z y)
```

Here the `seq'` operator is used to identify the two cases for x terminating or not terminating. Its definition is as follows:

```
def seq' : option (mcrl2 α) → mcrl2 α → option (mcrl2 α)
| none y := y
| (some x) y := x · y
```

Using this definition of transitions, we translated the first set of μCRL axioms to lemmas and then proved them. This was done in part by creating specific other lemmas that can adapt these definitions to Lean logic. We call these types of lemmas iff-lemmas. They use Lean's logic to prove the axioms more easily. See below for an iff-lemma that lifts the alternative composition to logical formulas.

```
lemma transition.alt_iff (x y : mcrl2 α) (a z) :
transition (x + y) a z ↔
transition x a z ∨ transition y a z :=
begin
  split,
  { rintro (h | h),
    { left, assumption },
    { right, assumption }},
  { rintro (h | h),
    { exact transition.altl h },
    { exact transition.altr h }}
end
```

Below the difference between using iff-lemmas and not using any is shown. Note the difference in length of the proof, and how we have to do every step one at a time. The proof using iff-lemmas, on the other hand, is one line, and is done wholly using Lean's logic. This can be seen by the additional usage of lemmas about $\wedge$, $\vee$, and $\exists$.

```
lemma transition.alt_dist (x y z : mcrl2 α)
(x' : option (mcrl2 α)) (a) :
transition ((x + y) · z) a x' ↔
transition ((x · z) + (y · z)) a x' :=
begin
  simp only [transition.alt_iff,
             transition.seq_iff,
             and_or_distrib_left,
             exists_or_distrib]
end

lemma transition.alt_dist₂ (x y z : mcrl2 α)
(x' : option (mcrl2 α)) (a) :
transition ((x + y) · z) a x' ↔
transition ((x· z) + (y · z)) a x' :=
begin
  split,
  { intro h,
    cases h,
    cases h_h,
    { apply transition.altl,
```

```
      apply transition.seq,
      assumption},
    { apply transition.altr,
      apply transition.seq,
      assumption}},
  { intro h,
    cases h,
    { cases h_h,
      apply transition.seq,
      apply transition.altl,
      assumption},
    { cases h_h,
      apply transition.seq,
      apply transition.altr,
      assumption}}
end
```

The only lemma that was not provable in this way was the associativity of the sequential composition operator, due to the definition of the sequential transition rule. However, this axiom does hold, but needs bisimulation.

Example 3.2.1. Consider the two μCRL processes $x = a \cdot (b \cdot (c \cdot d))$ and $y = ((a \cdot b) \cdot c) \cdot d$. They are bisimilar, but cannot be considered equal by the transition rules. We can easily see that x has one a-transition to the process $b \cdot (c \cdot d)$ and y has one a-transition to the process $(b \cdot c) \cdot d$. These processes are not definitionally equal, and hence the sequential associativity axiom cannot be shown using 1 step transition lemmas.

3.3 Towards a Quotient

Recall from Definition 2.1.1 that a symmetric relation is a basic bisimulation if it fulfils two conditions. First there is the forward rule, which states that for any two related processes x and y, if x has an a-transition to a process x', then y must have an a-transition to a process y' such that x' and y' are related via the bisimulation. Secondly, there is a rule for termination. If a process that has terminated is related to another process, then that process must also have terminated.

In Lean, these rules can be combined via the use of the `option.rel` relation. This is a relation on the `option` monad that extends a relation on the base type by relating `none` with `none`.

```
inductive option.rel (r : α → β → Prop) : option α → option β → Prop
/-- If ‘a ~ b‘, then ‘some a ~ some b‘ -/
| some {a b} : r a b → rel (some a) (some b)
/-- ‘none ~ none‘ -/
| none       : rel none none
```

As the second rule of bisimulations implies that termination is only related to itself, and termination is modelled in Lean as `none`, this is sufficient. Then, we only need to make sure that the forward rule and symmetry hold.

We wish to view bisimilarity itself as a relation, so we want to define `bisim x y` for two μCRL processes `x` and `y`. This is done by relating `x` and `y` if there is a bisimulation `R` such that `R x y` holds. See below for our definition of bisimilarity.

```
def is_bisimulation (R : mcrl2 α → mcrl2 α → Prop) : Prop :=
  (∀x y : mcrl2 α, (∀x' a, R x y → transition x a x' →
   ∃y', transition y a y' ∧ option.rel R x' y')) ∧ symmetric R

def bisim (α : Type) : mcrl2 α → mcrl2 α → Prop
| x y := ∃R : mcrl2 α → mcrl2 α → Prop,
         (R x y) ∧ is_bisimulation R
```

To define the process algebra as a quotient modulo bisimulation, we have to prove various theorems. Of note are of course reflexivity, symmetry and transitivity, but there are also congruence rules for the operators.

```
lemma bisim.reflexive :
∀x, bisim x x

lemma bisim.symmetric :
∀x y, bisim x y → bisim y x

lemma bisim.transitive :
∀x y z, bisim x y → bisim y z → bisim x z

lemma bisim.alt :
∀x y x' y', bisim x y → bisim x' y' → bisim (x + x') (y + y')

lemma bisim.seq :
∀x y x' y', bisim x y → bisim x' y' → bisim (x · x') (y · y')
```

Looking at our definition of bisimulation, we see that we need to supply a relation satisfying the conditions of bisimilarity. For example, to show transitivity a specific definition of the composition of relations is used (see below). Then, we prove that this composition, given two bisimulations, is a bisimulation. That suffices to show transitivity of bisimilarity. Notice how the definition of `R_comp` includes a symmetric rule.

```
inductive R_comp (R₁ R₂ : mcrl2 α → mcrl2 α → Prop) :
mcrl2 α → mcrl2 α → Prop
| stepl {a b} (h : ∃c, R₁ a c ∧ R₂ c b) : R_comp a b
| stepr {a b} (h : ∃c, R₁ a c ∧ R₂ c b) : R_comp b a
```

After defining relations for each of the above rules and using them to prove the lemmas, we defined the quotient and the class structure for our base μCRL class.

As mentioned before, many of the axioms can be proven using the transition rules directly. This is lifted to the bisimulation by the use of a relation `R_add` and a lemma `R_add_congr` which use the lemma for the axiom in transition form to prove the bisimilarity.

