Universidad Complutense de Madrid

Modeling and Verification of Concurrent Processes

Degree Final Project

Doble Grado en Ingeniería Informática y Matemáticas

Facultad de Informática y Facultad de Ciencias Matemáticas

July 2017

Author:

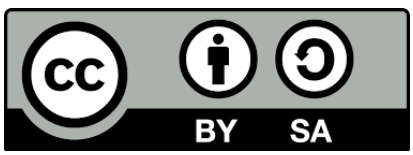
Hussein Hassan Harrirou

Supervised by professor:

David de Frutos Escrig

To my family and friends.
No deadlock was found in the
production of this work.

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.



Resumen

En este proyecto se pretende dar un fondo teórico básico, en forma de una sintaxis algebraica y una semántica denotacional basada en grafos, para el modelado de procesos concurrentes, con el propósito final de verificar propiedades sobre estos modelos. Para alcanzar esta meta, se ha elegido el lenguaje de modelado y herramienta de verificación $\mu CLR2$, desarrollada en la Technische Universiteit Eindhoven, y hemos seguido el libro "Modeling and analysis of communicating systems" de Jan Friso Groote (profesor en dicha universidad) y Mohammad Reza Mousavi (profesor en Högskolan i Halmstad). Primero definiremos la sintaxis usada para modelar los procesos y su contrapartida semantica usando Labeled Transition Systems. A continuación, explicaremos las diferentes equivalencias entre procesos y cuándo deben usarse. En la tercera parte, definiremos la sintaxis de las formulas lógicas para expresar las condiciones a verificar. Despues mostraremos las técnicas que se aplican a los LTS para verificar dichas fórmulas. Finalmente, presentamos dos casos prácticos de modelado y verificación, aplicados al protocolo Needham-Schroeder de clave pública, y al protocolo TCP.

Palabras clave: álgebra de procesos, bisimulación, concurrencia, verificación, mCRL2.

Abstract

In this project we aim to show a basic theoretical foundation, in the form of an algebraic syntax and a graph-based denotational semantic, for modeling concurrent processes, with the final purpose of verifying properties of these models. To achieve this goal, we have chosen the modeling language and verification tool $\mu CRL2$, developed in the Technische Universiteit Eindhoven, following the book "Modeling and analysis of communicating systems" by Jan Friso Groote (professor at that university) and Mohammad Reza Mousavi (professor at Högskolan i Halmstad). First, we will define the syntax used to model processes and their semantic counterpart as Labeled Transition Systems. Next, we will explain the different equivalences between processes, and when to use them. Thirdly, we will define the syntax of the formal logic used to express the conditions that will be verified. Later, we will showcase the techniques applied to LTS´s to verify those formulas. Finally, we will show two case studies corresponding to the Needham-Schroeder protocol and TCP protocol.

Keywords: process algebra, bisimulation, concurrency, verification, mCRL2.

Contents

1 Introduction

In this bachelor's thesis, we will give a basic introduction to concurrent systems, process algebra, process semantics and modal logics, in order to understand how things work and how to model and check concurrent systems with the mCRL2 tool.

1.1 Motivation

We, as humans, interact everyday with an immense quantity of devices of varied interface and functionality. Most of these systems have a well-defined way to interact with them, but those procedures are not universally defined, that is, they do not include all possibilities occurring in real life, but a small subset of "ideal" uses. Because of this, many times these systems fail, not because they are badly designed, but because they were underdesigned or perhaps the environment that surround them changed unexpectedly. In order to prevent some of these failures, we would have to check all possible states of the surrounding world that could affect the behavior of the system, but that is highly impractical. The solution that mathematicians and computer scientists have adopted is that of formally modeling these systems, taking into consideration some of those possible sudden risks, and then checking the desired properties of those models. This could sound very abstract at first, but for sure we expect that the plane we are flying in is safe from any kind of serious failures, such as errors in communications with the control tower or a crash on the piloting software. To put this into perspective, in early Pentium processors, one mathematician at Intel found a bug in the floating-point division instruction once millions of them had already been built, which costed the company $475 million to solve. This terrible spending could have been saved previously by a computer-assisted checking of an accurate model of the processor. In the case of IBM's cryptographic interface, the model checking did happen, showing an explicit way to bypass security and breach password encryption. With the increasing use of the Internet and the independency and interdependency of various systems around us (namely the Internet of Things), it is more neccesary than ever to model and check, to prevent undesired consequences.

1.2 Concurrent Systems

Imagine we have a coffee machine in the office. The minimum required operations a coffee machine needs to perform are two: take money and give coffee. To simplify things more, we can consider that once the coffee has been served, the machine returns to the initial state, asking for cash again. We will write this as an equation:

$$X = money.coffee.X$$

Let us add now a new functionality of the machine, serving tea. Then, one approach would produce the equation:

$$X = money.coffee.X + money.tea.X$$

while an alternative approach could lead us to:

$$X = money.(coffee.X + tea.X)$$

While "distributivity" from arithmetics would make us expect that the two equations are equivalent, that is not the case here, and the two systems would behave very differently. In fact, in the first case we do not have a choice, once we insert the coin into the machine. Instead, in the second one we can actually choose between coffee and tea as we would expect. The previous example simply uses sequential systems. Moving on to concurrent systems, we could have ten machines connected together so that the customers are served on a single output terminal: What would happen if two machines were giving a drink at the same time? Would they collide on the conveyor belt or would one of the machines wait for the other? In this last case, which one should be the one to wait? All these questions relate to the way two or more systems (beverage vending machines in this case) interact with each other. When systems grow in size and complexity, modeling using formal languages and letting a computer check the safety of the obtained system is the way to proceed. These machines, how they behave and their interactions, compose what we will call processes. Each process performs interactions with other processes in a parallel way. To study them, we have to properly define the way they behave and describe them with a precise language. To accomplish this we define process algebras.

2 Describing Processes

In this first chapter, we formally define the actions that compose processes, and we present labeled transition systems (LTS) as a method to represent the behavior of processes. Finally, we introduce operators in process algebra that we will use to textually describe sequential and parallel processes. We also give a way to transform this algebraic representation to an equivalent LTS, i.e. the operational semantics.

2.1 Actions and LTSs

2.1.1 Actions

First we introduce the concept of action, which intuitively is the atom of processes.

Definition 1 (Action)**.** An action is the smallest observable event in a process. These actions may be parameterized including data to express specific interactions with precise information. We denote the set of possible actions by Act.

For example, we can consider the behavior of a simple computer represented by actions *power_ on*, *work*, and *power_ off*. We can also capture the actions of a keyboard by *press(a)*, *release(w)*, using the key names as parameters.

It will be useful to define the concepts of multi-actions and timed actions.

Definition 2 (Multi-action)**.** Let $a, b \in Act$. The multi-action $a|b \in Act$ represents that both actions are performed at the (exact) same time.

Definition 3 (Timed action)**.** Let $a \in Act$, $t \in \mathbb{R}$. The timed action $a^c t$ denotes that the action a is performed at time t.

2.1.2 Labeled Transition Systems

The definitions and use of LTSs in this work follow those we usually find in the literature. However, we will include them here for the sake of self-containment.

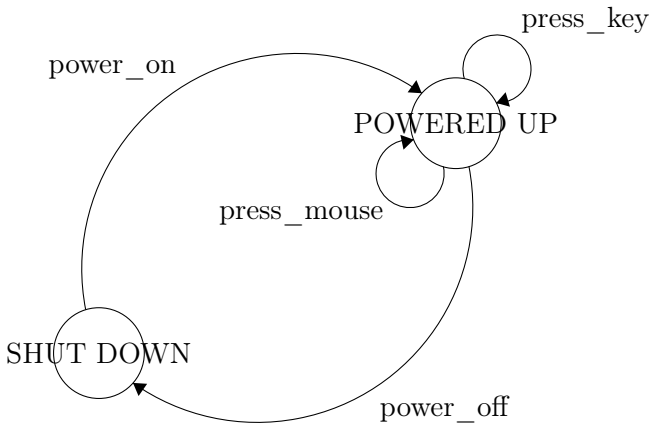
Definition 4 (Labeled Transition System)**.** A labeled transition system (LTS) is a tuple $(S, Act, \rightarrow, s_0, S_T)$, where:

- S is a set of states,
- Act is a set of actions,
- $(\rightarrow) \subset S \times Act \times S$ is the transition relation,

- $s_0 \in S$ is the initial state,
- $S_T \subset S$ is the set of terminating states.

When drawn, terminal states of labeled transition systems are marked with the ✓symbol. We will enforce that terminating states must not have outward transitions. This decision makes arriving at a terminating state a definitive act.

In the previous example, we talked about a computer with *power_on* and *power_off* actions. Using LTS notation to describe such a basic model of a computer, the obtained diagram would be the following:



2.2 A Process Algebra

We next introduce our process algebra. Because for the moment we will only give a syntactic definition, we will have to rely on a set of equations that conform our denotational semantics. These rules will not be given here and instead we will give the transformation from each operator and construct of process algebra to its equivalent LTS structure.

Throughout this section we will use P to denote the set of processes.

2.2.1 Sequential Processes

Next we see the basic ways of composing processes.

Definition 5 (Sequential composition)**.** Let $p, q \in P$. The sequential composition of p and q, $p \cdot q$, is a process that runs the process p and after finishing it runs the process q. The operator is usually omitted if there is no contextual ambiguity, writing the composition as pq.

Given the LTS denotation of p and q, and supposing that p_F is the set of final states in the LTS of p, and q_0 is the initial state of the LTS of q, then their sequential composition will be represented by the LTS that "fuses" each element of p_F with q_0 thus introducing overlapped "copies" of the transitions of q_0 after every state of p_F.

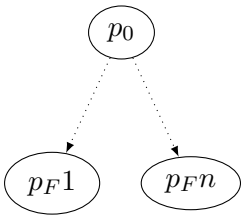


Figure 1: LTS of P

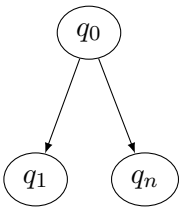


Figure 2: LTS of Q

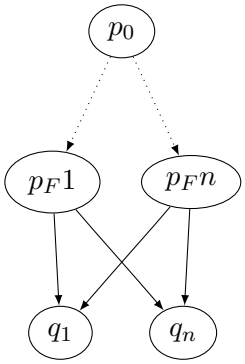


Figure 3: LTS of $P \cdot Q$

Definition 6 (Alternative composition)**.** Let $p, q \in P$. The alternative composition of p and q, $p + q$, is a process that either runs p or q. This operator is also called the choice operator.

The choice of processes p and q is translated to an LTS where the initial states p_0 and q_0 are fused, creating a new initial state which is the source of the transitions of the previous initial states. However, p_0 and q_0 are also preserved, as the initial choice is only made once, but we must preserve the possible loops of each individual process.

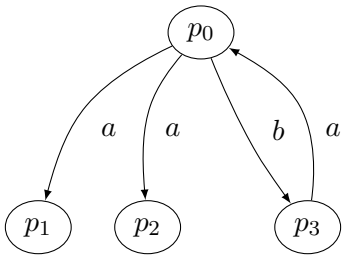


Figure 4: LTS of P

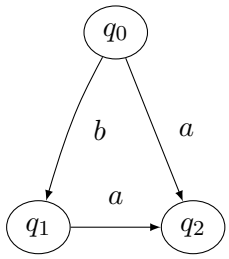


Figure 5: LTS of Q

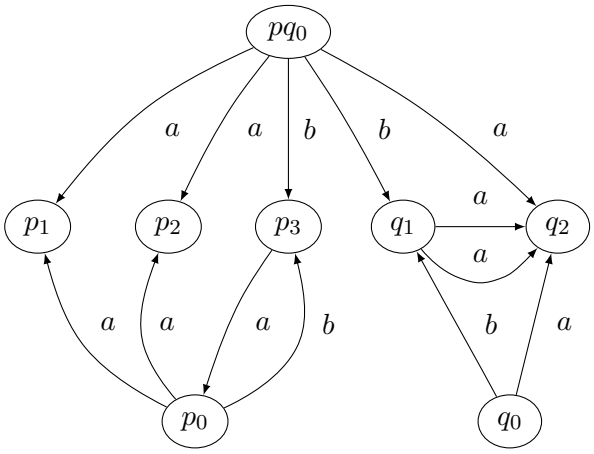


Figure 6: LTS of P+Q

Definition 7 (*Deadlock*)**.** The process *deadlock*, denoted by σ, is a process that cannot do anything at all, including that it cannot terminate, either.

Deadlock is not usually used as a "desired" component of a process, but rather as the state a process reaches when any uncontrollable error happens or when we arrive to the typical deadlock situation where a miscommunication between several components is produced. In fact, the translation of the *deadlock* process is the LTS with a single non-terminating state without transitions.

Figure 7: LTS of δ

Definition 8 (Conditional operator)**.** Let $c \in \mathbb{B}$, $p, q \in P$. The conditional operator $c \rightarrow p \diamond q$ behaves as p if c is true, or as q, otherwise.

Conditional operators are especially useful for specifying different behaviors of parameterized processes that depend on the value of the parameters. By using this operator, one can describe general processes that capture a single abstract behavior, even if the concrete behavior depends on a certain conditions on their parameters.

As an example, we can consider the *factorial(n)* process, which calculates $n!$. Using the common recursive definition, we need a base case, which will be implemented by a conditional operator.

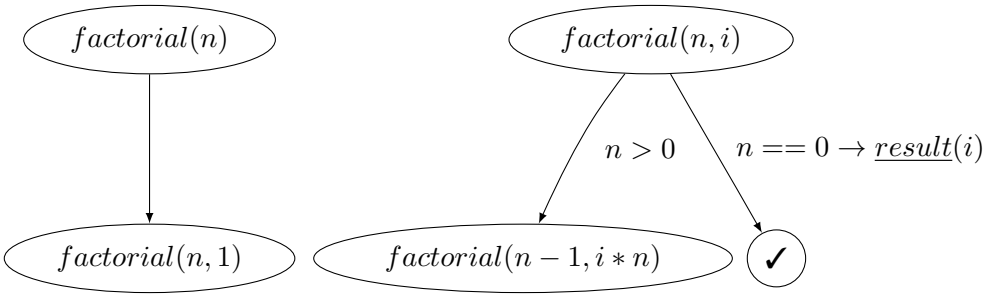


Figure 8: LTS of $factorial(n)$

Definition 9 (Sum operator)**.** The sum operator $\sum_{d:D} p(d)$ generalizes the choice operator for actions parameterized with data in arbitrary domains.

This operator lets us choose over a possibly infinite number of processes. The equivalent LTS construction is analogous to the binary sum one.

2.2.2 Recursive Processes

Processes are not always finite, and many systems need recursive definitions. These definitions are written as equations, that represent the relations that the instances of the (parameterized) process must satisfy. For example, *factorial(n,i)* process is recursively defined as follows:

$$factorial(n) = factorial(n, 1)$$

$$factorial(n, i) = i > 0 \rightarrow factorial(n - 1, i * n) \diamond \underline{result}(n)$$

This example shows what is called a *recursive specification*, a set of process equations with only a single variable in each left-hand side. If every occurrence of a defined process variable at the right-hand side is preceded by an action, then we say that the recursive specification is *guarded*, or that it is a *guarded recursive specification.*

Guardedness is needed if we want to model sound systems. Any process defined by an unguarded recursive specification would contain a "loop" where no action is performed, and thus does not soundly define the behavior of a process.

2.2.3 Parallel Processes

Definition 10 (Parallel composition)**.** Let $p, q \in P$. The parallel composition $p \| q$ is a process in which actions from p and q happen independently from each other, but also simultaneously. The process terminates when both p and q terminate.

Therefore, the parallel composition can perform actions from one process or the other, or both at the same time, without any order between them.
To facilitate the analysis of this operator, the following auxiliary operators are introduced.

Definition 11 (Left-merge and right merge operators)**.** Let $p, q \in P$. The left-merge of p and q, $p \lfloor\lfloor q$, is the process that parallelizes p and q, but guaranteeing that the first action will be from p. Analogously, the right-merge $p \rfloor\rfloor q$ ensures that the first action performed comes from q.

Definition 12 (Synchronization operator)**.** Let $p, q \in P$. The synchronization of p and q, $p|q$, is a parallelization of p and q that forces the simultaneous execution of some first actions of p and q.

It must be noticed that the synchronization operator (for processes) and the multiaction operator (for actions) are overloaded, the reason being that the meaning of both is "the same", but at different levels.

For two processes P and Q, the LTS of $P \| Q$ results by creating all possible (synchronized) combinations of their actions. Let us see an example to showcase the procedure.

