Seminar Report

Process Calculi for Concurrency

Markus Reiter and Michael Kaltschmid

28 February 2018

Supervisor: Vincent van Oostrom

Abstract

This report provides a brief overview of process calculi and showcases the *mCRL2* toolset. In particular we will focus on *ACP* and *μ-Calculus*, since *mCRL2* is based on both of them. At the end we provide a short introduction on how to use some of the tools provided by *mCRL2*.

Contents

1 Introduction

This report is about process calculi and *mCRL2*, a modern toolset for process specification and model checking. We will discuss *mCRL2* itself and the concepts it is based upon. We will provide a short overview of the model checking logic used by *mCRL2*, *μ-Calculus*, for a more in-depth explanation on model checking please refer to the related report *Model Checking on Büchi Automata* by Deni Juric and Thomas Wohlfarter.

There are various process calculi like *ACP*, *CCS*, *CSP*, *Join-Calculus*, *PEPA* and *π-Calculus*. In short, a process calculus is an approach for formally modelling concurrent systems. It provides algebraic laws for analyzing and transforming process descriptions and permits formal reasoning about equivalences between processes (e.g. using bisimulation [1]). We will focus on the process calculus and process specification syntax *mCRL2* uses, which is based on the Algebra of Communicating Processes [2].

We will also take a look at Labelled Transition Systems, since they are the low level counterpart to ACP and process calculi in general. The report is largely based on Modelling and Analysis of Communicating Systems [3] by Jan Friso Groote and Michel Reniers.

2 Labelled Transition Systems

An LTS (Labelled Transition System) is a directed labelled graph and it consists of a set of states and a set of transitions labelled with actions that connect the states. Additionally it must have an initial state. It is also important to note that it will deadlock if a reachable state does not terminate and has no outgoing transitions. If a state has more than one outgoing transition with the same label to different states, then it is non-deterministic.

An LTS is a tuple $(S, A, \rightarrow, s_0 >)$ where:

- S is a set of states
- A is a set of actions
- $\rightarrow \subseteq S \times A \times S$ is a transition relation
- $s_0 \in S$ is the initial state

In Figure 1 and Figure 2 you can see examples of two LTSs for alarm clocks. The first alarm clock on the left allows for repeated alarms which can be seen by the loop on top of the alarm state. With the second alarm clock it is only possible to signal the alarm once after it is set.

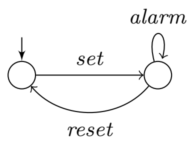


Figure 1

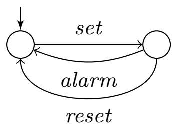


Figure 2

Both LTSs are deterministic since no state has more than one outgoing transition with the same label to a different state. However we can make the right LTS (Figure 2) non-deterministic by drawing a loop on top of the alarm state as you can see in Figure 3.

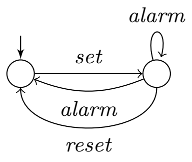


Figure 3

2.1 Bisimulation

Bisimulation is a binary relation between LTSs, where the related states of two LTSs behave the same way in the sense that each state has the same actions as its related state. Bisimulation [4] is not the only form of equivalence. We will show the difference between trace equivalence and strong bisimilarity as they are easy to show in an example.

- Trace equivalence:
  Two LTSs are equivalent iff they can perform the same sequences of actions, starting from their initial states

- Strong bisimilarity:
  If one LTS can perform an action a then the other LTS must also be able to perform an action a in a way that the resulting states are again related.

The two coffee machines in Figure 4 are trace equivalent but not strongly bisimilar because the resulting states are not related. As an example we can check what happens after inserting a coin. With the first coffee machine we reach state $P2$ and with the second coffee machine we reach either $Q2$ or $Q3$ depending on the choice of beverage. So the process of inserting a coin and then either getting a coffee or a tea is the same but the actions that are possible after inserting a coin differ.