```
inductive R_add {x y : mcrl2 α} :
mcrl2 α → mcrl2 α → Prop
| basel : R_add x y
| baser : R_add y x
| refl {a} : R_add a a

lemma R_add_congr {x y : mcrl2 α}
(h : ∀z a, (transition x a z) ↔ (transition y a z)) :
bisim x y :=
begin
  apply exists.intro R_add,
  apply and.intro R_add.basel,
  apply and.intro,
  { intros x₁ y₁ x₁' a h₁ h₂,
    apply exists.intro x₁',
    cases h₁,
    { rw ←h,
      apply and.intro h₂,
      exact option.rel.refl R_add_refl},
    { rw h,
      apply and.intro h₂,
      exact option.rel.refl R_add_refl},
```

```
    { apply and.intro h₂,
      exact option.rel.refl R_add_refl}},
  { exact R_add.symm}
end
```

This allows us to prove many lemmas in one line. For example:

```
lemma mcrl2.alt_dist {x y z : mcrl2 α} :
(x + y) · z ≈ x · z + y · z :=
  by exact R_add_congr (transition.alt_dist x y z)
```

As mentioned in Section 3.2, we need to prove the associativity of the sequential composition operator using bisimulation. This necessitates defining a relation, which is defined below:

```
inductive R_seq_assoc {x y z: mcrl2 α} :
mcrl2 α → mcrl2 α → Prop
| basel : R_seq_assoc (x · y · z)   (x · (y · z))
| baser : R_seq_assoc (x · (y · z)) (x · y · z)
| stepl {x a x'}
  (h_l : R_seq_assoc (x · y · z) (x · (y · z)))
  (h_r : R_seq_assoc (x · (y · z)) (x · y · z))
  (h₂ : transition x a (some x')) :
  R_seq_assoc (x' · y · z) (x' · (y · z))
| stepr {x a x'}
  (h_l : R_seq_assoc (x · y · z) (x · (y · z)))
  (h_r : R_seq_assoc (x · (y · z)) (x · y · z) )
  (h₂ : transition x a (some x')) :
  R_seq_assoc (x' · (y · z)) (x' · y · z)
| refl {x} : R_seq_assoc x x
```

Note how similar the step rules are to the process in Example 3.2.1. With this relation, proving the rule is straightforward, though still lengthy. A part of this proof is presented below. Note that the four cases after `cases h₁` all have the same structure as the case that has been left in, and how in this part, we use three of the rules defined in `R_seq_assoc`.

```
lemma mcrl2.seq_assoc {x y z : mcrl2 α} : (x · y) · z ≈ x · (y · z) :=
begin
  apply exists.intro R_seq_assoc,
  apply and.intro,
  exact R_seq_assoc.basel,
```

```
  apply and.intro,
  { intros x₁ y₁ x₁' a₁ h₁ h₂,
    cases h₁,
    {...},
    {...},
    { cases h₂,
      simp only [transition.seq_iff, ←exists_and_distrib_right,
        and_assoc, exists_comm, exists_eq_left],
      apply exists.intro h₂_z,
      apply and.intro,
      assumption,
      cases h₂_z,
      { apply option.rel.some,
        exact R_seq_assoc.refl},
      { apply option.rel.some,
        apply R_seq_assoc.stepr,
        apply R_seq_assoc.stepl,
        exact h₁_h_l,
        repeat {assumption}}},
    {...},
  {...}
end
```

To finish this section, we picture a snippet of the base instance for μCRL:

```
instance mcrl2'.base : mcrl2_base α (mcrl2' α) := {
  atom := λa, ⟦mcrl2.atom a⟧,
  alt := quotient.lift₂ (λa b : mcrl2 α, ⟦a + b⟧)
           begin
           intros a₁ a₂ b₁ b₂ h₁ h₂,
           simp,
           exact bisim.alt _ _ _ _ h₁ h₂
           end,
  seq := quotient.lift₂ (λa b, ⟦a · b⟧)
           begin
           intros a b a' b' ha hb,
           simp,
           exact bisim.seq ha hb
           end,
  (..),
```

```
  alt_dist := begin
    intros x y z,
    apply quot.induction_on₃ x y z,
    intros a b c,
    apply quotient.sound,
    exact mcrl2.alt_dist
  end,
  seq_assoc := begin
    intros x y z,
    apply quot.induction_on₃ x y z,
    intros a b c,
    apply quotient.sound,
    exact mcrl2.seq_assoc,
  (..)
}
```

We have defined the operators on the quotient using the base operators and the `quotient.lift` operator, which can lift a function `mcrl2` $\alpha \to$ `mcrl2` $\alpha \to$ `mcrl2` α to the quotient `mcrl2'` $\alpha \to$ `mcrl2'` $\alpha \to$ `mcrl2'` α. Then the axioms are proved, using quotient induction. This similarly reduces the proof to the basic case, allowing us to use the lemmas to complete the proof.

3.4 Adding operators

After defining the base μCRL class, the other operators were introduced one by one, starting with the parallel operators. For simplicity, we first attempted to only include the main parallel operator ||, but as the axioms for the others are too important this was decided against. As opposed to defining a separate communication function for actions, we decided to view it as part of the actions in the form of a sort of multiplication. This allows us to view the type of actions as a specific sort of semigroup: a commutative semigroup with a zero element. In the transition rules, we can then use $a * b \neq 0$ if we want to assume that a and b communicate. Note that this does mean that we need to make sure that 0 cannot have any transitions, as it would be a valid atom. To make sure that this does not happen, we have added a condition that a is not equal to 0 to our base atom transition rule. Let us picture these added and changed rules below:

```
inductive transition : mcrl2 α → α → option (mcrl2 α) → Prop
| atom {a} (h : a ≠ 0) : transition (atom a) a none
```

```
(...)
| par_l {x x' y a} (h : transition x a x') :
  transition (x || y) a (par' x' y)
| par_r {x y y' a} (h : transition y a y') :
  transition (x || y) a (par' x y')
| par_comm {x y x' y' a b}
  (h₁ : transition x a x') (h₂ : transition y b y')
  (h₃ : a * b ≠ 0) :
  transition (x || y) (a * b) (par' x' y')
| parl {x x' y a} (h : transition x a x') :
  transition (x |_ y) a (par' x' y)
| comm {x y x' y' a b}  (h₁ : transition x a x')
  (h₂ : transition y b y') (h₃ : a * b ≠ 0) :
  transition (x | y) (a * b) (par' x' y')
```

Note the $a \neq 0$ in the `atom` rule. Like the `seq'` operator from Section 3.2, the `par'` operator is used to handle all cases for x and y in one rule.