Let $P = a \cdot b$ and $Q = c \cdot d$, then $P \| Q = a \cdot b \| c \cdot d$. The LTS's are the following:

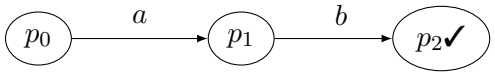


Figure 9: LTS of P

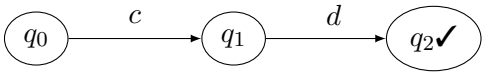


Figure 10: LTS of Q

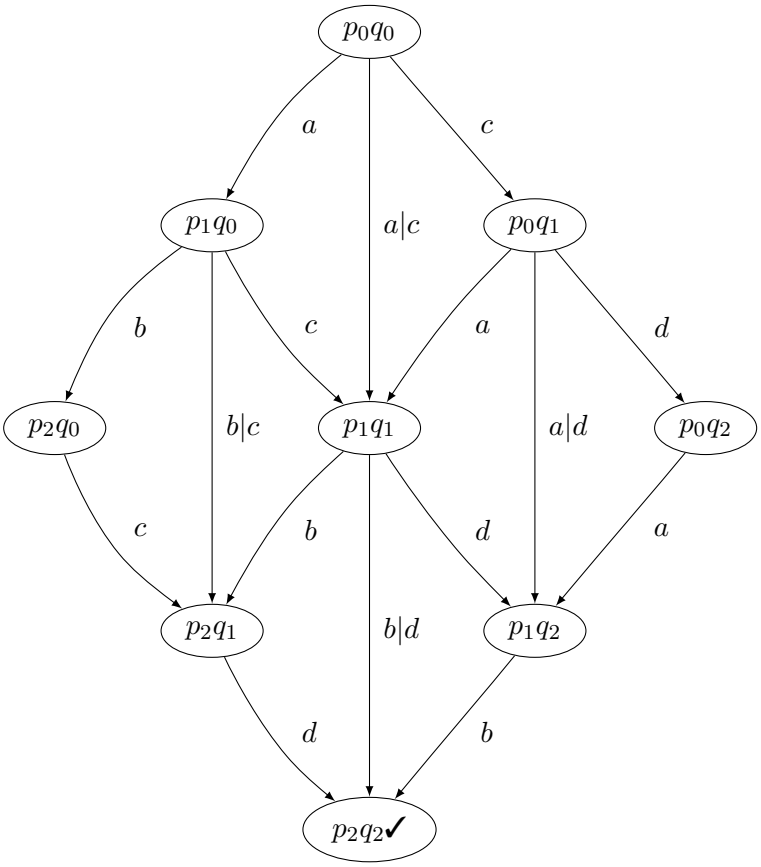


Figure 11: LTS of $P \| Q$

At it is clear in this example that there is no "easy" LTS composition for parallel operators: the fact that any possible interleaving must be generated produces a multiplicative explosion of the number of states.

2.2.4 Communication, Control and Simplification of Processes

This section is related to parallel processes, and how we represent the communication of data and synchronization of independent processes. We also show the operators that let us hide or block certain actions, which reduce the size of the state space, in order to simplify the analysis of the resulting LTSs.

Definition 13 (Communication)**.** Let $a_1|\cdots|a_n \in Act$, $c \in Act$. A communication $a_1|\cdots|a_n \rightarrow c$ represents the interaction that occurs between n systems when actions $a_1|\cdots|a_n$ are performed simultaneously, and the substitution of that interaction to a single action c, witness of the communication. This substitution is only possible when the data, if any, is equal in all substituted actions.

Definition 14 (Communication operator)**.** Let C be a set of communications, as we just

defined, $p \in P$. The communication of C applied to p, $\Gamma_C(p)$, results in the substitution of the multiactions that conform C, for their respective witness actions.

Definition 15 (Allow operator)**.** Let $\{a_i\}_{i \in I} \subset Act$. The allow operator, $\nabla_{\{a_i\}}$, applied over a process p, generates a process that forbids, e.g. blocks, the execution of actions not included in $\{a_i\}$.

With the allow operator we can block actions which we do not want to be performed. The main purpose of the allow operator is to block actions that should communicate but did not; most likely because the parallel composition generated combinations that are not coherent with the expected behavior, and should be blocked. This two operators are usually used together in the form of an allow of the communication of a process, this way reducing the space state to be explored by the automatic tools.

Definition 16 (Blocking operator)**.** Let $B \subset Act$, $p \in P$. The blocking operator of B in p, $\partial_B(p)$, substitues every action in p contained in B with the deadlock process σ.
This operator acts regardless of the data in the actions.

Definition 17 (Renaming operator)**.** Let R be a set of renamings of actions, of the form $a \to b, a, b \in Act$, and $p \in P$. The renaming of R in p, $\rho_R(p)$, renames the actions in p that appear in R, to their new names.

Now we introduce the notion of τ action or internal action, and the related hiding operator.

Definition 18 (Internal action)**.** The internal action τ, represents an action whose execution cannot be distinguished or perceived; and is, as its name suggests, internal to the system where it appears so that it is not observable from the outside.

This internal action has as primary purpose, the hiding of operations that a machine performs but do not concern us, or even cannot be seen by us. This action is very important for the verification and validation of systems, because later on we will see that it lets us discern from systems that could be thought to behave the same way, but in some cases do not.

Definition 19 (Hiding operator)**.** Let $I \subset Act$. The hiding of I, τ_I, is the process that substitutes the actions that appear in I by the internal action τ.

In practice, we will be interested in hiding actions in a two-step manner: by renaming them to an auxiliary action and then hiding that action. For this reason, the prehide operator is defined. In this work the auxiliary action will take the name of *int*, from "internal".

Definition 20 (Prehide operator)**.** Let $I \subset Act$. The prehide operator induced by I, Υ_I, renames all the actions in I to the auxiliary action int.

Let us model a handshake between two devices. We have a device P, which starts the communication sending the first packet of the handshake, a "Hello", and will wait for the corresponding "Acknowledge". The receiver Q, will receive the first message and send its "Acknowledge". We represent this behavior by:

$$P = hello_{out} \cdot ack_{in}$$

$$Q = hello_{in} \cdot ack_{out}$$

As we can see, these are the same processes from 11, barring different naming of the actions. Looking at the LTS, we see that there are many possible paths to the final state, some of which are not correct within the protocol we are modeling. For example, it is of no use that there is a path where P has received the "Acknowledge" message before Q has sent it! There is not even a reliable communication between the processes, since actions are executed simultaneously, but without any interaction between them. To solve this, we must first pair the actions that communicate ($hello_{out}$ with $hello_{in}$ and ack_{out} with ack_{in}), and then we will only allow those communicated actions. This follows the aforementioned scheme of allow-communicate processes. The resulting process is $\nabla_{\{hello,ack\}}\Gamma_{\{hello_{in}|hello_{out}\to hello, ack_{in}|ack_{out}\to ack\}}(P\|Q)$, and its LTS, operating step by step, is this:

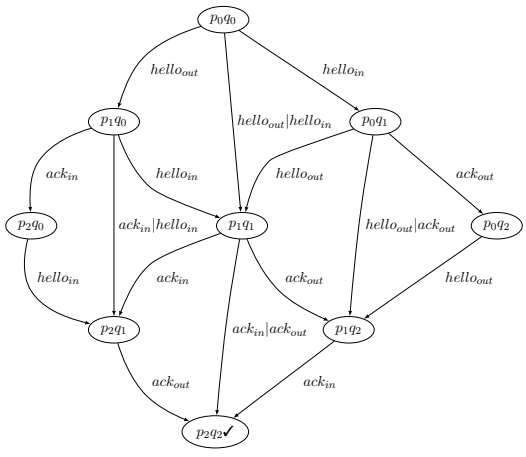


Figure 12: LTS of $P\|Q$

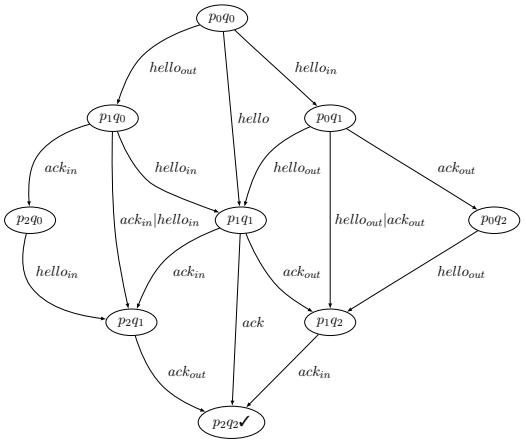


Figure 13: LTS of $\Gamma_{\{hello_{in}|hello_{out}\to hello, ack_{in}|ack_{out}\to ack\}}(P\|Q)$

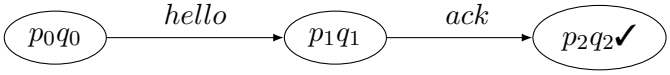


Figure 14: LTS of $\nabla_{\{hello,ack\}}\Gamma_{\{hello_{in}|hello_{out}\to hello,ack_{in}|ack_{out}\to ack\}}(P\|Q)$

2.3 Axioms of the Process Algebras

We indicated previously in this section the need for formal axioms for the process algebra in order to be able to operate on it. These axioms vary from simple, lengthy, straightforward and convoluted, and unfortunately must be left out of this work. To consult these axioms, please read Chapter 1, Sections 4 and 5 of [8].

3 Process Equivalence

The most basic task one is interested in when modeling and verifying processes is knowing if two processes are equivalent. In this sections I will present the different equivalences commonly used, their properties and when to use them.

3.1 Equivalences without internal actions

We will show here the equivalences intended for verifying properties that involve knowing all actions performed by the system.

3.1.1 Trace equivalence

The trace equivalence is the loosest of the equivalences usually used, that is, any processes that are equivalent by other definitions must be also trace equivalent, but not the converse. We can see trace equivalence as the equivalence of the sequences of actions possible from the initial states. This equivalence does not analyze communication between actions, and therefore we cannot "identify" deadlocks. Each trace is represented by the concatenation of the actions executed along. We also include the finishing symbol when we reach a terminating state so that no new action can be performed.

Definition 21 (Trace equivalence)**.** Let $A = (S, Act, \to, s_0, T)$ be a LTS. The set of traces starting from $t \in S$, $Traces(t)$, is the minimal set that satisfies:

- $\varepsilon \in Traces(t)$
- ✓$\in Traces(t) \iff t \in T$

- $t \xrightarrow{a} t' \wedge \sigma \in Traces(t') \Rightarrow a \cdot \sigma \in Traces(t)$

As we said previously, when we simply talk about the set of traces of an LTS we are referring to the one starting from the initial state of the LTS, $Traces(s_0)$. Now, two states are trace equivalent if their set of traces are equal, and two LTS are equivalent if their initial states are so.

The trace equivalence is to be used when the sequence of actions is the only thing we are interested in, without preserving any branching behavior. In this case, trace equivalence is the best suited, as it can reduce LTS to some smaller equivalent LTS, than the other equivalences. We must also consider the fact that computing the trace of an arbitrary LTS is not at all efficient. To be precise, to decide trace equivalence is a $PSPACE-Complete$ problem [11].

An example of two trace equivalent processes is given by processes $P = (a \cdot c) + (a \cdot c)$ and $Q = (a + b) \cdot c$. Their LTS's are:

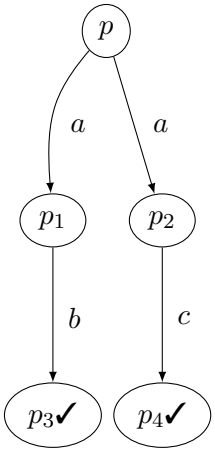


Figure 15: LTS of P

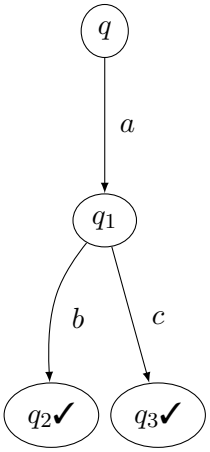


Figure 16: LTS of Q

It is obvious that both LTS's have as set of traces $\{\varepsilon, a, b, a \cdot c, b \cdot c, a \cdot c{\cdot}\checkmark, b \cdot c{\cdot}\checkmark\}$ and therefore they are trace equivalent.

3.1.2 Completed trace equivalence

Completed trace equivalence differs from trace equivalence in that it only includes sequences of actions that end in a terminating state or in a deadlock.

Definition 22 (Completed trace equivalence)**.** Let $A = (S, Act, \rightarrow, s_0, T)$ be a LTS. The set of completed traces starting from $t \in S$, $CompletedTraces(t)$, is the minimal set that satisfies:

- $\varepsilon \in CompletedTraces(t)$ if $t \notin T \wedge \nexists t \xrightarrow{a} t'$
- ✓$\in CompletedTraces(t) \iff t \in T$
- $t \xrightarrow{a} t' \wedge \sigma \in CompletedTraces(t') \Rightarrow a \cdot \sigma \in CompletedTraces(t)$

We can observe that the only difference between the definition of Traces(t) and CompletedTraces(t) is that we do not include the empty sequence by default, thus not generating all partial sequences previous to reaching a terminating or deadlocked state. As a consequence, we can build on top of traces equivalence a certain detection of deadlocks. The drawback of this is that it is not preserved by all the defined operators. Namely, the blocking operator applied to two complete trace equivalent LTSs can make the resulting LTSs not equivalent. This an inmediate consequence of the fact that whenever any state of an LTS has some transition leaving it, we have that $\forall t \in S$ CompletedTraces$(t) = \emptyset$.

In particular, for $P = a \cdot b \cdot P + a \cdot c \cdot P$ and $Q = a \cdot (b + c) \cdot P$, we have that both generate infinite sequences of a's interleaved with b's and c's. Because these sequences are infinite, complete trace sets for both processes are empty, and therefore both processes are complete trace equivalent.

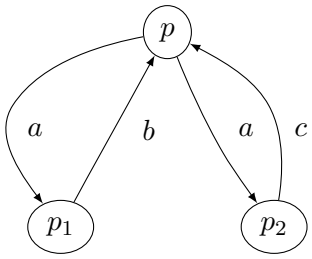


Figure 17: LTS of P

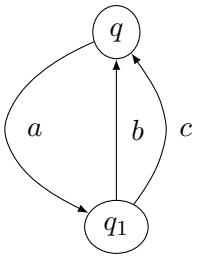


Figure 18: LTS of Q

But when the blocking operator of action c is applied to P and Q, the resulting processes become not complete trace equivalent, as we can easily see in the generated LTS's: the first, corresponding to P, has $a \cdot$✓as a complete trace, which is not the case when blocking Q.

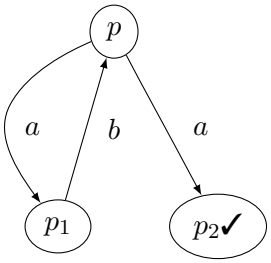


Figure 19: LTS of *BlockedP*

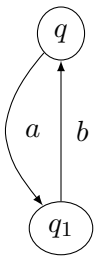


Figure 20: LTS of *BlockedQ*

3.1.3 Failure equivalence

Closely related to completed trace equivalence appears the failure equivalence, which solves preservation of equivalence by the blocking operator. The failure equivalence relates as many behaviors as possible, while preserving traces and deadlocks.

In order to define it, we introduce the set of failure pairs of a process, composed by a sequence of actions and some set of actions that (in some cases) cannot be performed after executing this sequence.

Definition 23 (Failure equivalence)**.** Let $A = (S, Act, \rightarrow, s_0T)$ be a LTS. A set F is called a refusal set of a state t if:

- $\forall a \in F \ \nexists t'$ such that $t \xrightarrow{a} t'$
- $\checkmark \in F \Rightarrow t \notin T$

The set of failure pairs of t, $FailurePairs(t)$ is defined as the minimal set of pairs that satisfies:

- $(\varepsilon, F) \in FailurePairs(t)$ if F is a refusal set of t.
- $(\checkmark, F) \in FailurePairs(t) \iff t \in T \wedge F$ is a refusal set of t
- $t \xrightarrow{a} t' \wedge (\sigma, F) \in FailurePairs(t') \rightarrow (a \cdot \sigma, F) \in FailurePairs(t)$

Again, two states are failure equivalent iff their failure pairs are the same, and two LTS are failure equivalent iff their initial states are.

Failure equivalence is used when we want to preserve deadlocks, some branching behavior and equivalence under application of the operators of the algebra. In practice, the size of the failure pairs is considerable, albeit empirically faster to compute than trace equivalence sets. [17]

Next we present an example showing that the complete-trace equivalent processes in our previous example are not failure equivalent.

For the LTS of P, we observe that its set of failure pairs is $\{(\varepsilon, U \subseteq \{b, c, \checkmark\}), (a, U \subseteq \{a, b, \checkmark\}), (a, U \subseteq \{a, c, \checkmark\}), \dots\}$; while for Q the set of failure pairs is $\{(\varepsilon, U \subseteq \{b, c, \checkmark\}), (a, U \subseteq \{a, \checkmark\}), \dots\}$, where the ellipsis represent pairs with traces of length greater or equal than two.

We clearly see that the pairs $(a, \{b\})$ and $(a, \{c\})$, among many other failure pairs of P, do not appear in Q's failure pairs set.

The next example showing a pair of failure equivalent processes will help us show the difference between the branching behavior captured by failure equivalence and the next equivalence: strong bisimulation.