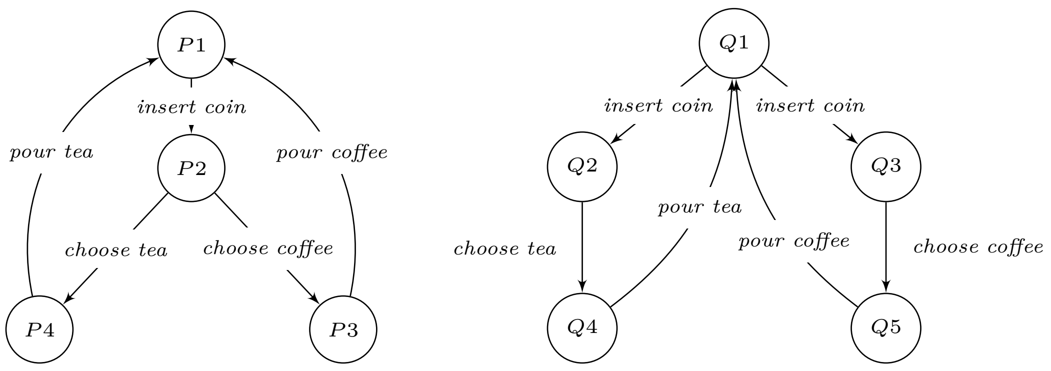


Figure 4: Bisimilarity of two coffee machines.

3 μ-Calculus

As already mentioned before μ-Calculus is an integral part of *mCRL2*. μ-Calculus shares some similarities with propositional logic with the extension that we are now able to speak about actions and traces in processes.
It is used to describe and verify properties of LTSs.

3.1 Hennessy-Milner Logic

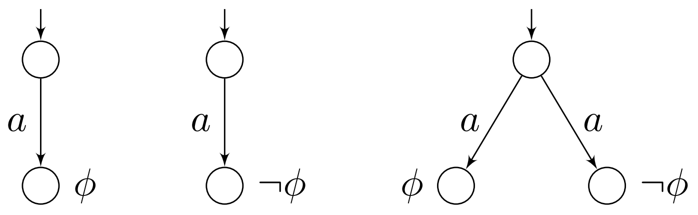
Hennessy-Milner Logic is a modal logic like LTL or CTL and for this reason we can see some similarities.

$$\phi ::= \mathit{true} \mid \mathit{false} \mid \neg\phi \mid \phi \wedge \psi \mid \phi \vee \psi \mid \phi \rightarrow \psi \mid \langle a \rangle \phi \mid [a]\phi$$

- *true* is true in each state of a process and *false* is never true
- $\neg$, $\wedge$, $\vee$ and $\rightarrow$ as in propositional logic
- $\langle a \rangle \phi$ is valid whenever an $a - action$ can be performed such that ϕ is valid after this a has been done
- $[a]\phi$ is valid when for every action a that can be done, ϕ holds after doing that a

3.2 Diamond and Box Modalities

Although $\langle a \rangle \phi$ - diamond modality and $[a]\phi$ - box modality look somewhat similar, yet they are very different. We can imagine the diamond modality like a F in LTL and the box modality like a G in LTL.
In order to show the difference between the two modalities we have four simple LTSs where we take a look whether the box modality or the diamond modality of a is valid.



	(a)	(b)	(c)	(d)
$\langle a\rangle\phi$	valid	invalid	valid	invalid
$[a]\phi$	valid	invalid	invalid	valid

Figure 5: Four LTSs.

In the first LTS (Figure 5a) a a action is possible to a state where ϕ holds and therefore both $\langle a\rangle\phi$ and $[a]\phi$ hold in the initial state.
In the second LTS (Figure 5b) there is no a action to a state where ϕ holds and for that reason neither $\langle a\rangle\phi$ nor $[a]\phi$ hold in the initial state.
In the third LTS (Figure 5c) there is an a action to a state where ϕ holds and an a action to a state where ϕ does not hold and therefore $\langle a\rangle\phi$ is valid since there is at least one a action that leads to a valid ϕ. $[a]\phi$ is invalid because not every a action leads to ϕ.
In the fourth LTS (Figure 5d) there is no a action at all and for that reason $\langle a\rangle\phi$ is not valid, because there needs to be an a action that can be performed. However $[a]\phi$ is valid, since there is no action that can be done.