```
def par' : option (mcrl2 α) → option (mcrl2 α) → option (mcrl2 α)
| none none := none
| (some x) none := x
| none (some y) := y
| (some x) (some y) := x || y
```

The encapsulation operator is defined as taking a set α and a process. This coincides with the usual definition, and the transition rule for it is shown below.

```
inductive transition : mcrl2 α → α → option (mcrl2 α) → Prop
(...)
| encap_pass {x y A} {a : α} (h₁ : a ∉ A)
  (h₂: transition x a y) :
  transition (encap A x) a (encap A <$> y)
```

The <$> operator functions as the `map` function on `option`. It is used to identify the cases for x terminating and not terminating.

Defining the quotient for the parallel and encapsulation operators was similar to the base case. Before continuing to the summation operator, we encountered two curiosities while implementing these operators. Firstly, showing that $x \,||\, y$ is equivalent to $x \between y + y \between x + x \mid y$ ran into the same issue as the associativity of $\cdot$ from Section 3.2. Secondly, now is when the deadlock constant was added. This was added alongside the parallel operator, as the

process needs to deadlock if a communication between two atoms that do not communicate is attempted. In the basic version, no processes can deadlock, so it was not needed at first.

Our last operator, the summation operator $\Sigma_{d \in D} f(d)$, was at first defined with type `set` $\alpha \to (\alpha \to$ `mcrl2` $\alpha) \to$ `mcrl2` α, as a sum over the actions from α. This was later noticed to be too limiting, as the intent is to allow it to sum over arguments of actions, as opposed to just actions. For example, in the ABP we can write

$$\Sigma_b r_E(b) \cdot (j \cdot s_F(b) + j \cdot s_F(\bot)) \cdot L,$$

where b is a boolean value. Our current definition cannot define this, as booleans are not actions on their own. We would have to define a set like $\{r_E(tt), r_E(ff)\}$ and similar, and then use some if-statements to get to the correct combinations. This would not do, so we changed the definition. The new definition was $\forall \beta$, `set` $\beta \to (\beta \to$ `mcrl2` $\alpha) \to$ `mcrl2` α. This was more lenient, but expressive enough for our purposes. In order to avoid allowing `set (mcrl2` α`)` in this operator, and invoking Girard's paradox, we needed to make `mcrl2 : Type 1`. This did not have any other impact on our process, however. Now we can write the above definition in Lean.

```
sum Bool {tt, ff} (λb, r_E(b) · (j · s_F(b) + j · s_F(⊥)) · L)
```

Because the type of `sum` quantifies over an undefined type, defining the quotient can not be done as easily as the other operators. There is no operator that can lift $\forall \beta$, `set` $\beta \to (\beta \to$ `mcrl2` $\alpha) \to$ `mcrl2` α to $\forall \beta$, `set` $\beta \to (\beta \to$ `mcrl2'` $\alpha) \to$ `mcrl2'` α. Hence, it required the use of the axiom of choice, in order to get a representative of the quotient class to use. This does make our instance `noncomputable`, but so far this has not led to any issues.

```
noncomputable instance mcrl2'.sum : mcrl2_sum α (mcrl2' α) := {
  sum := (λβ D f, quotient.mk (sum D (λa, quot.out (f a))))
  (...)
}
```

In essence, we took a function f of type $\beta \to$ `mcrl2'` α, and used the axiom of choice to get a representative `f b` for all $b \in \beta$.

In addition, for all the other operators we used quotient induction to prove the axioms on the quotient, but for the sum operator we could not do that, due to the use of the axiom of choice. Instead, we reason directly on the equivalence classes using congruence lemmas and transitivity of the equivalence relation.

Example 3.4.1. We aim to prove the axiom of $(\Sigma_{d \in D} P(d)) \cdot x = \Sigma_{d \in D}(P(d) \cdot x)$ for the newly defined quotient. In the snippet below, we can see relational reasoning on the quotient using transitivity and symmetry.

```
begin
  intros β D f x,
  simp [quotient.mk_eq_iff_out, quotient.eq_mk_iff_out],
  apply setoid.trans,
  { apply mcrl2'.quot_seq},
  { apply setoid.trans,
    { apply bisim.seq,
      { exact quotient.mk_out _},
      { exact setoid.refl _}},
    { apply setoid.trans,
      { apply mcrl2'.sum_seq},
      { apply setoid.symm,
        apply bisim.sum,
        intros a h,
        exact mcrl2'.quot_seq}}}
end
```

One of the sum axioms has a small difference compared to the theoretical definition: The rule of $\Sigma_{d \in D} x = x$ (where x does not depend on d) needs an extra requirement that the set D is nonempty. Otherwise we would be unable to apply the `sum` rule and so no transitions could be taken.

The final set of transition rules is depicted in Figure 3.1. Note how it is about the same amount of rules as the inductive definition from Section 3.1, while covering all the operators. We were able to prove all the axioms from Figure 2.1. Most of these proofs were straightforward, but some required an appeal to the bisimilarity structure. Figure 3.2 shows this, by marking the axioms proved using bisimilarity with a $*$ symbol over their equality.