Let be $P = a \cdot (b + c \cdot d) + a \cdot (f + c \cdot e)$ and $Q = a \cdot (b + c \cdot e) + a \cdot (f + c \cdot d)$,

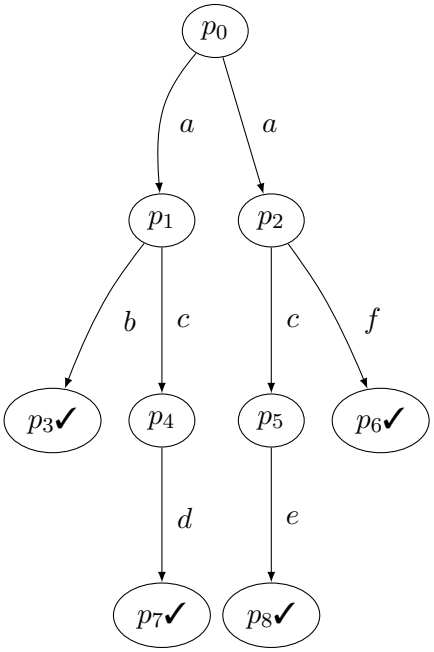


Figure 21: LTS of P

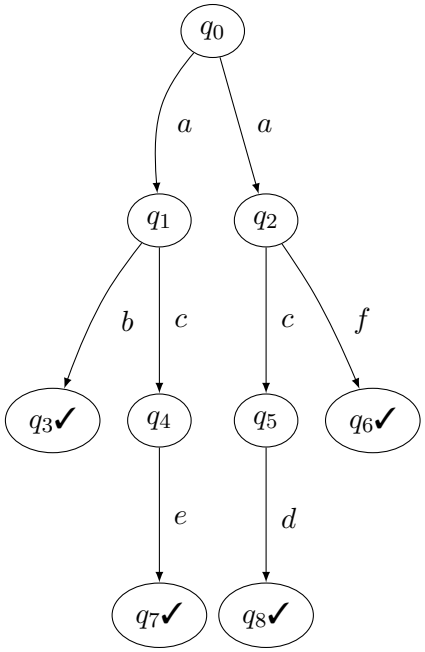


Figure 22: LTS of Q

Let us see that these two processes are failure-equivalent. Let us begin by computing the refusal sets for all nodes.

- For process P:
  - p_0 has $\{b, c, d, e, f, \checkmark\}$ and all its subsets as refusal sets,
  - p_1 has $\{a, d, e, f, \checkmark\}$ and all its subsets as refusal sets,
  - p_2 has $\{a, b, d, e, \checkmark\}$ and all its subsets as refusal sets,
  - p_3 has $\{a, b, c, d, e, f\}$ and all its subsets as refusal sets,
  - p_4 has $\{a, b, c, e, f\ \checkmark\}$ and all its subsets as refusal sets,
  - p_5 has $\{a, b, c, d, f\ \checkmark\}$ and all its subsets as refusal sets,
  - p_6 has $\{a, b, c, d, e, f\}$ and all its subsets as refusal sets,
  - p_7 has $\{a, b, c, d, e, f\}$ and all its subsets as refusal sets,
  - p_8 has $\{a, b, c, d, e, f\}$ and all its subsets as refusal sets.
- For process Q:

- q_0 has $\{b, c, d, e, f, \checkmark\}$ and all its subsets as refusal sets,
- q_1 has $\{a, d, e, f, \checkmark\}$ and all its subsets as refusal sets,
- q_2 has $\{a, b, d, e, \checkmark\}$ and all its subsets as refusal sets,
- q_3 has $\{a, b, c, d, e, f\}$ and all its subsets as refusal sets,
- q_4 has $\{a, b, c, d, f\ \checkmark\}$ and all its subsets as refusal sets,
- q_5 has $\{a, b, c, e, f\ \checkmark\}$ and all its subsets as refusal sets,
- q_6 has $\{a, b, c, d, e, f\}$ and all its subsets as refusal sets,
- q_7 has $\{a, b, c, d, e, f\}$ and all its subsets as refusal sets,
- q_8 has $\{a, b, c, d, e, f\}$ and all its subsets as refusal sets.

Next, we apply the definition to compute the failure pairs of the two initial states of these processes.

- For the initial state p_0 of P:
  - $(\varepsilon, \{b, c, d, e, f, \checkmark\}) \in FailurePairs(p_0)$, and the same for all the subsets of this refusal,
  - $(a, \{a, d, e, f, \checkmark\}) \in FailurePairs(p_0)$, and the same for all the subsets of this refusal,
  - $(a, \{a, b, d, e, \checkmark\}) \in FailurePairs(p_0)$, and the same for all the subsets of this refusal,
  - $(a \cdot b, \{a, b, c, d, e, f\}) \in FailurePairs(p_0)$, and the same for all the subsets of this refusal,
  - $(a \cdot c, \{a, b, c, e, f\ \checkmark\}) \in FailurePairs(p_0)$, and the same for all the subsets of this refusal,
  - $(a \cdot c, \{a, b, c, d, f\ \checkmark\}) \in FailurePairs(p_0)$, and the same for all the subsets of this refusal,
  - $(a \cdot f, \{a, b, c, d, e, f\}) \in FailurePairs(p_0)$, and the same for all the subsets of this refusal,
  - $(a \cdot c \cdot d, \{a, b, c, d, e, f\}) \in FailurePairs(p_0)$, and the same for all the subsets of this refusal,
  - $(a \cdot c \cdot e, \{a, b, c, d, e, f\}) \in FailurePairs(p_0)$, and the same for all the subsets of this refusal.
- For the initial state q_0 of Q:

- $(\varepsilon, \{b, c, d, e, f, \checkmark\}) \in FailurePairs(q_0)$, and the same for all the subsets of this refusal,
- $(a, \{a, d, e, f, \checkmark\}) \in FailurePairs(q_0)$, and the same for all the subsets of this refusal,
- $(a, \{a, b, d, e, \checkmark\}) \in FailurePairs(q_0)$, and the same for all the subsets of this refusal,
- $(a \cdot b, \{a, b, c, d, e, f\}) \in FailurePairs(q_0)$, and the same for all the subsets of this refusal,
- $(a \cdot c, \{a, b, c, d, f \checkmark\}) \in FailurePairs(q_0)$, and the same for all the subsets of this refusal,
- $(a \cdot c, \{a, b, c, e, f \checkmark\}) \in FailurePairs(q_0)$, and the same for all the subsets of this refusal,
- $(a \cdot f, \{a, b, c, d, e, f\}) \in FailurePairs(q_0)$, and the same for all the subsets of this refusal,
- $(a \cdot c \cdot e, \{a, b, c, d, e, f\}) \in FailurePairs(q_0)$, and the same for all the subsets of this refusal,
- $(a \cdot c \cdot f, \{a, b, c, d, e, f\}) \in FailurePairs(q_0)$, and the same for all the subsets of this refusal.

As we can see, $FailurePairs(p_0) = FailurePairs(q_0)$, and therefore P and Q are failure equivalent.

3.1.4 (Strong) bisimulation

Bisimulation, or strong bisimulation, is the main equivalence relation used in process analysis, and also the finest of them. Bisimulation computation can be thought as the search of any possible difference which would cause their inequivalence. Therefore, intuitively, bisimulation equivalent processes are practically indistinguishable.

One of the reasons why bisimulation is more useful in practice than the other equivalences is that the algorithm for checking bisimulations is a decision problem in P. However, it is also true that sometimes it becomes too fine, so that two processes that would "behave the same" at a certain level of required observability, could be deemed inequivalent.

Definition 24 (Bisimulation)**.** Let $A = (S, Act, \rightarrow, s_0, T)$ be a LTS. Strong bisimulation is any symmetric binary relation $R \subseteq (S \times S)$, such that $\forall s, t \in S$, if sRt:

- $s \xrightarrow{a} s' \Rightarrow \exists t', t \xrightarrow{a} t' \wedge s' R t'$

- $t \xrightarrow{a} t' \Rightarrow \exists s', s \xrightarrow{a} s' \wedge s' R t'$
- $s \in T \iff t \in T$

Two states are strongly bisimilar if, and only if, there is a strong bisimulation relating them; and two LTS are strongly bisimilar if, and only if, their initial states are bisimilar.

An interesting property of strong bisimulation is that for every LTS there exists a unique minimal bisimilar equivalent LTS.

Reusing the previous example, we can see why P and Q are not strongly bisimilar. If that were the case, there would exist a bisimulation relating p_0 and q_0. But then, since we have that $p_0 \xrightarrow{a} p_1$, there must be some state q_α bisimilar to p_1 (which must be q_1, because we have that $p_1 \xrightarrow{b} p_3$). But then, in a similar way, we obtain that p_4 and q_4 should be bisimilar, but this is clearly not the case, because the transitions leaving these states are totally different.

3.1.5 Van Glabbeek linear time-branching time spectrum

All of these process equivalences, and many others, were classified in a famous diagram that compared their strength by Van Glabbeek [18]. The diagram displays the finest equivalences at the top and the coarsest at the bottom, and the arrows represent a refinement relation.

A more detailed classification and diagram are shown in [6].

3.2 Equivalences with internal actions

The equivalences previously mentioned are not enough when the processes communicate and have internal actions. As we have seen, neither trace nor bisimulation equivalences consider the internal actions as special actions. As a consequence, these actions can be "observed" as they would be ordinary actions. Therefore, we need some adequate, more sophisticated definitions in order to capture their desired internal characteristics.

3.2.1 Weak trace equivalence

Weak trace equivalence is the τ analogous to trace equivalence. In this case, the natural approach of simply "ignoring" the execution of internal actions, but allowing them, is enough, and we collapse traces including internal actions in the usual way (i.e. as done in automata theory with ε transitions). As with trace equivalence, weak trace equivalence is defined by the equality of two sets of traces, in this case weak traces.

Weak trace equivalence is, as one could expect, the weakest of all equivalences that abstract internal behavior, not preserving deadlock nor branching behavior. The calculation of the smallest equivalent LTS is very hard; but in some cases, we can gain a better understanding of the behavior of a system by observing the weak trace equivalent system, and then changing the model if we find easily visible errors or apply a finer equivalence if necessary.

Definition 25 (Weak trace equivalence)**.** Let $A = (S, Act, \rightarrow, s_0, T)$ be a LTS. The set of weak traces starting from a state t, $WTraces(t)$, is the minimal set that satisfies:

- $\varepsilon \in WTraces(t)$
- $\checkmark \in WTraces(t) \iff t \in T$
- $t \xrightarrow{a} t', a \neq \tau \wedge \sigma \in WTraces(t') \Rightarrow a \cdot \sigma \in WTraces(t)$
- $t \xrightarrow{\tau} t' \wedge \sigma \in WTraces(t') \Rightarrow \sigma \in WTraces(t)$

While for traces, and thus for trace equivalence, we can obtain the corresponding weak version in a very natural (an easy) way, this is not certainly the case for bisimulation, where many technical subtleties produce a collection of weak variants.

3.2.2 Branching bisimulation

With branching bisimulation, we introduce a simple way of abstracting the execution of τ transitions when defining a bisimulation. The unobservability of τ actions is captured by allowing the τ action to be simulated by another τ action, but also doing nothing, since nothing is "observed" when a τ transition is executed.

Definition 26 (Branching bisimulation)**.** Let $A = (S, Act, \rightarrow, s_0, T)$ be a LTS. The branching bisimulation in A is symmetric binary relation $R \subseteq (S \times S)$ such that $\forall s, t \in S$, if sRt, then we have:

- If $s \xrightarrow{a} s'$, then either:
  - $a = \tau \wedge s'Rt$
  - There exists a sequence of (zero or more) τ transitions $t \xrightarrow{\tau^*} t'$ such that $sRt' \wedge t' \xrightarrow{a} t''$
- $s \in T$, there exists a sequence of (zero or more) τ transitions $t \xrightarrow{\tau^*} t'$ such that $sRt' \wedge t' \in T$

As usual, equivalence of two LTS represents just the equivalence of the corresponding initial states.

Since branching bisimulation does not observe τ transitions neither their goal, it cannot observe τ transitions that leave the state of the system the same (the so-called τ-loops). By removing τ-loops, branching bisimulation captures a notion of fairness, meaning that a system will never get stuck in an (unobservable) τ-loop.

A state with a τ-loop is called a divergent state.

3.2.3 Divergence preserving branching bisimulation

But sometimes, we are interested in distinguish between divergent and non-divergent states, so that this (basic) branching bisimilarity totally abstracts from divergence. For this purpose, we can extend the definition of branching bisimulation with one additional rule:

Definition 27 (Divergence preserving branching bisimulation)**.** Let R be a branching bisimulation relation. We say that the relation R is a divergence preserving branching bisimulation if, and only if, it satisfies the following condition:

- If sRt then there exists an infinite sequence $s \xrightarrow{\tau} \cdots$ if and only if there is an infinite sequence $t \xrightarrow{\tau} \cdots$.

3.2.4 Rooted branching bisimulation

The two previous bisimulations lack a very important property: congruence. In figure 23 and 24 we can see that the two LTS's of each figure are branching bisimilar. We would expect that by combining an LTS from one figure with one LTS of the other, every combination should be equivalent. This is not the case with branching bisimulation, because there is no branching bisimilar state to our initial state after performing a τ action.

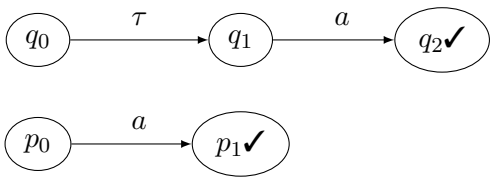


Figure 23: Branching bisimilar a's

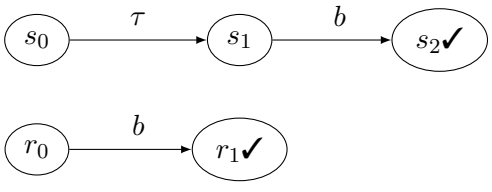


Figure 24: Branching bisimilar b's

To solve this problem, the rooted version of branching bisimulation is introduced.

Definition 28 (Rooted branching bisimulation)**.** Let $A = (S, Act, \rightarrow, s_0, T)$ be a LTS. A rooted branching bisimulation is a branching bisimulation such that, $\forall s, t \in S$, if sRt then:

- $s \xrightarrow{a} s' \Rightarrow \exists t', t \xrightarrow{a} t' \wedge s'$ is branching bisimilar to t'.

As opposed to the previous bisimulation definitions, this one is not defined recursively. This is because the only needed change concerns the first transition of the equivalent states, as those reached after the first transition are only required to be branching bisimilar, and not rooted branching bisimilar.

We can notice that if we were to define these bisimulations in a recursive way, we would simply end up with the definition of strong bisimulation!

3.2.5 (Rooted) Weak bisimulation

Finally, we introduce the "plain" weak forms of bisimulation. In this case part of the "branching information" that branching bisimulation had to preserve needs not to be preserved anymore, and thus we obtain a coarser semantics to the handling of τ actions.

Definition 29 (Weak bisimulation)**.** Let $A = (S, Act, \rightarrow, s_0, T)$ be a LTS. A weak bisimulation is a symmetrical binary relation R such that $\forall s, t \in S$, with sRt:

- If $s \xrightarrow{a} s'$, either
  - $a = \tau \wedge s'Rt$
  - $\exists t \xrightarrow{\tau^*} \xrightarrow{a} \xrightarrow{\tau^*} t'$ such that $s'Rt'$
- $s \in T \Rightarrow \exists t \xrightarrow{\tau^*} t', t' \in T$

Definition 30 (Rooted weak bisimulation)**.** Let $A = (S, Act, \rightarrow, s_0, T)$ be a LTS. A rooted weak bisimulation on A is a weak bisimulation relation R which when sRt, also satisfies:

- $s \xrightarrow{\tau} s' \Rightarrow \exists t \xrightarrow{\tau} \xrightarrow{\tau^*} t'$ such that s' is weakly bisimilar to t'.

3.3 Basic criteria when choosing an equivalence relation

After defining all these (certainly many) relations between LTSs, we will briefly discuss which one to choose, depending on our intended goals, using some simple examples that intuitively show the reasons why the indicated one would be the most adequate in each case.

3.3.1 The criteria to use when we do not have any internal steps

In LTSs without internal steps, there will be a clear statement:

Strong bisimulation is always safe.

The reason why this is true is that, strong bisimilarity only relates processes that are behaviorally indistinguishable, and therefore when we identify two bisimilar systems there is no risk of losing any information.

In the opposite end of the spectrum, we have trace equivalence, where the LTSs are reduced much more, easily losing some branching information. As a consequence, we cannot use trace equivalence when interested in the identification of deadlocks. But traces are still enough to characterize some reachability properties, thus remaining useful to establish safety properties.

An intermediate choice would be failure equivalence, preserved by the application of the usual operators in process algebras. This equivalence is highly useful as it has been empirically shown to be efficient to calculate [17], and with absence of non-deterministic parallelism, failure and trace equivalences are practically equivalent.

For example, if we were to implement the embedded system of a modern vehicle, we would expect that no component deadlocks; for instance, when we need to use the electronically controlled break, or when the parking assistance system steers the wheel. For these systems, if there were no non-determinism, we could analyze the failure equivalent LTS to verify whether or not deadlocks could appear, but trace equivalence would not be enough, for the reasons discussed above.

3.3.2 The criteria to take into account when considering internal steps

With internal steps, divergence preserving branching bisimulation would be the safe option playing the role of strong bisimulation in the previous case.

Now, plain branching bisimilarity (without preserving divergences), would be enough as far as liveness properties that could be (theoretically) lost when divergences appear are not important, or when we assume that those liveness properties (mainly bounded bypass, as we are dealing with potential infinite loops) will be guaranteed by other means.

Moreover, in practice weak bisimulation works as branching bisimulation in many cases, so it can be safely used as an alternative.

But since weak trace equivalence removes all the internal actions, the (exact) branching behavior can be lost in some cases, so we should carefully study if weak bisimilarity is

enough depending on the exact information that we would need to preserve.