3.3 Regular Formulae

In many cases we'd like to allow for more than one single action in a modality and in this case Regular formulas are useful.
An example would be that after two or more arbitrary actions, a specific action must happen or after two receive actions, a send action must follow.
Regular formulas are based on Action formulas. An Action formula is a set of multi-actions, which is as follows:

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

- α represents the set with exactly the multi-action α.
- *true* is the set of all multi-actions.
- *false* is the empty set.
- $\overline{af}$ denotes the complement of the set af.
- $\cup$ and $\cap$ denote union and intersection, respectively.

The modal formula $\langle true\rangle\langle a\rangle true$ expresses that an arbitrary action followed by an action a can be performed.
The formula $[true]false$ expresses that no action can be done.

Regular formulae extend action formulae by allowing actions in modalities.

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

- ϵ is the empty sequence of actions.
- $R_1 \cdot R_2$ represents the concatenation of R_1 and R_2.
- $R_1 + R_2$ denotes the union of R_1 and R_2.
- R^* denotes zero ore more repetitions.
- R^+ denotes one ore more repetitions.

Since $R_1 \cdot R_2$ denotes the concatenation of R_1 and R_2, we can write $\langle a \cdot b \cdot c\rangle true$ instead of $\langle a\rangle\langle b\rangle\langle c\rangle true$ true. Both express that the sequence of actions a, b and c can be performed.
In contrast to that $[a \cdot b + c \cdot d]false$ means that neither the sequence $a \cdot b$ nor $c \cdot d$ is possible.
Also $[a^+]\phi$ says that ϕ must hold in any state reachable by one or more a actions.
Furthermore $[\epsilon]\phi = \langle\epsilon\rangle\phi = \phi$ therefore we can always perform no action to stay in the same state.

Two other commonly used formulae are the *always* and *eventually* modalities.

$$\Box\phi = [true^*]\phi \qquad\qquad \diamond\,\phi = \langle true^*\rangle\phi$$

where $\Box\phi$ means that ϕ holds in all reachable states and $\diamond\phi$ says that there is a sequence of actions after which ϕ holds.

3.4 Safety and Liveness Properties

The *always* and *eventually* modalities can be further referred to as Safety and Liveness properties. These find their use case also in practice as they are the typical properties one would want to check in environments like distributed systems. For example to determine the responsiveness of a server or if a program terminates.

3.4.1 Safety Property

$\Box\phi$ says that something bad will never happen.

It is best to show this with an example. The following is a program with a critical region and two actions, enter and leave.

$$[\mathit{true}^* \cdot \mathit{enter} \cdot \overline{\mathit{leave}}^* \cdot \mathit{enter}]\mathit{false}$$

In this example it is impossible to enter twice in a row. There has to be a leave action in between two enter actions.

3.4.2 Liveness Property

$$\Diamond\phi \text{ means that Something good will happen.}$$

In order to demonstrate the liveness property we have a program that sends and receives messages.

$$[\mathit{true}^* \cdot \mathit{send}]\langle \mathit{true}^* \cdot \mathit{receive}\rangle \mathit{true}$$

Every time a message is sent, it can eventually be received.

3.5 Fixed Point Modalities

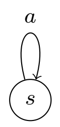
By extending the Hennessy-Milner logic with fixed points we have:

$$\mathit{af} ::= \alpha \mid \mathit{true} \mid \mathit{false} \mid \neg\phi \mid \phi \wedge \psi \mid \phi \vee \psi \mid \phi \rightarrow \psi \mid \langle a\rangle\phi \mid [a]\phi \mid \mu X.\phi \mid \nu X.\phi \mid X.$$

- X is a set of states
- $\mu X.\phi$ is the minimal fixed point
- $\nu X.\phi$ is the maximal fixed point
- minimal and maximal fixed point operators are each other's duals

$$\neg\nu X.\phi = \mu X.\neg\phi \qquad\qquad \neg\mu X.\phi = \nu X.\neg\phi$$

To make us more familiar with the concept of fixed points and especially with the difference between the minimal fixed point and the maximal fixed point, we have an example with a simple LTS with exactly one state and a loop with an action a.