```
inductive transition :
  mcrl2 α → α → option (mcrl2 α) → Prop
| atom {a} (h : a ≠ 0) : transition (atom a) a none
| altl {x y z a} (h : transition x a z) :
  transition (x + y) a z
| altr {x y z a} (h : transition y a z) :
  transition (x + y) a z
| sum {β : Type} {f : β → mcrl2 α} {a' z a A}
  (ha' : a' ∈ A) (h : transition (f a') a z) :
  transition (sum A f) a z
| seq {x y z a} (h : transition x a z) :
  transition (x · y) a (seq' z y)
| encap_pass {x y A} {a : α} (h₁ : a ∉ A)
  (h₂: transition x a y) :
  transition (encap A x) a (encap A <$> y)
| par_l {x x' y a} (h : transition x a x') :
  transition (x || y) a (par' x' y)
| par_r {x y y' a} (h : transition y a y') :
  transition (x || y) a (par' x y')
| par_comm {x y x' y' a b}
  (h₁ : transition x a x') (h₂ : transition y b y')
  (h₃ : a * b ≠ 0) :
  transition (x || y) (a * b) (par' x' y')
| parl {x x' y a} (h : transition x a x') :
  transition (x |_ y) a (par' x' y)
| comm {x y x' y' a b}  (h₁ : transition x a x')
  (h₂ : transition y b y') (h₃ : a * b ≠ 0) :
  transition (x | y) (a * b) (par' x' y')
```

Figure 3.1: The final definition of the transition rules.

$$
\begin{aligned}
x + y &= y + x \\
x + (y + z) &= (x + y) + z \\
x + x &= x \\
(x + y) \cdot z &= x \cdot z + y \cdot z \\
(x \cdot y) \cdot z &\overset{*}{=} x \cdot (y \cdot z) \\
x \,||\, y &\overset{*}{=} x \between y + y \between x + x \mid y \\
a \between x &= a \cdot x \\
a \cdot x \between y &\overset{*}{=} a \cdot (x \,||\, y) - \\
(x + y) \between z &= x \between z + y \between z \\
a \mid b &= c \text{ if } a \text{ and } b \text{ communicate to } c \\
a \mid b &= \delta \text{ if } a \text{ and } b \text{ do not communicate} \\
a \cdot x \mid b &= (a \mid b) \cdot x \\
a \mid b \cdot x &= (a \mid b) \cdot x \\
a \cdot x \mid b \cdot y &= (a \mid b) \cdot (x \,||\, y) \\
(x + y) \mid z &= x \mid z + y \mid z \\
x \mid (y + z) &= x \mid y + x \mid z
\end{aligned}
\qquad
\begin{aligned}
x + \delta &= \delta \\
\delta \cdot x &= \delta \\
\partial_H(\delta) &= \delta \\
\partial_H(a) &= a \text{ if } a \notin H \\
\partial_H(a) &= \delta \text{ if } a \in H \\
\partial_H(x + y) &= \partial_H(x) + \partial_H(y) \\
\partial_H(x \cdot y) &\overset{*}{=} \partial_H(x) \cdot \partial_H(y) \\
\delta \between x &= \delta \\
\delta \mid x &= \delta \\
x \mid \delta &= \delta
\end{aligned}
$$

$$
\begin{aligned}
\Sigma_{d \in D} x &= x \\
\Sigma_{d \in D} P(d) &= \Sigma_{d \in D} P(d) + P(d_0) \text{ for } d_0 \in D \\
\Sigma_{d \in D} (P(d) + Q(d)) &= \Sigma_{d \in D} P(d) + \Sigma_{d \in D} Q(d) \\
(\Sigma_{d \in D} P(d)) \cdot x &= \Sigma_{d \in D} (P(d) \cdot x) \\
(\Sigma_{d \in D} P(d)) \between x &= \Sigma_{d \in D} (P(d) \between x) \\
(\Sigma_{d \in D} P(d)) \mid x &= \Sigma_{d \in D} (P(d) \mid x) \\
x \mid (\Sigma_{d \in D} P(d)) &= \Sigma_{d \in D} (x \mid P(d)) \\
\partial_H(\Sigma_{d \in D} P(d)) &= \Sigma_{d \in D} (\partial_H(P(d))) \\
(\forall e \in D, P(e) = Q(e)) &\implies \Sigma_{d \in D} P(d) \overset{*}{=} \Sigma_{d \in D} Q(d)
\end{aligned}
$$

Figure 3.2: μCRL axioms for all the operators formalized. The axioms that were proved using bisimulation are marked with a $*$ symbol over their equality.

4

Translating the ABP

Now that our model for μCRL in Lean is expressive enough (missing only the hiding operator), we will attempt to formalize the Alternating Bit Protocol (ABP) from Section 2.2, to test it. As we need to define a type α of atomic actions for μCRL in Lean, a large part of defining the ABP will be defining this set of actions that can be taken. Hence, this chapter will be split in two sections: one to cover the set of actions, and one to cover the actual definition of the ABP.

4.1 The Actions

The set of actions needs to have a multiplication that is associative and commutative. It also needs to include a zero element, and the "j" placeholder action. Of course, we need to model the send, receive and communicate actions for each channel. These actions take arguments that denote data that gets communicated. Note that not every channel takes the same arguments. For example, channel A only takes an element from an abstract datatype as its argument, while channel C takes two arguments: the data and an acknowledgment bit. Our first implementation gave each set of channels their own semigroup, which we then combined into the larger set. This proved cumbersome, in part because we needed to define multiple zero elements, one for each subsemigroup, as well as the one for the main set. The final implementation uses a single semigroup, that gives the actions an argument of a special type `Args`, that distinguishes the types of channels. See below for this implementation. A convenient part of this, is how simple the definition of the multiplication becomes; the definition of the `mul` operator only requires four lines.