As an example, let us think of a cryptographic communication protocol, in which the internal actions are not interesting to us by themselves, and we only want to know if there is a way the password (think of it as a private key, like in RSA) could be leaked to an intruder. In this case, we do not care about τ-loops, so we can use plain branching bisimulation. However, we could not use weak bisimulation, nor weak trace equivalence, because of the high parallelism (and thus non-determinism) in the system.

Another example could be the modeling of a routing protocol, like RIP, where we would like to ensure that the routing is set up correctly always, and that the system does not get stuck computing divergent calculation (as it occurs in RIPv1). This latter property of absence of livelocks is related to the (possible) presence of τ loops, so that divergence preserving branching bisimulation should be imposed to remain safe.

4 Modal Logics

Up until now, we have shown how to model a concurrent system using a high-level (algebraic) syntax and the semantic LTS representation, and how equivalences preserve properties that we want to verify. Next, we will focus on the formalism that allows to model properties, and verify them over the LTS previously produced. These formalism are the modal logics. In this work, we will briefly introduce the Hennessy-Milner logic and an important extension, the μ-calculus.

4.1 Hennessy-Milner Logic

The Hennessy-Milner logic follows the syntax of propositional logic, extending it with two quantifier-like modal operators, related to the usual first-order $\exists$ and $\forall$. Its defining grammar is as follows:

$$\varphi ::= true \mid false \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \rightarrow \varphi \mid \langle a \rangle \varphi \mid [a]\varphi$$

Any logic statement, applied to a certain state, can only evaluate to True or False. The usual logic operators $\vee$, $\wedge$, $\neg$, have their usual meanings. Also, the implication operator is defined to ease the writing of formulas, as it can be rewritten with $\wedge$ and $\neg$.
The main modal operator is the diamond modality, $\langle a \rangle$. $\langle a \rangle \varphi$ evaluates to True when *there exists* an a-transition from the current state that leads to a state where φ is true. Its dual, the box modality, $[a]\varphi$, represents if *every* transition from the current state ends in a state that satisfies φ.

Because our main goal is to analyze the dynamics of concurrent systems (via their semantic LTS representation), it is clear that these two modal operators are crucial to state the required properties.

As with first-order logic, the modalities here verify that not $\langle\rangle\varphi = []\neg\varphi$, $\neg[]\varphi = \langle\rangle\neg\varphi$.

As usual, we will say that the labeled transition system satisfies the modal formula if, and only if, it is satisfied by the initial state.

4.2 μ-calculus

The previous definition of the Hennessy-Milner will not be enough for us, as we have a rather problematic structure some of our transition systems: recursive definitions. To solve this limitation, we must extend the modal logic thus introducing *fixed-point* modalities.

A fixed-point p of a function f is an element that verifies the equation $p = f(p)$. In our case, the functions will be recursive logic formulas; and the domain, the set of states of the LTS. In general, these fixed-points will not be unique, but with respect to the lattice produced by the set inclusion relation, we will have a theorem ensuring the existence of least and greatest fixed points. Respectively, we can distinguish two especially interesting fixed-point modalities: the μ and ν operators.

The minimal fixed-point, $\mu X.\varphi(X)$, is satisfied by the states in the smallest set of states where φ is satisfied. Similarly, the maximal fixed point, $\nu X.\varphi(X)$, is satisfied by the states in the largest set of states where φ is satisfied.

Taking $\varphi(X) = X$ we obtain a trivial example:

$$\mu X.X$$
$$\nu Y.Y$$

where $\mu X.X$ is obviously equivalent to false and $\nu Y.Y$ equivalent to true, because the empty set and the full set of states are the smallest and largest solutions of the equation $X = X$, respectively.

Another way to understand the fixed-point modalities is by considering the minimal fixed-point as a recursive function that is satisfiable in a certain finite number of steps. On the other hand, the maximal fixed point would be true after infinitely many steps. These two metaphors bring us the notions of liveness and safety, respectively.

A liveness property of a condition ensures that the condition will successfully occur within a finite number of steps. Dually, a safety property ensures that the condition will not happen within an infinite number of steps. For example, the convergence of a certain algorithm can be thought as a liveness property, whereas the absence of deadlocks would be a safety property.

In order to use fixed-point modalities, we first need to prove the existence of fixed-points. If the fixed-point variable in a fixed-point modality appears evenly preceded by negations, then there exists a unique solution to the modality. You can find a proof of this property in appendix E.

To use the full potential of fixed-point modalities, we have to combine minimum and maximum fixed-points. With this we obtain new capabilities, such as stating that a property must happen if it is available a potentially infinite number of times, or if some other action only happens a finite number of times. The latter is a notion of fairness. We can find interesting fairness properties in multiqueued systems like gas stations or supermarkets, where the systems should guarantee that every queue will eventually get served forever.

Fairness for an **enter** action can be specified by the following formula:

$$\nu X.\mu Y.((\langle enter\rangle Y \vee [true]X) \wedge (\neg\langle enter\rangle X \vee [true]Y))$$

This formula states that for every state there is a finite sequence of states where the action **enter** cannot be performed. This is useful, paired with the previous formula, to verify that a mutual exclusion lock is fair (i.e. every thread using the lock has the opportunity to **enter** its critical section, without getting live-locked).

4.3 Regular formulas

If we consider the actions of a concurrent system as an alphabet Σ, we can define the regular expressions over said alphabet. Using these newly defined "transition languages", we can make the formulas more meaningful. The action languages are defined, as usual, with the following grammar:

$$af ::= \alpha \mid true \mid false \mid \overline{af} \mid af \cap af \mid af \cup af$$

Here, $true$ and $false$ represent *any* actions and *no* actions, respectively. In other words, $true$ is the full language of words; and $false$, the empty language. The negation $\overline{af}$, union $\cup$, and intersection $\cap$ of action languages have the usual meaning.

To be formally correct, we must define the μ-calculus over action languages. Because actions only appear in the modalities, we only need to define:

$$\langle L\rangle\varphi = \bigvee_{\alpha\in L}\langle a\rangle\varphi$$
$$[L]\varphi = \bigwedge_{\alpha\in L}[a]\varphi.$$

Based on these languages, we can define the regular expressions with the common operators:

$$R ::= \varepsilon \mid af \mid R \cdot R \mid R + R \mid R^* \mid R^+$$

But, why do we extend modalities to work with regular expression? The answer is simple: even if adding regular expressions does not add semantic capabilities to μ-calculus, it lets us define formulas that would otherwise be quite large and complex. For example, if we have to check that a property is satisfied when a (finite) even number of a actions are executed, we would write the following formula:

$$\nu X.[a][a](\varphi \wedge X)$$

With regular expression, knowing that the language describing words with a finite even number of a actions is $(aa)^*$, we would rewrite the formula above as:

$$[(aa)*]\varphi$$

These two formulas are equivalent, but the latter is much more concise and understandable. Another example would be the condition that needs to be satisfied to get sound mutual exclusion locks, which are described by the formula:

A more complex example would be the condition that needs to be satisfied to get sound mutual exclusion locks, which are described by the formula:

$$[true * .lock(i).(!unlock(i)) * .lock(j)]false$$

This formula states that, if the lock is locked, no other locking action will be performed until it is unlocked again. Writing this formula in pure μ-calculus syntax would increase its complexity greatly. The verification of this property on some lock implementations can be found in appendix C. .

4.4 Checking Modal Formulas

Once we have defined the formulas we want to check, and the labeled transition systems where they should be checked, there are multiple ways to check the validity of a formula, but we will show here the direct method applied to LTSs. For Hennessy-Milner formulas, the procedure is quite simple. By labeling the states of the LTS with subformulas of the logical statement we can verify the satisfiability of the formula: the formula is satisfiable at the initial state if, and only if, the initial state's label is the complete formula. This process is defined like this:

- First label all states with *true*.
- For each subformula φ, for each state s:
  - Label s with $\neg\varphi$ if it is not labeled with phi
  - Label s with $\varphi \wedge \psi$ if it is labeled with φ and with ψ
  - Label s with $\varphi \vee \psi$ if it is labeled with φ or with ψ
  - Label s with $\langle a \rangle \varphi$ if there is a transition $s \xrightarrow{a} s'$ and s' satisfies φ
  - Label s with $[a]\varphi$ if for all transitions $s \xrightarrow{a} s'$, s' satisfies φ

Because in Hennessy-Milner formulas there is no recursion, the set of subformulas is finite, and thus the algorithm terminates.

In order to check μ-calculus formulas, extending the former algorithm will not suffice. When handling minimal fixed points $\mu X.\varphi(X)$, we will label the states with all subformulas of φ, including X. In the case of maximal fixed points $\nu X.\varphi(X)$, the process is reversed, initially labeling the states with X and removing them from a state when it does not satisfy φ after the iterations stabilize.

5 Syntactic Manipulation of Processes

In previous chapters we have presented several equivalence relations between Labeled Transition Systems. They capture very precise relationships, but require us to obtain in an explicit way the semantic representation of processes. In this chapter we lay down some derivation rules that can be applied to formulas of process algebra to (try to) conclude whether or not they are equivalent.

The main rules used in practice are the induction over data types, the Recursive Specification Principle (RSP), and Koomen's fair abstraction rule. The first two rules relate process formulas that are bisimilar, whereas Koomen's rule has a version for branching bisimulation and another for weak bisimulation.

These rules are used to check if an arbitrary equation is provable, but their generality makes the process of checking provability very cumbersome. For this reason we will have to transform our equations to a process linear form, with a method called linearization. We will not go into much detail about linearization, only let us print out that it can be used as an intermediate step to decide very efficiently strong bisimulation[10].

5.1 General Derivation Rules

To anyone who has worked with derivation rules on the theory of programming languages, these rules are self-explanatory. Nevertheless, we will give a succinct explanation for the more complex ones.

Here, Γ represents the context, the statements above the horizontal bar are the hypotheses and the ones below the bar are the conclusions. When we write $x \notin \Gamma$, we are describing that, as a requirement for the rule to be true, x should not be a free variable on the context Γ.

The congruence rule states that equality is preserved after concatenating processes (with the dot operator). The extensionality rule states that, if x is not a free variable, then we can decompose it from a previous equality, implying the equality of the preceding terms. The abstraction rule states that, if x is not a free variable, then we can surround equal elements by a lambda over x while preserving equality.

The α conversion rule states that we can rename the free variable of a lambda expression, as long as we rename it on the inner terms as well.

The β conversion rule states that we can apply an element to a lambda expression by substituting all appearances of the free variable for the argument.

$\dfrac{}{\Gamma \cup \{p = q\} \vdash p = q}$	start
$\dfrac{}{\Gamma \vdash p = p}$	reflexivity
$\dfrac{\Gamma \vdash q = p}{\Gamma \vdash p = q}$	symmetry
$\dfrac{\Gamma \vdash p = q \quad \Gamma \vdash q = r}{\Gamma \vdash p = r}$	transitivity
$\dfrac{\Gamma \vdash p_1 = q_1 \quad \Gamma \vdash p_2 = q_2}{\Gamma \vdash p_1 \cdot p_2 = q_1 \cdot q_2}$	congruence
$\dfrac{\Gamma \vdash p \cdot x = q \cdot x}{\Gamma \vdash p = q}$	extensionality
$\dfrac{\Gamma \vdash p = q}{\Gamma \vdash \lambda x : D.p = \lambda x : D.q}$	abstraction
$\dfrac{}{\Gamma \vdash \lambda x : D.p = \lambda y : D.(p[x := y])}$	$\alpha - conversion$
$\dfrac{}{\Gamma \vdash (\lambda x : D.p)q = p[x := q]}$	$\beta - conversion$

Table 1: Derivation rules for equality in our process algebra

The other rules come from standard first-order logic.

For logical formulas, rules are very straightforward. The only caveat to notice is that we assume that previously we transform every logical statement to compositions of ($true = false$) and $\Rightarrow$ (which is a functionally complete set for propositional logic, see [19]), and $\forall$.

5.2 Induction over Data Types

This rule will let us work over the structure of data types to prove equality.

Let D be some structured data type (or sort, in mCRL2 nomenclature). The following rule stands:

This states that, for a formula φ to be true for a sort D, it must be true for every possible constructor of D and every combination of their parameters.

This induction rule works with booleans and natural numbers, as well as user defined types, finite or infinite.

$\dfrac{}{\Gamma \cup \{\varphi\} \vdash \varphi}$	generalized start
$\dfrac{\Gamma \vdash true = false}{\Gamma \vdash \varphi}$	principle of explosion
$\dfrac{\Gamma \cup \{\varphi\} \vdash \psi}{\Gamma \vdash \varphi \Rightarrow \psi}$	$\Rightarrow$ introduction
$\dfrac{\Gamma \vdash \varphi \Rightarrow \psi \quad \Gamma \vdash \varphi}{\Gamma \vdash \varphi \Rightarrow \psi}$	$\Rightarrow$ elimination
$\dfrac{\Gamma \vdash \varphi}{\Gamma \vdash \forall x : D.\varphi,\ x \notin \Gamma}$	$\forall$ introduction
$\dfrac{\Gamma \vdash \forall x : D.\varphi}{\Gamma \vdash \varphi[x := p]}$	$\forall$ elimination
$\dfrac{}{\Gamma \vdash \forall x : D.\varphi = \forall y : D.(\varphi[x := y])}$	α-conversion for $\forall$

Table 2: Derivation rules for μ-calculus formulas

$$\frac{\{\Gamma \vdash \bigwedge_{D_i=D} \varphi[x := x_i] \Rightarrow \varphi[x := c_i(x_1, \ldots, x_n)] | c_i \text{ is a constructor of } D\}}{\Gamma \vdash \varphi} \quad \text{induction}$$

5.3 Recursive Specification Principle

The RSP will be the tool to prove equality between recursive process equations. To ensure the uniqueness of the possible solution, we have a condition similar to the one applied in the modalities of μ-calculus, called guardedness. To properly formalize this principle, we shall first introduce the process operators.

A process operator is a mapping with domain D, where D can be the set of processes P or the set of mappings from some tuple of data types to P. Therefore, a process operator can be applied just to a process, or to one parameterized process that depends on data. Thus, we can think of process operators as mappings $D \longrightarrow D$.

Definition 31 (Guarded process)**.** A process variable $X : D \longrightarrow P$ is guarded in a process p if, and only if:

- If $p = \delta$ or $p = \alpha$, then X occurs guarded in p.
- If $p = p_1 + p_2$, $p_1||p_2$, $p_1|p_2$, $c \to p_1 <> p_2$, $p_1 << p_2$, then X occurs guarded in p_1 and p_2.
- If $p = \sum_{d:D} p_1$, $p_1^c t$, $c \longrightarrow p_1$, $t >> p_1$, $p_1 << t$, $\Gamma_C(p_1)$,$\nabla_V(p_1)$, $\Delta_B(p_1)$, $\rho_R(p_1)$, then X occurs guarded in p_1.
- If $p = p_1.p_2$, $p_1||\_p_2$, then X occurs guarded in p_1.

The RSP ensures that a recursive equation $X = \Psi X$ with, Ψ a process operator, can have only a single solution when X is guarded in Ψ. This is equivalent, in a way, to saying that Ψ can only have one fixed point.

The RSP is stated formally by the rule:

$$\frac{\{\Gamma \vdash X = \Psi X \quad Y = \Psi Y\}}{\Gamma \vdash X = Y} \quad \text{recursive specification principle}$$

5.4 Koomen's Fair Abstraction Rule

Koomen's fair abstraction rule (KFAR) will help us prove process equalities when using hiding operators. In order to do this in a correct way, we have to ensure that when hiding actions we are not modifying the process behavior. The property required is fairness. This is, the branches affected by the hiding operators must let other actions happen eventually. KFAR was an idea found implicitly in Koomen's signaling modeling [13], but was first explicitly presented as a rule in [4].

Two different versions exist, one for branching bisimulation, and another for weak bisimulation.

$$\frac{\Gamma \vdash X = i \cdot X + Y}{\Gamma \vdash \tau \cdot \tau_{\{i\}}(X) = \tau \cdot \tau_{\{i\}}(Y)} \quad \text{KFAR (branching bisimulation)}$$

$$\frac{\Gamma \vdash X = i \cdot X + Y}{\Gamma \vdash \tau_{\{i\}}(X) = \tau \cdot \tau_{\{i\}}(Y)} \quad \text{KFAR (weak bisimulation)}$$

Let us note that Groote-Mousavi's axioms for rooted branching and weak bisimulation imply these two KFAR rules.

5.5 Linearization, Greibach form and applications

A linear process equation (LPE) is a process equation that satisfies some "shape" conditions. With normalized forms like the one linear process equations have, analysis and manipulation becomes easier and is standardized.

In LPEs, the left-hand side variable is the only variable used in the right-hand side. This variable is always preceded by one action in the right-hand side, thus making the LPE a guarded recursive specification.