Example 3.1. The first question is if $\mu X.\langle a\rangle X$ and $\nu X.\langle a\rangle X$ hold for the state s.
The states in question are the empty set $X = \emptyset$ and the set of all states $X = \{s\}$.
$X = \emptyset$ is the same as $X = \langle a\rangle X$, which can be further reduced to $\mathit{false} = \langle a\rangle \mathit{false}$, which we can see is valid.
Therefore $\mu X.\langle a\rangle X$ - minimal fixed point is valid for all states in the empty set but not in s and $\nu X.\langle a\rangle X$ - maximal fixed point is valid for all states in the largest set $\{s\}$.

4 ACP

ACP is the Algebra of Communicating Processes and is used to describe parallel and concurrent processes. We will see later on that *mCRL2* uses a syntax very close to the mathematical notation of ACP.

In ACP, a process is defined [2] as follows:

$$P ::= a \mid \delta \mid \tau \mid P + P \mid P \cdot P \mid (P || P) \mid (P |\lfloor P) \mid (P | P) \mid \tau_I(P)$$

Here, a represents an atomic action, δ is the deadlock action
and τ is the silent action.

$a + b$ means either a or b (*alternative operator*), $a \cdot b$ means a followed by b (*sequencing operator*).

$|\lfloor$ is the left-merge operator, it is the generalized version of $||$ (*merge operator*) and denotes concurrency.

Example 4.1. $(a \cdot b)|\lfloor(c \cdot d)$ ensures that the left branch is started first, in this case that a occurs first, so only the sequences *abcd*, *acdb* and *acbd* are possible.

$(a \cdot b)||(c \cdot d)$ is therefore actually an alternative between $(a \cdot b)|\lfloor(c \cdot d)$ and $(c \cdot d)|\lfloor(a \cdot b)$.

$|$ is the called the communications operator and is used do pass data between actions.

Example 4.2. $r(d)|w(d)$ means that the value d is communicated from action w from the right side to r on the left side.

τ_I is the abstraction operator. It is used to hide certain actions.

Example 4.3. $\tau_{\{c\}}((a + b) \cdot c)$ is equal to $(a + b) \cdot \tau$, which can be reduced to $a + b$.

5 mCRL2

mCRL2 is a toolset which is used for specifying and analysing behaviour of distributed systems and protocols. We chose this tool because it uses an ACP-like syntax for its formal specification language for processes as well as μ-Calculus for model checking.

5.1 Process Specification

Let's first take a look at the *mCRL2* syntax for specifying processes. Usually, a process specification file will have a file name of the form `*.mcrl2`.

```
act a, b, c, d, e;

proc P = a . b . c;
     Q = d + e;
     R = (a + b) . c . d . e;

init P;
```

Figure 6: The *mCRL2* process specification syntax.

As you can see in the first line in Figure 6, we specify all available actions using the `act` keyword. This relates to atomic actions in the ACP syntax. Now that we have defined the actions, we can specify the process itself, or multiple processes, if there are more than one. We do this using the `proc` keyword, followed by the name of the process. In Figure 6 we have three processes, `P`, `Q` and `R`. We then equate every process with a combination of actions, action sequences, action alternatives. Processes themselves can also be used on the right-hand side, which is essential for recursion. In *mCRL2*, we use `.` to denote a sequence and `+` to denote an alternative. This means that process `P` is a sequence of actions `a`, `b` and `c`. Process `Q` is a choice between either action `d` or action `e`. And finally, process `R` is a choice between action `a` and `b`, followed by `c`, `d` and `e`. Note the parentheses around `a + b`, since `.` binds stronger than `+`. Finally, the `init` keyword specifies the starting process, in this case `P`.