```
@[derive decidable_eq]
inductive Int_Channels : Type
```

```
| B : Int_Channels
| C : Int_Channels

@[derive decidable_eq]
inductive Ack_Channels : Type
| E : Ack_Channels
| F : Ack_Channels

@[derive decidable_eq]
inductive Ext_Channels : Type
| A : Ext_Channels
| D : Ext_Channels

@[derive decidable_eq]
inductive Args (α : Type) [decidable_eq α] : Type
| Int : Int_Channels → option (α × bool) → Args
| Ext : Ext_Channels → α → Args
| Ack : Ack_Channels → option bool → Args

inductive Act (α : Type) [decidable_eq α] : Type
| zero : Act
| j : Act
| r : Args α → Act
| s : Args α → Act
| c : Args α → Act

def mul : Act α → Act α → Act α
| (r x) (s y) := if x = y then c x else zero
| (s x) (r y) := if x = y then c x else zero
| _ _ := zero
```

Note how we handle the corruption of messages in the definition of `Args`: using the `option` type. For the internal channels this means we need to use the tuple type as well, as `option (α → bool)` does not have the correct behaviour when used as an argument. This means that `c I none` is a corrupted message in both cases.

Also note how we allow send actions for the incoming channel and receive actions for the outgoing channel. These will not be used, but make the definition simpler. Because we define the actions used in the equations ourselves, this will not influence correctness of

the ABP.

4.2 The Definition

As the definition of the ABP is mutually recursive, an intuitive definition of the ABP in Lean would also be recursive. A snippet of our first definition is below, though note how we have introduced the shorthand `Ext.recv I d := mcrl2.atom (Act.r (Args.Ext I d))` and similar to help with readability:

```
mutual def S, T, R, K, L, Fin
with S : bool → mcrl2 (Act α)
| b := mcrl2.sum D (λd, (Ext.recv A d) ·
                        (Int.send B d b) ·
                        (T d b n))
with T : α → bool → mcrl2 (Act α)
| d b  := (Ack.send F b) · (S (not b)) +
       ((Ack.recv F (bnot b)) + (Ack.recv F none)) ·
       (Int.send B d b) · (T d b n)
with Fin : mcrl2.mcrl2 (Act α)
|  := mcrl2.encap H (S ff  || R ff || K || L)
```

There is an axiom in μCRL that defines a solution for every recursive equation. The formalization by Bezem and Bol (7) takes this axiom, and postulates it as true in Coq. However, with μCRL as an inductive type, this axiom creates an inconsistency, as seen in Pierce (13, Chapter 21). This inconsistency rests in the nature of elements of the inductive type as finite terms. Postulating a solution of an infinite recursive process will assume that there is a finite process that is larger than every process, which is a contradiction. Hence, we will not do the same, but instead will create a partial implementation. We will give the definition an extra natural number, which limits the number of times it can be unfolded. This can be seen in our inductive definition of the variables for the ABP below, where each element has an argument `n : nat`.

```
inductive Step (α : Type) : Type
| S (b : bool) (n : nat) : Step
| T (d : α) (b : bool) (n : nat) : Step
| R (b : bool) (n : nat) : Step
| K (n : nat) : Step
| L (n : nat) : Step
| Fin (n : nat) : Step
```

Furthermore, Lean's default procedure for proving that a definition is decreasing is not sufficient for our purposes. We need to redefine the `sizeof` function for this new type to only take this new parameter into account. Otherwise, Lean would use the parameters of type α and `bool` for the invocations of `S` and `R` and be unable to prove that the definition of the `Fin` operator is decreasing.

```
def Step.sizeof' : Step α → nat
| (Step.S _ n) := n
| (Step.T _ _ n) := n
| (Step.R _ n) := n
| (Step.K n) := n
| (Step.L n) := n
| (Step.Fin n) := n
```

The main step to defining the ABP is to give the interpretation for the `Step` type. This is done in a new definition, which we will call `ABP : Step α → mcrl2 (Act α)`. This definition needs to be recursive on the n argument of the operators, and so we need base and inductive cases for them all. Figure 4.1 shows our definition. The interpretation of `S` is below:

```
def ABP (α : Type) [decidable_eq α]  (D : set α) : Step α → mcrl2 (Act α)
| (Step.S b 0) :=
  mcrl2.sum D (λd, (Ext.recv A d) ·
                   (Int.send B d b))
| (Step.S b (nat.succ n)) :=
  mcrl2.sum D (λd, (Ext.recv A d) ·
                   (Int.send B d b) ·
                   (ABP (Step.T d b n)))
```

Note the two cases, and the two actions: they are $r_A(d)$ and $s_B(d, b)$ respectively, as in the definition of `S` in Definition 2.2.1. The inductive case additionally has a recursive call to `T` that follows the definition. This is left out in the base case in order to allow the execution to terminate.

Finally, to finish the definition we define a function that will start the ABP given a set of input data and a natural number:

```
inductive H : set (Act α)
| r {x} (h : ∀I d, x ≠ Args.Ext I d) : H (Act.r x)
| s {x} (h : ∀I d, x ≠ Args.Ext I d) : H (Act.s x)
```

```
def ABP_True {β : Type} [decidable_eq β] : set β → nat → mcrl2 (Act β)
| X n := mcrl2.encap H (ABP β X (Step.Fin n))
```

After defining a type of data β, and giving this function a set X of that type and a natural number n, it will execute n actions of the ABP, transferring data from X.