Definition 32 (Linear process equation)**.** An LPE is a recursive process equation with

the following shape:

$$X(d:D) = \sum_{i \in I} \sum_{e_i : E_i} c_i(d, e_i) \to \alpha_i(d, e_i) \cdot X(g_i(d, e_i)) + \sum_{j \in J} \sum_{e_j : E_j} c_j(d, e_j) \to \alpha_{\delta_j}(d, e_j)$$

Where $I, J \subset \mathbb{N}$, $\{E_k\}_{k \in K \subset \mathbb{N}}$ are sorts, $\{c_k\}_{k \in K \subset \mathbb{N}}$ are parameterized conditions, $\{\alpha_k\}_{k \in K \subset \mathbb{N}}$ are parameterized actions, $\{\alpha_{\delta_k}\}_{k \in K \subset \mathbb{N}}$ are parameterized actions or δ, and $\{g_k\}_{k \in K \subset \mathbb{N}}$. In most of the applications we will find linear equations were the α_{δ_k} terms are, in fact, δ.

Linearization is a multiple step process. Starting from a guarded recursive specification we transform it to a Greibach normal form, with a intermediate pre-Greibach normal form.

Because this process is quite cumbersome, we need to point out to [8] and [3] for details.

5.5.1 τ-priorization

Process parallelization usually produces a big space state explosion. When using parallel operators in combination to hiding operators, most transitions tend to be τ-transitions. To reduce unnecessary transitions where choices cannot be made because of close τ-transitions, we use τ-priorization. We will assume as hypothesis τ-confluence, that is, that processes contain no τ-loops.

For an LTS A, a τ-priorization of A is another LTS, of which transition set is given by a $A' \subseteq A$, where for every state s, if there is any transition $s \xrightarrow{\tau} s'$, any transition $s \xrightarrow{a} s''$ with $a \neq \tau$ is removed.

We have just presented these rules here as an example of "syntax oriented" transformation that can be applied to processes in order to obtain some "normal" forms so that semantical equivalences can be "easily" checked once these "simplified" forms are available.

Theorem 5.1. *Let A_1 and A_2 be LTSs. If A_2 is a τ-priorization of A_1, then A_1 and A_2 are branching bisimilar.*

6 Modeling time

6.1 Historical context

Dealing with time has been a problematic topic in the algebraic description of processes. On [2], Baeten and Bergstra discuss how to generalize the definition of ACP (Algebra of

Communicating Processes, a precursor of mCRL2), where they point out the necessity of a formal generalization and operational semantics for timed processes in order to be able to verify properties without limitations. Other proposals involve the use of discrete timings, as in [9] and [5]. As we can see, there is not an absolute consensus in whether use discrete time with stepped delays, which lacks the descriptive dynamism of an arbitrary slicing of time; or real valued time (which presents difficulties if no limit in precision is imposed in recursive processes with decreasing time margins).

6.2 From actions to timed actions

As we defined at the beginning of this work, actions are the atom of our processes, and thus take no time to execute. By adding a timestamp to actions, we will be stating that such actions must be performed at that exact point in time. Actions with a timestamp are denoted by $a^c t$, where t is the time in a dimensionless representation. The fact that timed actions must occur in an specific moment brings us to the main constraint of timed actions: there must be time coherence between the executed actions of a trace. This translates to invalid or incoherent traces that end in a deadlock state.

In mCRL2, timestamps following actions in timed actions represent time points in a globally scoped timeline. Therefore, we must ensure that we define correctly timed actions so no unintended deadlocks happen. One consequence of introducing this kind of timing system is that, because inconsistent traces end up in deadlocks, many of the possible traces produced by the combinatorial explosion of parallelized processes disappear, as many of these traces do not satisfy the requirements due to impossible reorderings of timed actions. As an example, we can see that, for actions $a^c 1$, $b^c 2$, and $c^c 3$, the only possible ordering is $a \cdot b \cdot c$ in their respective times, and that if we were to write the process $a^c 1 || b^c 2 || a^c 3$, the only proper trace would be the one we just wrote before.

Between timed actions, time must run. When time runs, we are not performing any action. This is what we call idling. Behaviorally, between two sequential timed actions (or multiactions) we consider that the system idles. When defining a system, we can specify that there can be idling at a given time. If there exists another action that must execute at that time, then two things might happen: we perform the action or we idle and skip that action.

6.3 Timed Labeled Transition Systems

To define timed LTS's, we only need to consider adding the two concepts mentioned before: time coherence and idling.

For time coherence, we will forbid that two sequential transitions execute action in reverse time order. This is the so called progress requirement, as time must strictly increase after each action.

$$s \xrightarrow{a}_t s' \xrightarrow{a'}_{t'} s'' \Rightarrow t < t'$$

For idling, we must correctly represent that if an action will occur at a given time, the system can idle until that time comes. Furthermore, if a system can idle up to a given time, it can obviously idle up to nearer points of time. This is called the density requirement, as the ability to idle is dense in time intervals.

To represent idling, a new relation is introduced, indicating with s $\rightsquigarrow_t$ that from state s we can idle up to time t. Note that this relation does not have a destination state, the state stays the same.

The transformation from an untimed LTS to a timed one is quite simple, we only have to introduce the progress and density requirements as follows: assign time 0 to the initial state of the LTS and give arbitrary positive time points to terminal states. Then assign times to each transition, giving increasing timestamps in each transition.

6.4 Behavioral equivalence revamped

To work with timed LTS's, we need to also extend the definitions of our behavioral equivalences. We will only indicate the slight differences and show the formal definitions.

6.4.1 Timed trace equivalence and timed weak trace equivalence

The definition of the Traces(s) set changes slightly. When an idle relation comes from a state, we must add the timestamp of the idling in the new TTraces (timed traces) set.

Definition 33 (Timed trace equivalence)**.** Let $A = (S, Act, \rightarrow, \rightsquigarrow, s_0, T)$ be a timed LTS. The set of timed traces starting from $s \in S$, $TTraces(s)$, is the minimal set that satisfies:

- $\varepsilon \in TTraces(s)$
- $\checkmark \in TTraces(s) \iff s \in T$
- $s \rightsquigarrow_t \Rightarrow t \in TTraces(s)$
- $s \xrightarrow{a}_t s' \wedge \sigma \in TTraces(s') \Rightarrow a_t^c \cdot \sigma \in TTraces(s)$

Definition 34 (Timed weak trace equivalence)**.** Let $A = (S, Act, \rightarrow, \rightsquigarrow, s_0, T)$ be a timed LTS. The set of timed weak traces starting from $s \in S$, $WTTraces(s)$, is the minimal set that satisfies:

- $\varepsilon \in WTTraces(s)$
- $\checkmark \in WTTraces(s) \iff s \in T$
- $s \rightsquigarrow_t \Rightarrow t \in WTTraces(s)$
- $s \xrightarrow{a}_t s', a_t^c \neq \tau \wedge \sigma \in WTTraces(s') \Rightarrow a_t^c \cdot \sigma \in WTTraces(s)$
- $s \xrightarrow{\tau}_t s' \wedge \sigma \in WTTraces(s') \Rightarrow \sigma \in WTraces(s)$

6.4.2 Timed strong and branching bisimulations

For timed strong bisimulation, only considering timestamps and idling will suffice, the definition does not change much more than that.

Definition 35 (Timed strong bisimulation)**.** Let $A = (S, Act, \rightarrow, \rightsquigarrow, s_0, T)$ be a timed LTS. The strong bisimulation is a symmetric binary relation $R \subseteq (S \times S)$ where $s, t \in S, sRt$ if, and only if:

- $s \xrightarrow{a} s' \Rightarrow \exists t', t \xrightarrow{a} t' \wedge s'Rt'$
- $s \in T \iff t \in T$

In the case of timed branching bisimulation, things get more complicated. If we recall the definition of branching bisimulation, a transition on one state must be simulated by a series of τ actions and a non-τ action in the other state. Here, we must make the timings of these intermediate τ transitions match with the concept of bisimilarity. For this, a timed branching bisimulation relation is parameterized by a time variable, representing a relation where related elements use that time as a starting referential point. On top of that, we must also add support for idling.

Definition 36 (Timed branching bisimulation)**.** Let $A = (S, Act, \rightarrow, \rightsquigarrow, s_0, T)$ be a timed LTS. For $t \in \Re\ t \geq 0$, t-timed branching bisimulation is a symmetric binary relation $R_t \subseteq (S \times S)$ where $s, u \in S, sRu$ if, and only if:

- If $s \xrightarrow{a}_{t'} s'$ with $t' > t$, then $\exists t = t_0 < t_1 < \cdots < t_n < t_{n+1} = t'$ such that either:
  - $a = \tau \wedge u = u_0 \xrightarrow{\tau}_{t_1} u_1 \xrightarrow{\tau}_{t_2} \ldots \xrightarrow{\tau}_{t_n} u_n \rightsquigarrow'_t, \forall i \in N, w \in \Re, 0 \leq i \leq n \wedge t_i \leq w < t_{i+1} \wedge sR_w u_i \wedge s'R_{t'} u_n$

- $a \neq \tau \wedge u = u_0 \xrightarrow{\tau}_{t_1} u_1 \xrightarrow{\tau}_{t_2} \ldots \xrightarrow{\tau}_{t_n} u_n \xrightarrow{a}_{t'} u', \forall i \in N, w \in \Re, 0 \leq i \leq n \wedge t_i \leq w < t_{i+1} \wedge sR_w u_i \wedge s'R_{t'}u'$
- $\exists t', t \xrightarrow{a} t' \wedge s'Rt'$

- If $s \rightsquigarrow_{t'}, \exists t = t_0 < t_1 < \cdots < t_n < t_{n+1} = t'$ such that $u = u_0 \xrightarrow{\tau}_{t_1} u_1 \xrightarrow{\tau}_{t_2} \ldots \xrightarrow{\tau}_{t_n} u_n \rightsquigarrow'_t, \forall i \in N, w \in \Re, 0 \leq i \leq n \wedge t_i \leq w < t_{i+1} \wedge sR_w u_i \wedge sR_{t'}u_n$

- $s \in T \iff u \in T$

6.5 Timed processes and timed formulas

The extension of timed processes is simple, we just need a way of labeling actions with their respective timestamps. For this, the *at* operator is introduced, and is written as we have already seen in timed actions. For equational definition of processes, the *at* operator cannot be used, as it can only be used in atomic actions. To represent that an equation has a time dependence, we must pass a time parameter as an argument of the parameterized equation. For this to be useful, timestamps must have the possibility of being variable and depend on a free variable.

Much like timed processes, we will use the *at* operator following actions to represent timestamps in μ-calculus formulas. Another additions to μ-calculus are the introduction of two new basic elements, *delay* and *yaled*, used to represent existence of (or lack thereof) idling. The *delay* operator represents that a delay occurs, meaning that no action is performed; its counterpart, yaled, expresses the concept that an action must occur, and that the system cannot idle indefinitely. Both elements can be followed by timestamps just like timed actions do.

7 mCRL2

mCRL2 is a specification language and tool for modeling and specification of concurrent systems. mCRL2 specification language is a one to one transformation from process algebra explained here, assigning operators and keywords for basic operations like choice and sequential composition and complex operations like parallel composition, hiding, sum, etc.

7.1 Language specification

7.1.1 mCRL2 process specification language

System specification is encapsulated in a single **.mcrl2** extension file.

A mCRL2 file usually starts with **sort** declarations. Sorts can be constructors without parameters, or general constructors with possibly multiple parameters, both with a preceding **struct** keyword. Sort declarations can act as aliases for other sorts, mainly mappings from sorts to sorts. Finally, sorts can be unspecified, with an incomplete declaration. Unspecified sorts can later be specified by declaring constants.

mCRL2 adopts a similar typing system as Haskell, but adding Cartesian products of sorts, denoted by a hash (#) separator.

Parameter variables need not be previously declared as variables, but must be free variables. Type self-reference is permitted in parameterized constructors. It is currently not possible to have type parameters for arbitrary sorts, with only the predefined List, Set, and Bag able to be parameterized with the contained sort. Here are some examples of sort definitions:

```
sort IP = struct ip1 | ip2;
sort Card = struct heart | tile | clover | pike;
sort Tree = struct leaf(Card) | node(Tree, Card, Tree);
sort Message = struct m(source:IP, destination:IP, content:List(Char));
```

Sorts can be followed by **map** declarations and **eqn** definitions.

Map declarations specify the type of the mapping. To define the behavior of mappings, equation definitions are required. Equation definitions have pattern matching capabilities, and have proven to be very useful with constructed sorts. Variables for equation definitions must be declared previously with a **var** statement. In the next example we can see how the accessors of a tree structure can be defined:

```
map left : Tree -> Tree;
```

```
var l, r : Tree;
    a : A;
eqn left(node(l,a,r)) = l;
map data = Tree -> Tree;
var l, r : Tree;
    a : A;
eqn data(node(l,a,r)) = a;
    data(leaf(a)) = a;
```

After specifying sorts and sort functions, we declare all actions that will appear in our specification. As some actions are parameterized, we must write the Cartesian product of the parameters the same way we declared the type of a map. Actions created as a result of communication operators have to be declared here as well. In the following example, the definition of some communication actions are defined:

```
act send, receive, communicate : Computer # Computer # Message;
    turn\_on, turn\_off : Computer;
```

Next, we will define all the required recursive specifications. We can use processes to modularize behavior inside a system. Process declaration is preceded by a **proc** keyword. Each process is defined by an equation. The left-hand side of the equation is composed by the process name and typed parameters. In the right-hand side we write a process using the algebraic operators. This next example shows how to recursively define a sender\receiver process:

```
proc Client(inputQueue : List(Message), outputQueue : List(Message)) =
     sum m:Message.recieve(m).Client(m |> inputQueue, outputQueue) +
     send(rhead(outputQueue)).Client(inputQueue, tail(outputQueue));
```

Finally, we have to declare the initial instance to verify. This part is preceded by the **init** keyword. Usually, the **init** part is composed by some processes in parallel, with a combination of communication, hiding and allow operators. This is a possible **init** definition:

```
init allow({communicate},
         comm({send|receive->communicate},
              Client([],[]) || Client([],[])
         )
     );
```

Processes are defined using process algebra operators. Process operators in mCRL2 are written as shown in figure 25.

Process algebra	mCRL2 operator	μ-calculus	mCRL2 expression
$P + Q$	P`+`Q	true	`true`
$P \cdot Q$	P`.`Q	false	`false`
δ	`delta`	$\mu X.\Phi_X$	`mu` X`.Phi`$_X$
$B \to P \diamond Q$	B `->` P `<>` Q	$\nu X.\Phi_X$	`nu` X`.Phi`$_X$
$\sum_{d:D} P$	`sum` d`:`D`.`P	$\forall d : D.\Phi_d$	`forall` d`:`D`.` Phi_d
$P \| \| Q$	P `||` Q	$\exists d : D.\Phi_d$	`exists` d`:`D`.` Phi_d
$P \| Q$	P `|` Q	$\Phi \Rightarrow \Psi$	Phi `=>` Psi
$\Gamma_C(P)$	`comm(`C`,`P`)`	$\Phi \vee \Psi$	Phi `||` Psi
$\nabla_{\{a_i\}}(P)$	`allow({`$a_1 \ldots a_n$`,`P`)`	$\Phi \wedge \Psi$	phi `&&` psi
$\partial_B(P)$	`block(`B`,`P`)`	$[a]\Phi$	`[`a`]` Phi
$\rho_R(P)$	`comm(`R`,` P`)`	$\langle a \rangle \Phi$	`<`a`>` Phi
$\tau_I(P)$	`hide(`I`,` P`)`	$\neg\Phi$	`!`Phi

Figure 25: mCRL2 textual representation.

7.1.2 mCRL2 μ-calculus language

To specify μ-calculus formulas, we must write separate files from our .mcrl2 files. Their extension does not matter, as the tool does not process them by themselves. Following the examples packaged with the tool, we will use the .mcf extension, which we presume means μ-calculus formula.

The syntax of a .mcf file is quite simple, the formula to be evaluated is written directly in μ-calculus syntax.

μ-calculus's written syntax is given in figure 25.

7.1.3 Other internal languages

The mCRL2 toolset has many utilities and transformations we can apply to our process modeling. Many of those transformations cover sections mentioned in this text, and that have their own syntax.

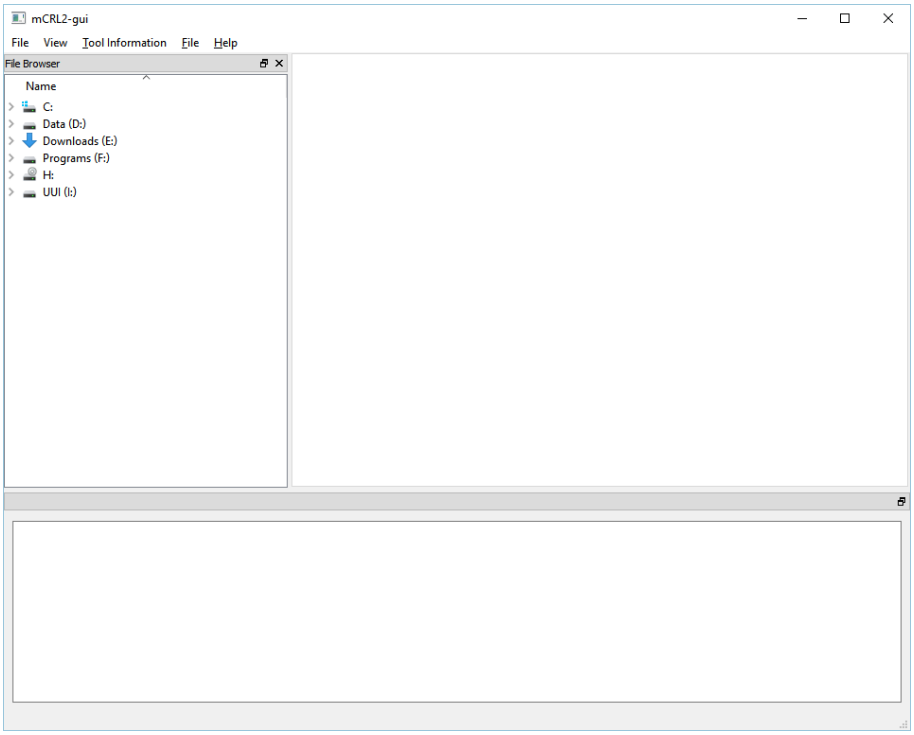
The only ones worth mentioning are the textual representation of (non-)parameterized boolean equations, which can be output in the last phase of model verification, that is, when we are combining both model specification and μ-calculus formula.