To demonstrate the power of ACP and μ-Calculus in combination with the *mCRL2* toolset, we will now look at an implementation of the two coffee machines (Figure 4) in *mCRL2*.

First, we have to define the actions for coffee machine 1.

```
act insert_coin, choose_coffee, choose_tea,
                   pour_coffee,   pour_tea;
```

The `insert_coin` action is self-explanatory. `choose_coffee` and `choose_tea` denote the actions of pressing either the button for coffee, or the button for tea. And finally, `pour_coffee` and `pour_tea` denote the action of dispensing either coffee or tea.

Since coffee machine 2 is trace equivalent to coffee machine 1, it has the same actions.

Next, we define the process for coffee machine 1.

```
proc P = insert_coin . (
           choose_coffee . pour_coffee +
           choose_tea . pour_tea
         ) . P;
```

We start by inserting a coin (`insert_coin`). Then, we are left with a choice between choosing coffee and pouring coffee or choosing tea and pouring tea. We see that choosing coffee or tea is tightly coupled to pouring it, since this is a choice between two sequences `(choose_coffee . pour_coffee)` and `(choose_tea . pour_tea)`. This is of course because we don't want the machine to dispense tea when in fact we chose coffee.

The last action is `P` itself, which results in a recursion. This is because we want the machine to be reusable.

We now define the process for coffee machine 2.

```
proc Q = (
           (insert_coin . choose_coffee . pour_coffee) +
           (insert_coin . choose_tea . pour_tea)
         ) . Q;
```

For this machine, we specify a process named `Q`. This process starts with a choice, which is either a sequence of inserting a coin, choosing coffee and pouring coffee or inserting a coin, choosing tea and pouring tea. In comparison to the first machine, the `insert_coin` action is now part of the choice between coffee or tea.

Lastly, we need to specify the starting process for both machines,

```
init P;
```

and

```
init Q;
```

respectively.

5.2 Process Properties

Now that we have specified the processes for two coffee machines, we want to define some properties they adhere to.

In *mCRL2*, these properties are defined in `*.mcf` (**m**odel **c**hecking **f**ormula) files and expressed by regular formulae.

We start with a simple property: The machine does only pour coffee after coffee is chosen. For this, we use a box modality which is false if any sequence of actions that are not `choose_coffee` is followed by `pour_coffee`.

```
[!choose_coffee* . pour_coffee]false
```

The *mCRL2* syntax for formulae uses `[` and `]` for box modalities.

The second property is the counterpart to the first. The machine should only pour tea if in fact we chose tea.

```
[!choose_tea* . pour_tea]false
```

With our third property, we want to ensure that the machine is reusable.

```
[true* . (pour_tea + pour_coffee) . insert_coin]true
```

After any valid sequence of actions (`true*`) which ends with pouring either coffee or tea (`pour_tea + pour_coffee`), inserting a coin is possible.

Lastly, we define a property which guarantees that the machine does not allow choosing a beverage without payment.

```
[!insert_coin* . (choose_tea + choose_coffee)]false
```

For that, we define that any sequence of actions which are *not* `insert_coin`, followed by choosing coffee or tea, is invalid.

5.3 Model Checking

We have now seen how to specify processes and process properties using *mCRL2*, but we have not yet used any tool provided by it.

For model checking we need three tools, which are provided by *mCRL2* as command line utilities. First, we need `mcrl22lps` to convert our `*.mcrl2` process specification into an LPS [5, Linear Process Specifications]. An LPS is a process specification with a very simple structure, which we can then convert into a so-called PBES [5, Parametrized Boolean Equation Systems] using `lps2pbes`. *mCRL2* can then check wether a given formula holds by solving this set of equations. This is done with the `pbes2bool` tool.

We will now look at a specific example using our coffee machines.

We assume the following directory structure[1]:

```
./
├─ coffee_machine_1.mcrl2
├─ coffee_machine_2.mcrl2
└─ properties
   ├─ choose_coffee_to_pour_coffee.mcf
   ├─ choose_tea_to_pour_tea.mcf
   ├─ insert_coin_after_pour.mcf
   └─ no_free_coffee.mcf
```

[1]You can find all files in the accompanying ZIP archive if you want to try these tools yourself.