```
def ABP (α : Type) [decidable_eq α]  (D : set α) : Step α → mcrl2 (Act α)
| (Step.S b 0) := mcrl2.sum D (λd, (Ext.recv A d) ·
                          (Int.send B d b))
| (Step.S b (nat.succ n)) :=
  mcrl2.sum D (λd, (Ext.recv A d) ·
              (Int.send B d b) ·
              (ABP (Step.T d b n)))
| (Step.T d b 0) := (Ack.send F b) +
               ((Ack.recv F (bnot b)) + (Ack.recv F none)) ·
               (Int.send B d b)
| (Step.T d b (nat.succ n)) :=
  (Ack.send F b) · (ABP (Step.S (not b) n)) +
  ((Ack.recv F (bnot b)) + (Ack.recv F none)) ·
  (Int.send B d b) · (ABP (Step.T d b n))
| (Step.R b 0) := (mcrl2.sum D (λd, (Int.recv C d b) ·
             (Ext.send Ext_Channels.D d))) +
  (mcrl2.sum D (λd, ((Int.recv C d (bnot b)) + (Int.recv_fail C)) ·
               (Ack.send E (bnot b))))
| (Step.R b (nat.succ n)) :=
  (mcrl2.sum D (λd, (Int.recv C d b) ·
               (Ext.send Ext_Channels.D d) · ABP (Step.R (bnot b) n))) +
  (mcrl2.sum D (λd, ((Int.recv C d (bnot b)) + (Int.recv_fail C)) ·
               (Ack.send E (bnot b)) · ABP (Step.R b n)))
| (Step.K 0) :=
  mcrl2.sum D (λd,
  mcrl2.sum {tt, ff} (λb, (Int.recv B d b) ·
                  (Act.fill · (Int.send C d b) +
                   Act.fill · (Int.send_fail C))))
| (Step.K (nat.succ n)) :=
  mcrl2.sum D (λd,
  mcrl2.sum {tt, ff} (λb, (Int.recv B d b) ·
                  (Act.fill · (Int.send C d b) +
                   Act.fill · (Int.send_fail C)))) · ABP (Step.K n)
| (Step.L 0) :=
  mcrl2.sum {↑tt, ↑ff} (λb, (Ack.recv E b) ·
                    (Act.fill · (Ack.send F b) +
                     Act.fill · (Ack.send F none)))
| (Step.L (nat.succ n)) :=
  mcrl2.sum {↑tt, ↑ff} (λb, (Ack.recv E b) ·
                    (Act.fill · (Ack.send F b) +
                     Act.fill · (Ack.send F none))) · (ABP (Step.L n))
| (Step.Fin 0) := mcrl2.deadlock
| (Step.Fin (nat.succ n)) :=
  (ABP (Step.S ff n) || ABP (Step.R ff n) || ABP (Step.K n) || ABP (Step.L n)
```

Figure 4.1: Our definition of the ABP.

5

Towards a Proof of Correctness

While we were unable to complete the formalization of μCRL and the ABP, we have made steps toward formalizing the hiding operator, and we would like to discuss these here. Afterwards, some comments on the possibilities of formalizing the ABP conclude this section.

5.1 The Hiding Operator

Defining the hiding operator required us to extend our semigroup definition for α to include the hidden action. This included adding some properties as well, because we wish for the hidden action to not communicate and not be the result of any communications. We also need the hidden action and the 0 to be distinct, though we can show this using the `tau_act` property.

```
class comm_semigroup_with_zero_and_tau (S₀ : Type*) extends
    semigroup_with_zero S₀, comm_semigroup S₀ :=
(tau : S₀)
(tau_mul : ∀a : S₀, tau * a = 0)
(mul_tau : ∀a : S₀, a * tau = 0)
(tau_act : ∀a b : S₀, a * b ≠ tau)

lemma tau_ne_zero (α : Type) [comm_semigroup_with_zero_and_tau α] :
(tau : α ) ≠ (0 : α) :=
begin
  intro h,
  apply _inst_1.tau_act,
  rw h,
```

```
  apply mul_zero,
  use 0,
end
```

Adding the operator's constructors and defining the transition rules for it were not too much trouble, as the hidden action process is a constant, and the hiding operator's syntax is familiar, similar to the encapsulation operator.

```
inductive mcrl2 (α : Type) : Type 1
(...)
| tau : mcrl2
| abstract : set α → mcrl2 → mcrl2
```

We were then able to define iff-lemmas for both of the operators:

```
lemma transition.tau_iff (a : α) (z) :
transition mcrl2.tau a z ↔ (a = tau ∧ z = none)

lemma transition.abs_iff (I : set α) (x a z) :
transition (abstract I x) a z ↔
  (((a = tau) ∧ (∃a', (a' ∈ I ) ∧ ∃z', z = abstract I <$> z'
                                ∧ transition x a' z')) ∨
  (a ∉ I ∧ ∃z', z = abstract I <$> z' ∧ transition x a z'))
```

The iff-lemmas allowed us to prove most of the axioms for the hiding operator outright. For example:

```
lemma transition.hide_alt (I : set α) (x y a z) :
transition (abstract I (x + y)) a z ↔ transition ((abstract I x) + (
    abstract I y)) a z :=
begin
  simp [transition.abs_iff, transition.alt_iff, and_or_distrib_left,
        exists_or_distrib, or_assoc, or.left_comm]
end
```

The point where it got complicated was in creating the quotient. As mentioned in Section 2, the hiding operator needs new types of bisimulation to become a congruence. These need to be reflected in Lean. The requirement of "having a sequence of tau actions" is reflected in a new inductive predicate, which denotes whether two processes have a sequence of hidden actions between them. As we do not consider all the processes in-between, this is sufficient for our definition.

```
inductive has_tau_sequence : option (mcrl2 α) → option (mcrl2 α) → Prop
| refl {x} : has_tau_sequence x x
| trans {x y z} (h : transition x tau y) (h' : has_tau_sequence y z) :
    has_tau_sequence x z
```

Now we can define the two types of bisimulation. For ease of reading, we have split up the cases for branching bisimulation, as below:

```
def none_branch (R : option (mcrl2 α) → option (mcrl2 α) → Prop) : Prop
:= ∀y : option (mcrl2 α), R none y →
  (has_tau_sequence y none ∧ R none none)

def some_branch (R : option (mcrl2 α) → option (mcrl2 α) → Prop) : Prop
:= ∀x : mcrl2 α, ∀ y x' a, R x y → transition x a x' →
((a = tau ∧ R x' y) ∨
 (∃z : mcrl2 α, ∃y', has_tau_sequence y z ∧ R x z ∧ transition z a y' ∧
  R x' y'))

def is_branching_bisimulation (R : option (mcrl2 α) → option (mcrl2 α) →
    Prop) : Prop
:= none_branch R ∧ some_branch R ∧ symmetric R

def b_bisim (α : Type) [comm_semigroup_with_zero_and_tau α] :
option (mcrl2 α) → option (mcrl2 α) → Prop
| x y := ∃R : option (mcrl2 α) → option (mcrl2 α) → Prop, (R x y) ∧
    is_branching_bisimulation R
```

Note how branching bisimulation requires a relation `R` on `option (mcrl2 α)`, as now it is possible for `none` to be related to processes other than itself. Rooted branching bisimulation, in contrast, is simpler.