This transformation is automatically done by the toolset, and we have not needed to modify and of the produced .pbes files in our examples.

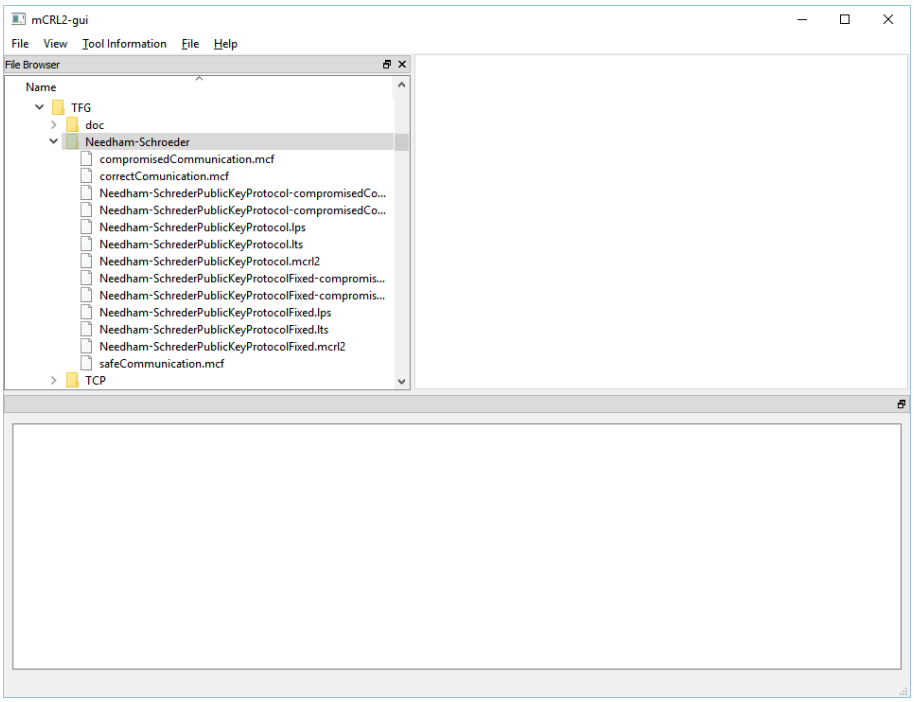
7.2 Using mCRL2

During the development of this project, we used the mCRL2 toolset over Windows with its original GUI. This has proven to facilitate the process of calling the programs that form the whole toolset. There are other releases for OSX and Linux-based systems, but in the following steps we will show how to validate and analyze in Windows. Workflow may vary in other platforms. We start by downloading the Windows installer from http://www.mcrl2.org/web/user_manual/download.html.

After installing, we open the program mcrl2-gui. This window will appear:



We must navigate through the file system to where our files reside.



We see there are two .mcrl2 files, *Needham-SchroederPublicKeyProtocol.mcrl2* and *Needham-SchroederPublicKeyProtocolFixed.mcrl2*. There are also two .mcf files, *compromisedCommunication.mcf* and *correctComunication.mcf*.

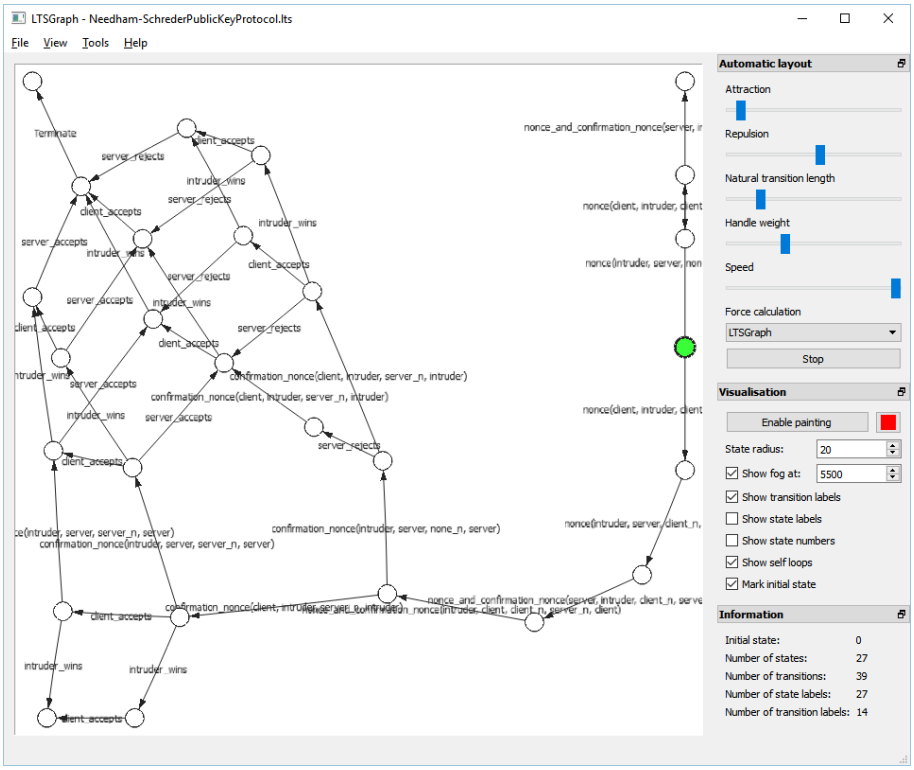
We will proceed to check that *Needham-SchroederPublicKeyProtocol.mcrl2* verifies the μ-calculus formula *compromisedCommunication.mcf*.

We start by right-clicking the .mcrl2 file, and click in the transformation `mcrl22lps`, which will transform our mCRL2 process model to its standard Grebach Linear Process equation System. A new tab will appear in the right section of the interface. We can see all the available options for this tool. In this case, we recommend to set the linearization method to "stack" for models that have an extremely large state-space unless the host computer lacks sufficient resources. Once run, a new file with extension .lps will be produced.

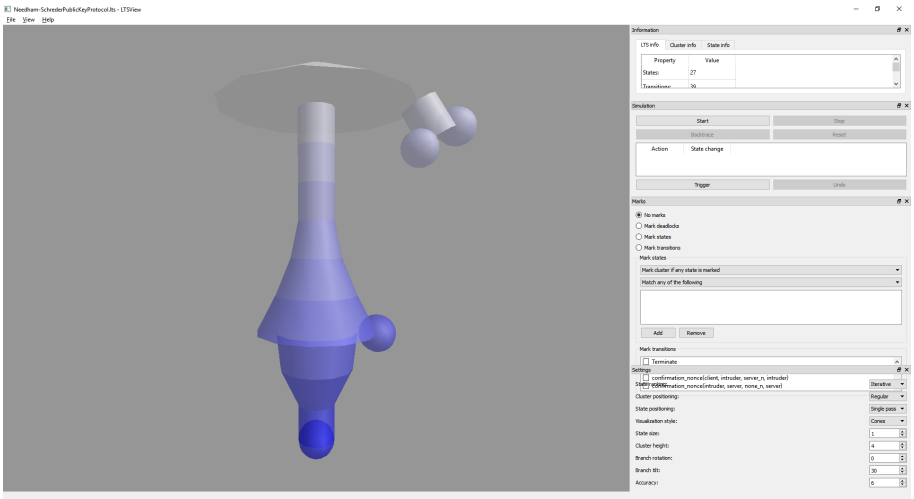
Right-clicking on the .lps file, we are given several options. In the analysis menu, we can get information about the LPS with the tool lpsinfo, which will show basic statistics about the structure of the LPS. Below, by using the command `lpspp` we can pretty-print the LPS back to mCRL2 language, this is useful if we want to see what is the produced Grebach form. In the transformation menu, we are given the choice to apply many transformations, many of which simplify and make more efficient further verification steps. Because most do not advance to the next step, we will only mention the ones we will use, `lps2lts` and `lps2pbes`. `lps2lts` transforms the linear process system to its equivalent labeled transition system. This transformation is useful because an LTS acts as a good visual representation of how a small system behaves. This tool produces a file with .lts extension. `lps2pbes` will be the tool we usually use in this case, transforming the linear process system to a behaviorally equivalent set of parameterized boolean equations. In this tool we use as input the μ-calculus formula as well, getting a .pbes file in return.

On a .pbes file, we can finally check if our model behaves like we would want it to. To do this, we right-click into the analysis menu, clicking in the tool `pbes2bool`.

On a .lts file, we can visualize the labeled transition system. There are a few tools for this, but the two most useful have been `ltsview` and `ltsgraph`. `ltsgraph` creates a 2D/3D image with nodes as circles and transitions as labeled arrows. This representation is quite useful as the tool lets us move nodes and distribute them around the canvas with "physics-based" algorithms. Nodes can be modified by color and size, and an option for 3D display can be chosen to enable better visualization of large but clustered LTS's. The graph can be exported to image and XML files for later use.



`ltsview` displays the LTS as a tubular tree shape, where sequential actions are represented by cylindrical columns and multiple transitions to different nodes are represented by conical trunks widening where most states are. Final states are represented with spheres, indication no outward transitions exist. In this tool we can select to mark deadlocks and terminating nodes, states and transitions, and walk action traces step by step, visualizing the states we traverse in the 3D figure.



The last tool we will use is `ltscompare`, which implements equivalence checking algorithms for the behavioral equivalences shown in this text. This tool will compare two .lts files, identifying if they are is any counter-example to its equivalence. The accepted behavioral equivalences are:

- Strong bisimilarity.
- Branching bisimilarity.
- Divergence-preserving branching bisimilarity.
- Weak bisimilarity.
- Divergence-preserving weak bisimilarity.
- Trace equivalence.
- Weak trance equivalence.
- Simulation equivalence (not discussed in this work).

Using this comparison tool, we can check correct behavioral implementation by checking against a model of the specification.

Appendices

In the following appendices, we show examples of modeling and verification using the mCRL2 toolset. Source code for the models and resulting files from the analyses can be found in the following GitHub repository: https://github.com/hhassan1/ConcurrentSystemsTFG. Finally, we will show the proof of μ-calculus monotonicity on evenly negated formulas, which ensures the existence of least and greatest fixed points.

A Needham-Schroeder protocol

The Needham-Schroeder protocol is an authentication protocol for potentially vulnerable networks. It relies on the security provided by encryption to safely send authentication keys through the network. There are two versions of the protocol, one based on symmetric encryption and one based on asymmetric public-private key encryption, both introduced by Roger Needham and Michael Schroeder at Xerox Palo Alto in the late 70's[15]. Unfortunately, for both versions of the protocol there were attacks that compromised the intended safety[7][14]. Be noted that, for the public-private key encryption version, a long 17 years passed until the error was found and a solution was given, showcasing the importance of automated verifications.

We will show how the public-private key version of Needham-Schroeder's protocol can be compromised by a man-in-the-middle attack, where the attacker (hereinafter the intruder) impersonates the originator of a communication presumed safe. After showing how it can be attacked, we will add the modifications needed to solve the error, as shown in [14], and prove that it now is, in fact, a safe protocol in regards to man-in-the-middle attacks.

A.1 Description of the protocol

The public-private key version of the Needham-Schroeder protocol defines a set of operations (similarly to an API), and an order in which they must be performed between two entities that wish to authenticate their identities. As the entity who starts the communication is the one which starts the protocol, from now on we will call this entity the client; and the other entity, which receives the starting protocol message, will be called the server. The idea of the protocol is sending nonces (unique arbitrary numbers) generated by the client and the server, using encryption to prevent prying eyes. The encrypted nonces are passed back and forth once, so that both client and server know that the other party is who it says to be. This is possible because, assuming the unlikelihood of a coincidence of

nonces, an intruder would need to decrypt the messages and return the nonces to their respective entities, which would imply a breach in the encryption algorithm.

A.2 Attack pattern and how to fix it

The intruder will perform a man-in-the-middle attack (i.e. the intruder acts as an intermediary between the client and the server). Once the client has connected to the intruder (by error or with the help of malware or phishing), the intruder then would impersonate the server with the client's connection and vice-versa. Because the client made a connection with the intruder, the intruder is able to decrypt the nonce of the client and send it to the server; but it will not be able to decrypt the incoming nonce of the server, as it is encrypted with the public key of the client. By passing the encrypted message to the client and getting the confirmation back, the intruder is now able do decrypt and get the server's nonce, thus successfully completing the attack.

The solution is quite simple: the nonce that is sent by the server must have appended the servers identification. Because this message is encrypted with the client's key, the intruder is not able to modify it. The client would then notice the difference between the identification of its connection (the intruder) and the message it has received.

A.3 Modeling and verification

This appendix's files are located in the Needham-Schroeder folder. The files named `‘‘Needham-SchroederPublicKeyProtocol’’` refer to the original version of the protocol, whereas `‘‘Needham-SchroederPublicKeyProtocolFixed’’` refer to the fixed protocol. Two μ-calculus formulas are provided: `compromisedCommunication`, which checks if the intruder can compromise the authentication, and `‘‘correctCommunication’’`, which checks that both originator and destination of the communications have, as far as they individually know, succeeded on authenticating.

In our model, we ignore the existence of an authentication server and assume that the name lookup has already been done in some way. This is valid, as the underlying problem of the protocol does not involve the intervention of the authentication server. Another important paradigm applied is to try to build a solution-agnostically model, that is, without biasing the model to produce the known solution. For this, the intruder is modeled as a process that can perform the operations defined by the protocol, but without any implicit order, thus allowing a bigger and more realistic generated state-space. The only restrictions in which the intruder is constrained is that it can only perform communications to the destinations that make sense. Also, encrypted data that cannot be decrypted by the

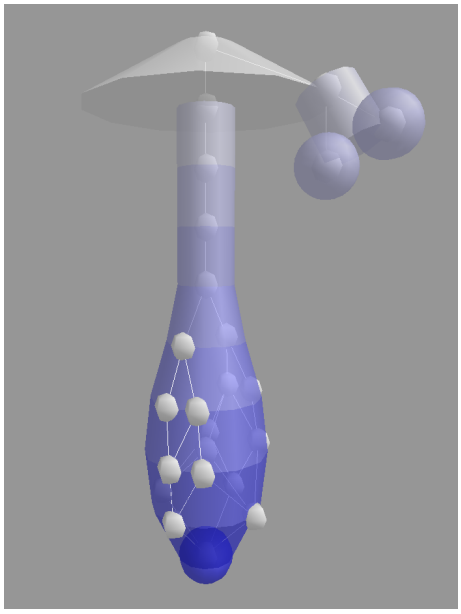


Figure 26: Original LTS in hierarchical form

intruder is only passed through, thus preserving the safety provided by the encryption algorithm.

After linearising the process specification, we can produce its LTS. We can see that the state-space generated is quite small: only 24 states and 33 transitions, with 3 deadlocks, 2 of which represent a failure in the communication at the start (this is the case when the intruder communicates with the server before the client has sent any information to the intruder). There is only one trace where the intruder wins (modulo ordering of the actions we use to show correct protocol communication).

Because this problem has such a small state-space, further simplification using behavioral equivalences is not needed.

Applying the μ-formula that checks if the intruder can win, we find that, as we expected, the formula is true. The tool also gives us the justification, in terms of the satisfied parameterized boolean equation that deduce the result. These formulas are harder to read than the actions they represent, but if the state-space is small one can use the `ltsgraph` tool with the "Show state labels" option and follow this trace by hand.

Now, using the model of the fixed version of Needham-Schoreder's protocol, we get a slightly smaller LTS, with only 13 states and 14 transitions. We can simulate the execution with LTSView and in fact we can see that there are only 3 traces (modulo reordering of actions), one in which client and server reject the authentication, and other two where the intruder fails to use the protocol API correctly. Therefore, we can conclude that the addition of the server's identity on the confirmation message sent by the client.