We want to check that coffee machine 1 never gives out free coffee.

First, we convert our process specification into an LPS:

```
mcrl22lps coffee_machine_1.mcrl2 coffee_machine_1.lps
```

Next, we convert the LPS we just created into a PBES:

```
lps2pbes -f properties/no_free_coffee.mcf coffee_machine_1.
   lps properties/no_free_coffee.coffee_machine_1.pbes
```

Here, with the `-f` flag, we specify the formula which should be checked.

Finally, we use `pbes2bool` to solve the PBES.

```
pbes2bool properties/no_free_coffee.coffee_machine_1.pbes
```

If we additionally pass the `-v` flag, we can see the strategies used to solve the equation. Here is the output for our property:

```
Guessing input format: PBES in internal format
pbes2bool parameters:
  input file:            properties/no_free_coffee.
     coffee_machine_1.pbes
  data rewriter:         jitty
  substitution strategy: 0
  search strategy:       breadth-first
  solution strategy      lf
  erase level:           none
Loading PBES in pbes format...
Processed 1 and generated 1 boolean variables.
Generated 1 BES equations in total, generating BES
Solving a BES with 1 equations using the local fixed point
   algorithm.
Solving equations of rank 0.
The solution for the initial variable of the pbes is true
true
```

With the appropriate flags, you can manually specify which strategies should be used. These can be found in the documentation [5, pbes2bool] or by using the tool's help flag (`-h`).

In the last line of the output, we can see `true`. This boolean value represents wether the formula holds. This means that our property `no_free_coffee.mcf` holds for our first coffee machine `coffee_machine_1.mcrl2`

5.4 LTS

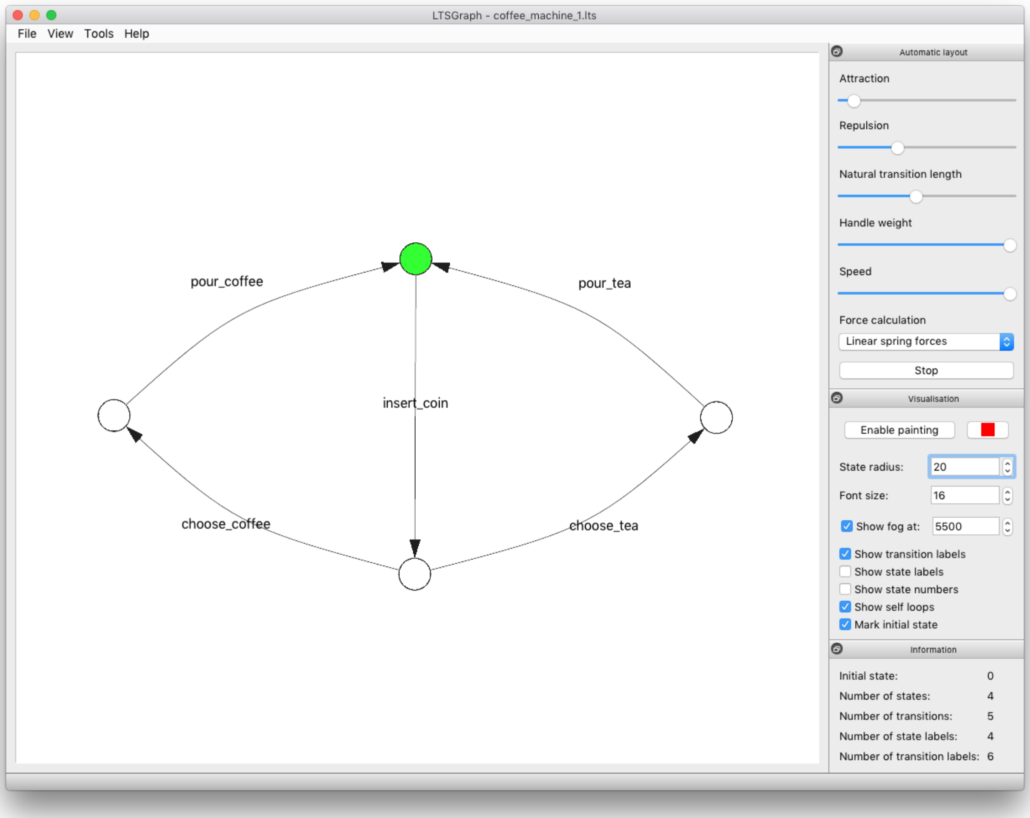
An LTS is a useful way of getting a visual overview of a process. The `lps2lts` command can be used to convert an LPS into an LTS.

```
lps2lts coffee_machine_1.lps coffee_machine_1.lts
```