```
def is_rb_bisimulation (R : mcrl2 α → mcrl2 α → Prop) : Prop :=
(∀x y : mcrl2 α, R x y → ∀x' a, transition x a x' →
  ∃y', transition y a y' ∧ b_bisim α x' y') ∧ symmetric R

def rb_bisim (α : Type) [comm_semigroup_with_zero_and_tau α] :
(mcrl2 α) → (mcrl2 α) → Prop
| x y := ∃R : (mcrl2 α → mcrl2 α → Prop), R x y ∧ is_rb_bisimulation R
```

Proving reflexivity and symmetry for both types of bisimulation was straightforward, but transitivity has proven more complicated. A paper by Basten we found late in our re-

search covers a way to show transitivity of the branching bisimulation (5). Unfortunately it requires another type of bisimulation, and links the new bisimulation to branching bisimulation in a way we have been unable to implement as of yet.

As a final part of this section, we can show the definition of the class we wish to create, and comment on the next steps.

```
class mcrl2_abs (α : Type) (M : Type 1)
[comm_semigroup_with_zero_and_tau α] extends mcrl2_sum α M :=
(tau : M)
(abs : set α → M → M)
(seq_tau : ∀x, seq x tau = x)
(tau_keep : ∀x y z, seq x (alt (seq tau (alt y z) ) y) = seq x (alt y z))
(hide_deadlock : ∀I, abs I deadlock = deadlock)
(hide_pass : ∀I a, a ∉ I → abs I (atom a) = atom a)
(hide_fail : ∀I a, a ∈ I → abs I (atom a) = tau)
(hide_alt : ∀I x y, abs I (alt x y) = alt (abs I x) (abs I y))
(hide_seq : ∀I x y, abs I (seq x y) = seq (abs I x) (abs I y))
(hide_sum: ∀I D : set α, ∀f, abs I (sum D f) = sum D (λa, abs I (f a)))
```

As it is now, `mcrl2'` cannot be an instance of this class, as the bisimilarity relation is too strict to allow some of those axioms to be proved. Instead, we have to define a new quotient that uses the rooted branching bisimulation as its equivalence relation.

```
@[instance] def setoid_bmcrl2 : setoid (mcrl2 α) :=
{r     := rb_bisim α,
iseqv :=
  begin
    repeat {apply and.intro},
    { apply rb_bisim_reflexive},
    { apply rb_bisim_symmetric},
    { apply rb_bisim_transitive}
  end
}