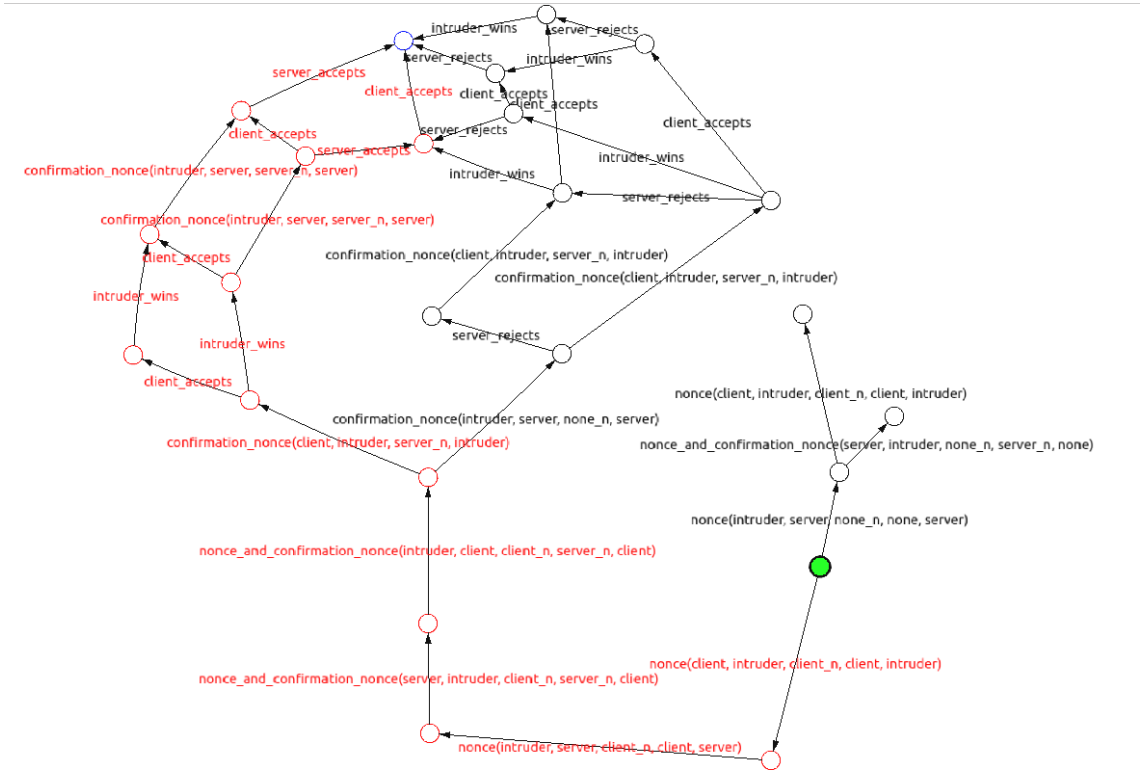


Figure 27: Original LTS with attack trace in red

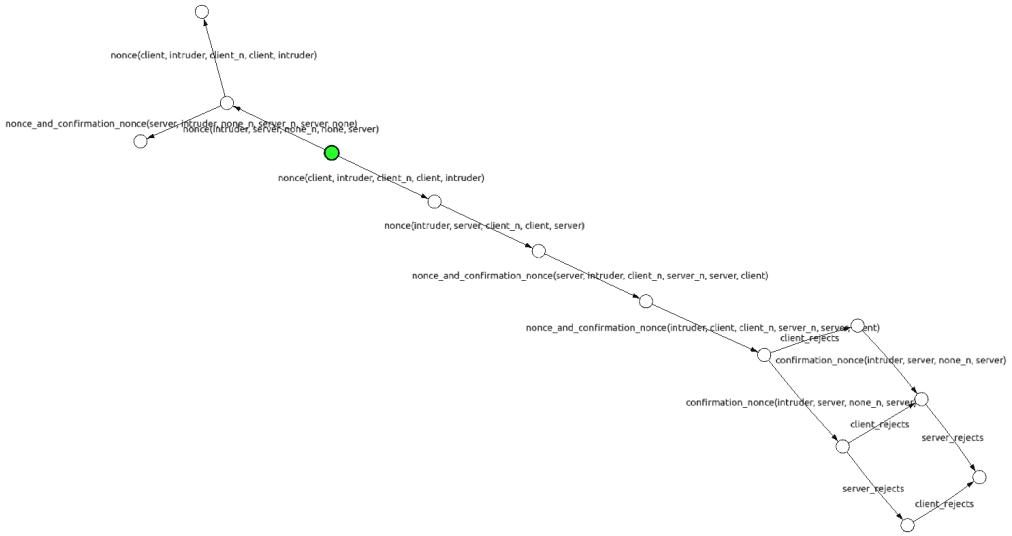


Figure 28: Fixed LTS with no attack trace

```
1: X(1,client,client,client_n,client_n,client,none_n,none_n,none,false,1,client,
    client_n,client,intruder,1,client,client,client_n,client_n,client,client)
4: Subst:true X(1,client,client,client_n,client_n,client,client_n,none_n,client,
    false,2,client,client_n,client,intruder,1,client,client,client_n,client_n,
    client,client)
5: Subst:true X(1,client,client,client_n,client_n,client,client_n,none_n,client,
    false,2,client,client_n,client,intruder,2,server,intruder,client_n,server_n,
    client,intruder)
8: Subst:true X(2,intruder,client,client_n,server_n,client,client_n,none_n,client
    ,false,2,client,client_n,client,intruder,3,client,client,client_n,client_n,
    client,intruder)
9: Subst:true X(1,client,client,client_n,client_n,client,client_n,none_n,client,
    false,3,client,server_n,intruder,intruder,3,client,client,client_n,client_n,
    client,intruder)
11: Subst:true X(1,client,client,client_n,client_n,client,client_n,server_n,
    client,false,4,client,client_n,client,client,3,client,client,client_n,client_
    n,client,intruder)
```

Figure 29: Attack trace result of applying ‘‘`compromisedCommunication`’’

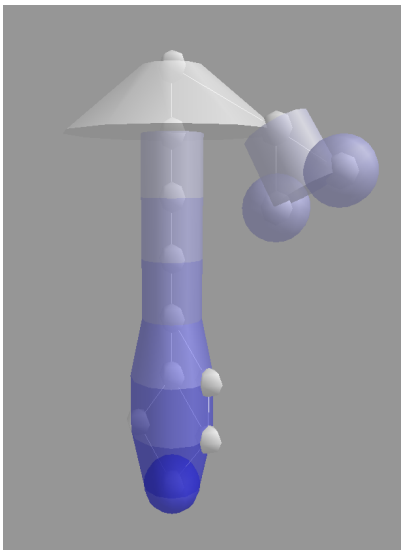


Figure 30: Fixed LTS in hierarchical form

After applying the previous μ-formula we obtain a negative result.

B TCP timed modeling

The Transmission Control Protocol is one of the most used protocols on the Internet. It ensures transmission of data preventing data loss and preserving the order in which the receiving end get the information. In order to ensure this properties, the protocol defines a series of procedures that both parties of a communication must apply. These procedures encompass retransmission of data and retransmission of acknowledgement messages, with the use of timers and a strict timeout standard.

Here, we model an abstraction of the Transmission Control Protocol, with the purpose of showing that no packet of data is lost, and that every packet read by the receiver is correctly ordered. We focused on the usage of timers and modeling them with our timed process algebra. Due to limitations on our computer, the verification of these conditions was not possible, as it seemed to make no progress on the phase of solving the parameterized boolean equation system. Nevertheless, the model is represented by a valid mCRL2 file and could be verified with the appropiate equipment.

The files of this model are located on the TCP folder.

C Mutual exclusion: Tie-Breaker and Ticket algorithms

In concurrent programming, parallel algorithms are used to improve performance over sequential implementations. A parallel algorithm runs over multiple threads of execution, sharing memory locations as a way to communicate between one another. Because totally parallelized memory accesses are not deterministic, we need a way to prevent parallelization of some parts of the code: the problem of mutual exclusion. A lock is a mechanism that limits access to some parts of code. Once a thread has locked a lock, no other thread may lock it, thus stopping until the lock has been unlocked. There are many ways of implementing locks, the most commonly used nowadays based on hardware-supported operations. Here we will model and verify mutual exclusion on the traditional implementations of locks, mainly the tie-breaker and ticket algorithms.

C.1 Tie-breaker algorithm

The tie-breaker, as defined for two threads, consists on an "in" flag for each thread and a shared "last" variable, which indicates which thread entered the locking procedure last. The two threads mark themselves as last threads to enter the lock, and actively wait until the they are the other thread has started the locking algorithm or the other thread's "in" flag is down. Once this condition is satisfied, the thread is ensured to have exclusive execution of the following code until its "in" flag is disabled again.

For modeling this algorithm, the pseudocode from [1] was used. While verifying the model, a flaw was found in the book's pseudocode: the order in which the lock sets up the "in" and "last" variables changes the behavior of the lock. In the book, the "in" flag is enabled and then the "last" variable is updated. When doing this, the lock does not ensure mutual exclusion, because the order in which the active wait checks the two conditions must not be the same as these assignments. Using the tools we are able to identify the trace of execution that led to the unsatisfied exclusiveness property and figure out a simple way to fix it.

The files of this appendix are located on the "Tie-breaker" folder. Here we have two models: the model as it was originally defined in [16], in the file "tiebreaker-peterson"; and the flawed model based on the pseudocode from [1], in the file "tiebreaker-andrews". We additionally have two μ-formulas to check, one for exclusiveness and one for fairness. We will show that this algorithm is only fair when using a weakly fair scheduler.

When applying the exclusiveness μ-formula to the flawed LTS, we get the following counterexample:

```
1: X1(1,c1,c1,c1,c1,c1,c1,true,c1,true,1,c1,c1,c1,c1,c1,c1,true,c1,true,
   false,false,c1)
4: Subst:false   X(1,c1,c1,c1,c1,c1,c1,true,c1,true,1,c1,c1,c1,c1,c1,c1,
   true,c1,true,false,false,c1)
6: Subst:false   X(2,c1,c1,c1,c1,c1,c1,true,c1,true,1,c1,c1,c1,c1,c1,c1,
   true,c1,true,false,false,c1)
9: Subst:false   X(2,c1,c1,c1,c1,c1,c1,true,c1,true,2,c1,c1,c1,c1,c1,c1,
   true,c2,true,false,false,c2)
19: Subst:false   X(2,c1,c1,c1,c1,c1,c1,true,c1,true,3,c1,c1,c1,c1,c1,c1,
   true,c1,true,false,true,c2)
32: Subst:false   X(2,c1,c1,c1,c1,c1,c1,true,c1,true,4,c1,c1,c1,c1,c1,c1,
   false,c1,true,false,true,c2)
50: Subst:false   X(2,c1,c1,c1,c1,c1,c1,true,c1,true,5,c1,c1,c1,c2,c2,c2,
   true,c1,false,false,true,c2)
80: Subst:false   X(3,c1,c1,c1,c1,c1,c1,true,c1,true,5,c1,c1,c1,c2,c2,c2,
   true,c1,false,true,true,c2)
114: Subst:false   X(4,c1,c1,c1,c1,c1,c1,true,c1,true,5,c1,c1,c1,c2,c2,c2,
   true,c1,false,true,true,c2)
147: Subst:false   Y(4,c1,c1,c1,c1,c1,c1,true,c1,true,6,c1,c2,c2,c1,c1,c1,
   true,c1,false,true,true,c2)
180: Subst:false   Y(5,c1,c1,c1,c1,c1,c1,true,c1,false,6,c1,c2,c2,c1,c1,c1,
   true,c1,false,true,true,c2)
```

Figure 31: Counterexample of exclusiveness on Andrews's Tie-breaker implementation

Now we can look for a trace where the actions `lock(c1)` and `lock(c2)` happen one after the other. After applying divergence-preserving branching bisimulation on the LTS, we find that a trace that satisfies this is:

```
writeTurn(c1).writeTurn(c2).write(c2,true).read(c1,false).write(c1,true).
   readTurn(c2).read(c2,true).readTurn(c2).lock(c1).lock(c2)
```

Once again, on the correction of the algorithm the exclusiveness formula is true, as can be checked using the toolset.

C.2 Ticket algorithm

The ticket algorithm relies on ordering the accesses to the lock, giving a "ticket" with a number to each thread that starts the locking mechanism. The algorithm also needs a

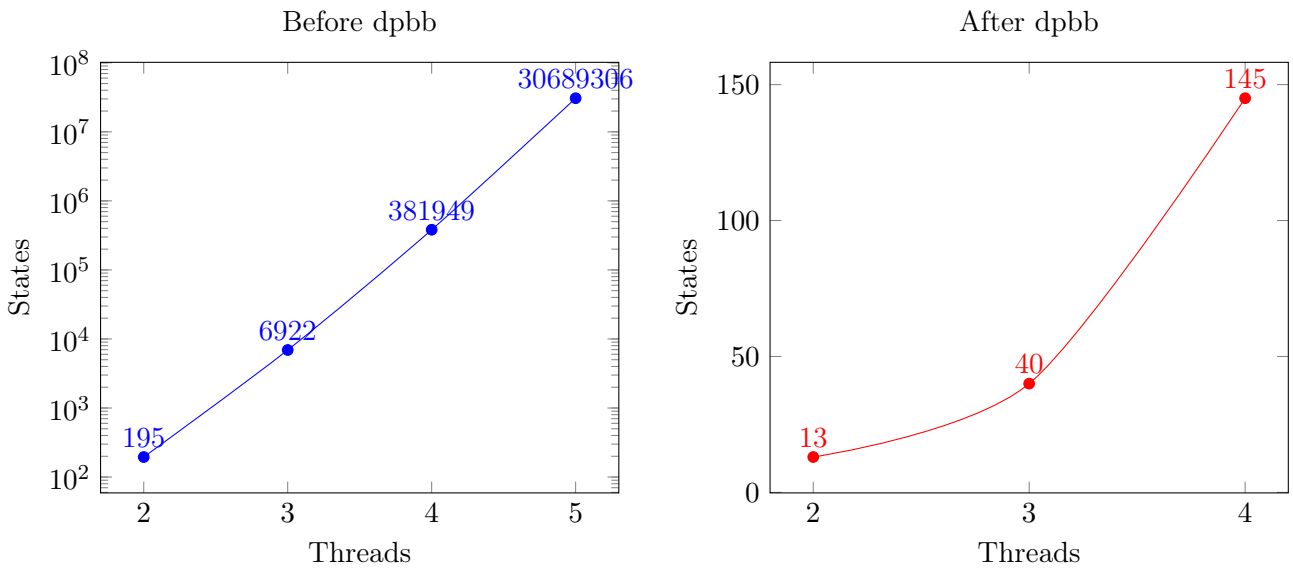


Figure 32: States in the ticket algorithm before and after divergence-preserving branching bisimulation

variable to hold the current turn in the sequence of tickets: the thread with a ticket equal to the turn is the one which will complete the locking process.

To achieve an ordered sequence of tickets, an atomic way of increasing a variable is needed. This is possible by using another lock, but that solution is highly inefficient. We instead will use a hardware-supported atomic operation: fetch-and-add. The fetch-and-add operation (FA) atomically returns the value of a variable and increases its value by one. After obtaining a ticket, a thread actively waits until it is its turn. After completing the locked code, the thread just needs to increase the value of the turn variable, so that the next thread can enter the exclusive section.

As with the previous lock, the pseudocode the pseudocode from [1] was used. In this case, after checking the μ-formula for exclusiveness, an affirmative response was given, meaning that this implementation is correct.

The files of this model are located on the "Ticket" folder.

While making the model, several values for the number of threads were tested. As one would expect, the more threads we have, the larger the resultant state-space gets. In figure 32 a chart of the states of the LTS versus the number of threads is given. As we can seen, the growth seems to be around two orders of magnitude for each new thread, indicating a exponential dependency between the size of the LTS and the number of threads.

To take into account, for two and three threads, it took the tool less than a second to produce the LTS; for four threads, it took about half an hour; and for five threads, it took

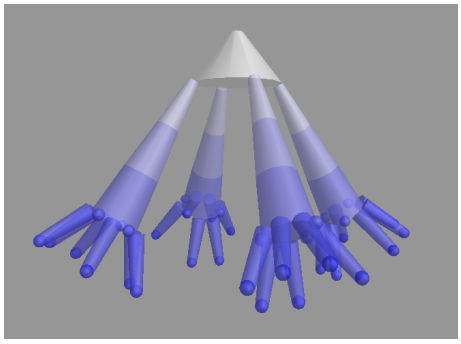


Figure 33: LTS for n=4 in ticket after divergence preserving branching bisimulation

seven hour. The last LTS couldn´t even be saved, as the computer ran out of its 16GB of RAM before it could be written to memory.

Not minding the original size of the LTSs, we will consider the size of their divergence-preserving branching bisimulating. In figure 33 we can see that the number of states decreases considerably. Unfortunately, the LTS for five threads could not be saved because of hardware limitations.

D Dining philosopher's problem

The dining philosopher's problem is a classic problem in the study of concurrent systems and deadlocks. At a dining table, five philosophers are sitting wanting to eat, but there are only five forks available. For this problem we have to assign the forks in a manner such that every philosopher can eat and no "deadlock" situation occurs.

The solution is quite simple, every philosopher grabs the fork on his right side and then the fork on the left side, except for the last philosopher, who does this reversing the order.

The files of this model are located on the "Philosophers" folder.

The resulting LTS has 2164 states, which decrease to 532 when applying divergence-preserving branching bisimulation, and further decreasing to 32 once weak trace equivalence is used. As we can see in figure 34, the weak trace equivalent LTS has a ovoid shape with six levels, each representing how many philosophers have eaten and have yet to think.

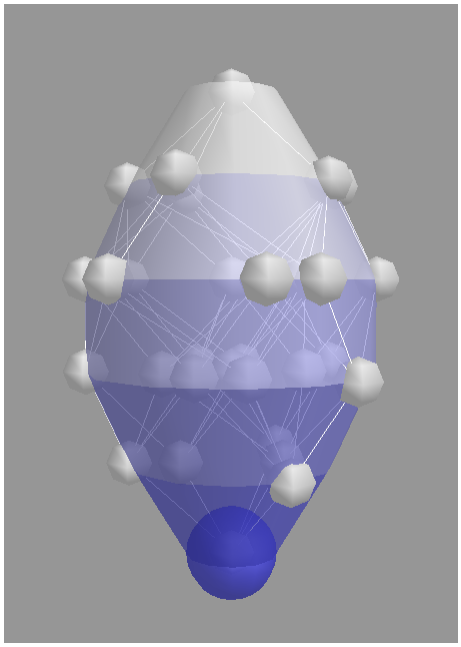


Figure 34: LTS for the philosophers problem after weak trace equivalence

E μ-calculus fixed point existence

In order to soundly use the μ-calculus, we must ensure that our formulas are well defined. For the modalities of μ-calculus, this means proving that evenly negated formulas are monotonically increasing. If that is the case, the Knaster-Tarski theorem ensures that, for a complete lattice and a monotonic function (i.e. the subsets of states ordered by size and the semantic function defined below) the set of fixed points of that function is also a complete lattice.

In [12], the semantics of μ-calculus are given. In the same document, Keiren indicates we only have to prove that the function Φ_η is monotonic. We will use this definition of the semantics with a slight change in notation.

Let Var be the set of variables used in fixed point modalities. Let L be an LTS, and S the set of states of L. Let $\eta : Var \to \mathcal{P}S$ the environment function, which assigns a set of states to a variable. Let $[X \to S'] \in (Var \to \mathcal{P}S) \to (Var \to \mathcal{P}S)$ be a function rewriting that redefines to S' the value of its argument when applied to X. The semantics of μ-calculus are as follows:

$$
\begin{aligned}
[\![true]\!]^\eta_L &= S \\
[\![false]\!]^\eta_L &= \emptyset \\
[\![X]\!]^\eta_L &= \eta(X) \\
[\![\neg\varphi]\!]^\eta_L &= S \setminus [\![\varphi]\!]^\eta_L \\
[\![\varphi \wedge \psi]\!]^\eta_L &= [\![\varphi]\!]^\eta_L \cap [\![\psi]\!]^\eta_L \\
[\![\varphi \vee \psi]\!]^\eta_L &= [\![\varphi]\!]^\eta_L \cup [\![\psi]\!]^\eta_L \\
[\![\varphi \implies \psi]\!]^\eta_L &= (S \setminus [\![\varphi]\!]^\eta_L) \cup [\![\psi]\!]^\eta_L \\
[\![\langle A \rangle \varphi]\!]^\eta_L &= \{s \in S | \exists_{t \in S, a \in A} s \to t \wedge t \in [\![\varphi]\!]^\eta_L\} \\
[\![[A] \varphi]\!]^\eta_L &= \{s \in S | \forall_{t \in S, a \in A} s \to t \implies t \in [\![\varphi]\!]^\eta_L\} \\
[\![\mu X.\varphi]\!]^\eta_L &= \mu T \subseteq S.\Phi^{X,\varphi}_\eta(T) \\
[\![\nu X.\varphi]\!]^\eta_L &= \nu T \subseteq S.\Phi^{X,\varphi}_\eta(T)
\end{aligned}
$$

where $\Phi^{X,\varphi}_\eta(T) : \mathcal{P}(S) \to \mathcal{P}(S) \coloneqq [\![\varphi]\!]^{\eta[X \to T]}_L$.

Let us show $\Phi^{X,\varphi}_\eta(T)$ is monotonically increasing for evenly negated φ's and monotonically decreasing for oddly negated φ's. Let us proceed by structural induction over the μ-calculus expression tree.

Proof. $\Phi_\eta^{X,\varphi}(T)$ monotony.

Let $\mathcal{P}(S)$ be a lattice ordered by the relation of $\subseteq$. Suppose T, $R \in \mathcal{P}(S)$ such that $T \leq R$:

- If $\varphi = true$, which is evenly negated:
  - $\Phi_\eta^{X,true}(T) = [\![true]\!]_L^{\eta[X\to T]} = S \leq S = [\![true]\!]_L^{\eta[X\to R]} = \Phi_\eta^{X,true}(R)$
- If $\varphi = false$, which is evenly negated:
  - $\Phi_\eta^{X,false}(T) = [\![false]\!]_L^{\eta[X\to T]} = \emptyset \leq \emptyset = [\![false]\!]_L^{\eta[X\to R]} = \Phi_\eta^{X,false}(R)$
- If $\varphi = X$, which is evenly negated:
  - $\Phi_\eta^{X,X}(T) = [\![X]\!]_L^{\eta[X\to T]} = \eta[X \to T](X) = T \leq R = \eta[X \to R](X) = [\![X]\!]_L^{\eta[X\to R]} = \Phi_\eta^{X,X}(R)$
- If $\varphi = Y \neq X$, which is evenly negated:
  - $\Phi_\eta^{X,Y}(T) = [\![Y]\!]_L^{\eta[X\to T]} = \eta[X \to T](Y) = Y \leq Y = \eta[X \to R](Y) = [\![Y]\!]_L^{\eta[X\to R]} = \Phi_\eta^{X,Y}(R)$
- If $\varphi = \neg\psi$:
  - If φ is evenly negated, and thus ψ oddly negated:
    - $\Phi_\eta^{X,\neg\psi}(T) = [\![\neg\psi]\!]_L^{\eta[X\to T]} = S \setminus [\![\psi]\!]_L^{\eta[X\to T]} \leq S \setminus [\![\psi]\!]_L^{\eta[X\to R]} = [\![\neg\psi]\!]_L^{\eta[X\to R]} = \Phi_\eta^{X,\neg\psi}(R)$
  - If φ is oddly negated, and thus ψ evenly negated:
    - $\Phi_\eta^{X,\neg\psi}(T) = [\![\neg\psi]\!]_L^{\eta[X\to T]} = S \setminus [\![\psi]\!]_L^{\eta[X\to T]} \geq S \setminus [\![\psi]\!]_L^{\eta[X\to R]} = [\![\neg\psi]\!]_L^{\eta[X\to R]} = \Phi_\eta^{X,\neg\psi}(R)$
- If $\varphi = \phi \wedge \psi$:
  - If φ is evenly negated, and thus ϕ and ψ evenly negated:
    - $\Phi_\eta^{X,\phi\wedge\psi}(T) = [\![\phi \wedge \psi]\!]_L^{\eta[X\to T]} = [\![\phi]\!]_L^{\eta[X\to T]} \cap [\![\psi]\!]_L^{\eta[X\to T]} \leq [\![\phi]\!]_L^{\eta[X\to R]} \cap [\![\psi]\!]_L^{\eta[X\to R]} = [\![\phi \wedge \psi]\!]_L^{\eta[X\to R]} = \Phi_\eta^{X,\phi\wedge\psi}(R)$
  - If φ is oddly negated, and thus ϕ and ψ oddly negated:
    - $\Phi_\eta^{X,\phi\wedge\psi}(T) = [\![\phi \wedge \psi]\!]_L^{\eta[X\to T]} = [\![\phi]\!]_L^{\eta[X\to T]} \cap [\![\psi]\!]_L^{\eta[X\to T]} \geq [\![\phi]\!]_L^{\eta[X\to R]} \cap [\![\psi]\!]_L^{\eta[X\to R]} = [\![\phi \wedge \psi]\!]_L^{\eta[X\to R]} = \Phi_\eta^{X,\phi\wedge\psi}(R)$
- If $\varphi = \phi \vee \psi$:
  - If φ is evenly negated, and thus ϕ and ψ evenly negated:

* $\Phi_\eta^{X,\phi\vee\psi}(T) = [\![\phi \vee \psi]\!]_L^{\eta[X\to T]} = [\![\phi]\!]_L^{\eta[X\to T]} \cap [\![\psi]\!]_L^{\eta[X\to T]} \leq [\![\phi]\!]_L^{\eta[X\to R]} \cap [\![\psi]\!]_L^{\eta[X\to R]} = [\![\phi \vee \psi]\!]_L^{\eta[X\to R]} = \Phi_\eta^{X,\phi\vee\psi}(R)$

– If φ is oddly negated, and thus ϕ and ψ oddly negated:

* $\Phi_\eta^{X,\phi\vee\psi}(T) = [\![\phi \vee \psi]\!]_L^{\eta[X\to T]} = [\![\phi]\!]_L^{\eta[X\to T]} \cap [\![\psi]\!]_L^{\eta[X\to T]} \geq [\![\phi]\!]_L^{\eta[X\to R]} \cap [\![\psi]\!]_L^{\eta[X\to R]} = [\![\phi \vee \psi]\!]_L^{\eta[X\to R]} = \Phi_\eta^{X,\phi\vee\psi}(R)$

- If $\varphi = \langle A\rangle\psi$:

– If φ is evenly negated, and thus ψ evenly negated:

* $\Phi_\eta^{X,\langle A\rangle\psi}(T) = [\![\langle A\rangle\psi]\!]_L^{\eta[X\to T]} = \{s \in S | \exists_{t\in S, a\in A} s \to t \wedge t \in [\![\psi]\!]_L^{\eta[X\to T]}\} \leq \{s \in S | \exists_{t\in S, a\in A} s \to t \wedge t \in [\![\psi]\!]_L^{\eta[X\to R]}\} = [\![\langle A\rangle\psi]\!]_L^{\eta[X\to R]} = \Phi_\eta^{X,\langle A\rangle\psi}(R)$

– If φ is oddly negated, and thus ψ oddly negated:

* $\Phi_\eta^{X,\langle A\rangle\psi}(T) = [\![\langle A\rangle\psi]\!]_L^{\eta[X\to T]} = \{s \in S | \exists_{t\in S, a\in A} s \to t \wedge t \in [\![\psi]\!]_L^{\eta[X\to T]}\} \geq \{s \in S | \exists_{t\in S, a\in A} s \to t \wedge t \in [\![\psi]\!]_L^{\eta[X\to R]}\} = [\![\langle A\rangle\psi]\!]_L^{\eta[X\to R]} = \Phi_\eta^{X,\langle A\rangle\psi}(R)$

- If $\varphi = [A]\psi$:

– If φ is evenly negated, and thus ψ evenly negated:

* $\Phi_\eta^{X,[A]\psi}(T) = [\![[A]\psi]\!]_L^{\eta[X\to T]} = \{s \in S | \forall_{t\in S, a\in A} s \to t \Longrightarrow t \in [\![\psi]\!]_L^{\eta[X\to T]}\} \leq \{s \in S | \forall_{t\in S, a\in A} s \to t \Longrightarrow t \in [\![\psi]\!]_L^{\eta[X\to R]}\} = [\![[A]\psi]\!]_L^{\eta[X\to R]} = \Phi_\eta^{X,[A]\psi}(R)$

– If φ is oddly negated, and thus ψ oddly negated:

* $\Phi_\eta^{X,[A]\psi}(T) = [\![[A]\psi]\!]_L^{\eta[X\to T]} = \{s \in S | \forall_{t\in S, a\in A} s \to t \Longrightarrow t \in [\![\psi]\!]_L^{\eta[X\to T]}\} \geq \{s \in S | \forall_{t\in S, a\in A} s \to t \Longrightarrow t \in [\![\psi]\!]_L^{\eta[X\to R]}\} = [\![[A]\psi]\!]_L^{\eta[X\to R]} = \Phi_\eta^{X,[A]\psi}(R)$

- If $\varphi = \mu Y.\psi$, which is evenly negated:

– If φ is evenly negated, and thus ψ evenly negated:

* $\Phi_\eta^{X,\mu Y.\psi}(T) = [\![\mu Y.\psi]\!]_L^{\eta[X\to T]} = \mu U \subseteq S.\Phi^{Y,\psi}_{\eta[X\to T]}(U) \leq \mu U \subseteq S.\Phi^{Y,\psi}_{\eta[X\to R]}(U) = [\![\mu Y.\psi]\!]_L^{\eta[X\to R]} = \Phi_\eta^{X,\mu Y.\psi}(R)$

– If φ is oddly negated, and thus ψ oddly negated:

* $\Phi_\eta^{X,\mu Y.\psi}(T) = [\![\mu Y.\psi]\!]_L^{\eta[X\to T]} = \mu U \subseteq S.\Phi^{Y,\psi}_{\eta[X\to T]}(U) \geq \mu U \subseteq S.\Phi^{Y,\psi}_{\eta[X\to R]}(U) = [\![\mu Y.\psi]\!]_L^{\eta[X\to R]} = \Phi_\eta^{X,\mu Y.\psi}(R)$

- If $\varphi = \nu Y.\psi$, which is evenly negated:

– If φ is evenly negated, and thus ψ evenly negated:

* $\Phi_\eta^{X,\mu Y.\psi}(T) = [\![\nu Y.\psi]\!]_L^{\eta[X\to T]} = \nu U \subseteq S.\Phi^{Y,\psi}_{\eta[X\to T]}(U) \leq \nu U \subseteq S.\Phi^{Y,\psi}_{\eta[X\to R]}(U) = [\![\nu Y.\psi]\!]_L^{\eta[X\to R]} = \Phi_\eta^{X,\nu Y.\psi}(R)$

– If φ is oddly negated, and thus ψ oddly negated:

$$* \ \Phi_{\eta}^{X,\mu Y.\psi}(T) = [\![\nu Y.\psi]\!]_L^{\eta[X\to T]} = \nu U \subseteq S.\Phi^{Y,\psi}_{\eta[X\to T]}(U) \geq$$
$$\nu U \subseteq S.\Phi^{Y,\psi}_{\eta[X\to R]}(U) = [\![\nu Y.\psi]\!]_L^{\eta[X\to R]} = \Phi_{\eta}^{X,\nu Y.\psi}(R)$$

To be strictly correct, we must prove that if $\forall X, Y \neq X \in Var, \forall \psi, \forall U, T, R \subseteq S$ if $T \subseteq R$, then $\Phi^{Y,\psi}_{\eta[X\to T]}(U) \subseteq \Phi^{Y,\psi}_{\eta[X\to R]}(U)$. This is in fact true, and we could adapt the previous structural induction to prove that, extending the lattice order to functions, we have that if $\eta \leq \chi$, then $\Phi_{\eta}^{Y,\psi}(U) \subseteq \Phi_{\chi}^{Y,\psi}(U)$. $\square$

References

[1] G.R. Andrews, *Foundations of multithreaded, parallel, and distributed programming*, Addison-Wesley, 2000.

[2] J. C. M. Baeten and J. A. Bergstra, *Real time process algebra*, Formal Aspects of Computing **3** (1991), no. 2, 142–188.

[3] J. C. M. Baeten, J. A. Bergstra, and J. W. Klop, *Decidability of bisimulation equivalence for process generating context-free languages*, J. ACM **40** (July 1993), no. 3, 653–682.

[4] J.C.M. Baeten, J.A. Bergstra, and J.W. Klop, *On the consistency of koomen's fair abstraction rule*, Theoretical Computer Science **51** (1987), no. 1, 129 –176.

[5] Jos CM Baeten and Jan A Bergstra, *Discrete time process algebra*, Formal Aspects of Computing **8** (1996), no. 2, 188–208.

[6] David de Frutos-Escrig, Carlos Gregorio-Rodríguez, Miguel Palomino, and David Romero-Hernández, *Unifying the linear time-branching time spectrum of process semantics*, Logical Methods in Computer Science **9** (2013), no. 2.

[7] Dorothy E. Denning and Giovanni Maria Sacco, *Timestamps in key distribution protocols*, Commun. ACM **24** (August 1981), no. 8, 533–536.

[8] Jan Friso Groote and Mohammad Reza Mousavi, *Modeling and analysis of communicating systems*, The MIT Press, 2014.

[9] Matthew Hennessy and Tim Regan, *A process algebra for timed systems*, Information and computation **117** (1995), no. 2, 221–239.

[10] Yoram Hirshfeld, Mark Jerrum, and Faron Moller, *A polynomial algorithm for deciding bisimilarity of normed context-free processes*, Theoretical Computer Science **158** (1996), no. 1, 143 –159.

[11] Paris C. Kanellakis and Scott A. Smolka, *Ccs expressions, finite state processes, and three problems of equivalence*, Information and Computation **86** (1990), no. 1, 43 –68.

[12] Jeroen JA Keiren, *Modal μ-calculus (version 1.1)* (2013).

[13] C.J. Koomen, *Algebraic specification and verification of communication protocols*, Science of Computer Programming **5** (1985), 1 –36.

[14] Gavin Lowe, *An attack on the needham-schroeder public-key authentication protocol*, Inf. Process. Lett. **56** (November 1995), no. 3, 131–133.

[15] Roger M. Needham and Michael D. Schroeder, *Using encryption for authentication in large networks of computers*, Commun. ACM **21** (December 1978), no. 12, 993–999.

[16] G.L. Peterson, *Myths about the mutual exclusion problem*, Information Processing Letters **12** (1981), no. 3, 115 –116.

[17] Antti Valmari, *Failure-based equivalences are faster than many believe* (1995).

[18] Rob van Glabbeek, *The linear time-branching time spectrum*, CONCUR'90 Theories of Concurrency: Unification and Extension (1990), 278–297.

[19] H. E. Vaughan, *Review: William wernick, complete sets of logical functions*, Journal of Symbolic Logic **7** (1942), no. 2, 99–99.