def bmcrl2' (α : Type) [comm_semigroup_with_zero_and_tau α] :=
quotient (@setoid_bmcrl2 α _)
```

What remains, is to prove that `bmcrl2'` is an instance of `mcrl2_abs`, which includes not only proving the axioms related to the hiding operator and hidden action, those in Figure 2.2, but also proving the axioms from Figure 2.1 again, for our new quotient. We imagine

that most of these proofs will be analogous to the case for basic bisimulation, but we have not put it into practice as of yet.

5.2 Future ABP

Our implementation of the ABP is not correct. This is in part because Lean does not allow possibly infinite programs, and in part because an axiom that would allow this sort of definition is inconsistent, so we changed our implementation to one that will eventually stop. In an ideal world, a future implementation without this constraint should be defined and proved correct. This will require a full formalization of the hiding operator, as well as of the process theory behind μCRL. In particular, the Cluster Fair Abstraction rule, used to remove loops of hidden actions, will be necessary.

If we could define μCRL using a coinductive type, it might be possible to create a definition of the ABP we can prove correct using greatest fixpoints.

6

Related Work

There are various other formalizations of μCRL and μCRL processes in interactive theorem provers. Most of these rest on one particular work, where μCRL is formalized into Coq. This paper, by Bezem and Bol, discusses their process, and formalizes the ABP into Coq as well (7). They implemented an embedding of μCRL, postulating their own types and keeping everything separate from the logic of Coq. They then formalized the later rules of μCRL, defined the ABP, and completed their paper by proving the correctness of their formalization. In contrast, we aimed to intuitively embed μCRL into Lean, to make use of its logic. Our formalization is not as complete as theirs, however. If we look at their implementation of actions, we can see that they explicitly separate the arguments of actions from their definition. By contrast, we leave that to the user, assuming that the definition of our type α is correct. They also mention the separation of sorts and the problem of allowing processes to be a sort. This is not a problem in Lean, as we have defined μCRL as a type universe higher than the type of actions.

Groote and Pol have also verified a μCRL protocol in Coq (11). However, most of the paper is spent verifying the protocol on paper, and only a small section is about the formalization. They take much of the groundwork from the paper by Bezem and Bol, and hence do not go into detail about their implementation. The authors mention some things they found lacking in Coq, which could be incorporated into our Lean formalization. Of note are mentions of metavariables as well as automatic definition unfolding, which are both Lean concepts. For example, they mention reasoning with transitivity. This is something that requires metavariables, for the in-between point, that Lean can do well. As for definition unfolding, Lean can do this automatically using the `simp` tactic. We imagine that these have been implemented in Coq as of now as well.

Finally for Coq formalizations, we have a formalization of Milner's scheduler in Coq by Korver and Springintveld (12). This paper rests on Bezem and Bol as well, but mentions that the computer verification found errors in a correctness proof on paper, which they were able to solve. They focus on the proof, not showing a lot of definitions. They reasoned a lot on the meta-level, which they found complicated to do in a theorem prover. In future work, they hope that much of this can be automated. We can see some of this automation in our work: the use of iff-lemmas is a type of automation, letting Lean's logic do a lot of the work.

Badban, Fokkink, Groote, Pang and Pol wrote another verification in μCRL, this time of the Sliding Window Protocol, in PVS (3). Their paper is also for a large part about the paper verification, with one section about the formal verification in PVS. They do not go into detail about their implementation of μCRL, only covering the most important parts of their proof, such as some later theorems. Part of this paper, such as the section on LPEs, could be applied to our formalization, as a next step.

Curiously, Stappers, Reniers, Webbe and Groote wrote μCRL itself as a μCRL specification (14). This is very comparable to what we have done, though it was more of an exercise on paper. Still, the idea of verifying μCRL is the same, and doing it in μCRL itself makes it even more interesting. They do not use the concept of bisimulation, but the way they describe the semantics of μCRL is very precise. In particular, they were able to describe the two kinds of actions defined in μCRL more clearly than we did: syntactic and semantic actions. In the process of formalizing μCRL in μCRL, they were able to find some inconsistencies in the μCRL transition rules, which have been resolved in our version of μCRL.

7

Conclusion

We have successfully formalized most of μCRL into Lean. All the core operators are there, and we have made a quotient type that follows the axioms of μCRL. The most interesting implementation was the sum operator, as it required use of more in-depth Lean concepts, as well as the axiom of choice. We have also begun defining the ABP in Lean, but with some caveats. What remains is completing the formalization of the hiding operator, including the quotient based on branching and rooted branching bisimilarity. As the final quotient should be based on rooted branching bisimilarity, it might have been a good idea to skip basic bisimilarity, and immediately start with the rooted branching version. We did not do this, and now we would need to redo a lot of proofs for rooted branching bisimulation. However, because basic bisimilarity is simpler, it did give us an opportunity to see how feasible formalizing μCRL in Lean is.

In order to finish formalizing the ABP, we would need to find a way to define its solution. We would need to formalize the process of linearization into Lean, and also find a way to adress the contradiction in the solution axiom. Linearization would require defining Linear Process Equations, and formalizing some important concepts related to finding solutions of equations, some of which require the presence of the hiding operator. To adress the contradiction, it would be prudent to see if a coinductive definition for μCRL would help.

If the μCRL formalization could be completed, this would mean that Lean is suitable for writing convenient μCRL correctness proofs. Then users could use Lean for formalizing μCRL specifications.

References

[1] Jeremy Avigad and John Harrison. Formally verified mathematics. *Commun. ACM*, 57(4):66–75, 2014. doi: 10.1145/2591012. URL https://doi.org/10.1145/2591012. 1

[2] Jeremy Avigad, Leonardo de Moura, and Soonho Kong. *Theorem Proving in Lean*. Carnegie Mellon University, 2014. URL https://leanprover.github.io/tutorial/tutorial.pdf. 1, 11

[3] Bahareh Badban, Wan J. Fokkink, Jan Friso Groote, Jun Pang, and Jaco van de Pol. Verification of a sliding window protocol in μCRL and PVS. *Formal Aspects Comput.*, 17(3):342–388, 2005. doi: 10.1007/s00165-005-0070-0. URL https://doi.org/10.1007/s00165-005-0070-0. 44

[4] Keith A. Bartlett, Roger A. Scantlebury, and Peter T. Wilkinson. A note on reliable full-duplex transmission over half-duplex links. *Commun. ACM*, 12(5):260–261, 1969. doi: 10.1145/362946.362970. URL https://doi.org/10.1145/362946.362970. 10

[5] Twan Basten. Branching bisimilarity is an equivalence indeed! *Inf. Process. Lett.*, 58 (3):141–147, 1996. doi: 10.1016/0020-0190(96)00034-8. URL https://doi.org/10.1016/0020-0190(96)00034-8. 40

[6] Jan A. Bergstra and Jan Willem Klop. Process algebra for synchronous communication. *Inf. Control.*, 60(1-3):109–137, 1984. doi: 10.1016/S0019-9958(84)80025-X. URL https://doi.org/10.1016/S0019-9958(84)80025-X. 1

[7] Marc Bezem, Roland N. Bol, and Jan Friso Groote. Formalizing process algebraic verifications in the calculus of constructions. *Formal Aspects Comput.*, 9(1):1–48, 1997. doi: 10.1007/BF01212523. URL https://doi.org/10.1007/BF01212523. 1, 15, 33, 43

[8] Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The lean theorem prover (system description). In Amy P. Felty and Aart Middeldorp, editors, *CADE-25*, volume 9195 of *Lecture Notes in Computer Science*, pages 378–388. Springer, 2015. doi: 10.1007/978-3-319-21401-6_26. URL https://doi.org/10.1007/978-3-319-21401-6_26. 1, 11

[9] Wan J. Fokkink. *Introduction to Process Algebra.* Texts in Theoretical Computer Science. An EATCS Series. Springer, 2000. ISBN 978-3-540-66579-3. doi: 10.1007/978-3-662-04293-9. URL https://doi.org/10.1007/978-3-662-04293-9. 1, 3

[10] Wan J. Fokkink. *Modelling Distributed Systems.* Texts in Theoretical Computer Science. An EATCS Series. Springer, 2007. ISBN 978-3-540-73937-1. doi: 10.1007/978-3-540-73938-8. URL https://doi.org/10.1007/978-3-540-73938-8. 1, 3

[11] Jan Friso Groote and Jaco van de Pol. A bounded retransmission protocol for large data packets. In Martin Wirsing and Maurice Nivat, editors, *AMAST '96*, volume 1101 of *LNCS*, pages 536–550. Springer, 1996. doi: 10.1007/BFb0014338. URL https://doi.org/10.1007/BFb0014338. 43

[12] Henri Korver and Jan Springintveld. A computer-checked verification of milner's scheduler. In Masami Hagiya and John C. Mitchell, editors, *TACS*, volume 789 of *LNCS*, pages 161–178. Springer, 1994. doi: 10.1007/3-540-57887-0_95. URL https://doi.org/10.1007/3-540-57887-0_95. 44

[13] Benjamin C. Pierce. *Types and programming languages.* MIT Press, 2002. ISBN 978-0-262-16209-8. 33

[14] Frank P. M. Stappers, Michel A. Reniers, Sven Weber, and Jan Friso Groote. Dogfooding the formal semantics of mCRL2. In Jonathan P. Bowen, Huibiao Zhu, and Mike Hinchey, editors, *35th Annual IEEE Software Engineering Workshop, SEW 2012,*, pages 90–99. IEEE Computer Society, 2012. doi: 10.1109/SEW.2012.16. URL https://doi.org/10.1109/SEW.2012.16. 44