We can now view the LTS with the `ltsgraph` command:

```
ltsgraph coffee_machine_1.lts
```

This opens a GUI (Figure 7) which allows you to interactively traverse the graph and view it in full.



Force calculation: Linear spring forces
State radius: 20
Font size: 16
☑ Show fog at: 5500
☑ Show transition labels
☐ Show state labels
☐ Show state numbers
☑ Show self loops
☑ Mark initial state

Initial state: 0
Number of states: 4
Number of transitions: 5
Number of state labels: 4
Number of transition labels: 6

Figure 7: `ltsgraph` window showing `coffee_machine_1.lts`.

As you can see in Figure 7, the green circle represents the starting state. In the *Automatic Layout* section, you can click *start* to automatically lay out the graph, which usually makes it much more readable. You can then drag the state circles around and by right-clicking them, you can fix their positions, so that they are not affected by the automatic layout anymore. One additional feature is *Exploration Mode*. You can access it from the *Tools* menu. In this mode, you can step from one state, beginning in the starting state, to all states that are reachable from it.

5.5 Bisimilarity

The last tool we want to demonstrate is `ltscompare`, which as the name implies is used to compare two LTSs.

We already know that our two coffee machines are trace equivalent, but not bisimilar. We now want to check this with `ltscompare`.

First, we need to generate the LTSs for our two coffee machines.

```
mcrl22lps coffee_machine_1.mcrl2 coffee_machine_1.lps
mcrl22lps coffee_machine_2.mcrl2 coffee_machine_2.lps

lps2lts coffee_machine_1.lps coffee_machine_1.lts
lps2lts coffee_machine_2.lps coffee_machine_2.lts
```

Next, we run `ltscompare` and pass `--equivalence` to specify which equivalence should be used, in our case `bisim`.

```
ltscompare --equivalence=bisim coffee_machine_1.lts
   coffee_machine_2.lts
```

Again, all available equivalences can be seen in the tool's help page (`-h` flag) or in the *mCRL2* documentation [5, ltscompare].

From the command, we get the following output:

```
LTSs are not equal (strong bisimilarity using the O(m log n)
    algorithm [Groote/Jansen/Keiren/Wijs 2017])
```

So in fact, the two LTSs are not bisimilar. We know, however, that they are trace equivalent. We can verify this using the `trace` option for `--equivalence`.

```
ltscompare --equivalence=trace coffee_machine_1.lts
   coffee_machine_2.lts
```

And indeed, we get the following output:

```
LTSs are equal (strong trace equivalence)
```

6 Conclusion

In this report we heard about *mCRL2* and integral concepts which it is based on. Specifically we went over Labelled Transition Systems, *ACP*, *μ-Calculus* in greater detail.

In order to be able to talk about *mCRL2* we tried to show how the different concepts and logics depend on each other and how they relate to *mCRL2*.

In the final chapter we then proceeded to showcase how we can test Bisimilarity and Trace Equivalence with *mCRL2*.

References

[1] *Process calculus — Wikipedia.* URL: `https://en.wikipedia.org/wiki/Process_calculus` (visited on 02/22/2017).

[2] *Algebra of Communicating Processes — Wikipedia.* URL: `https://en.wikipedia.org/wiki/Algebra_of_Communicating_Processes` (visited on 02/27/2017).

[3] *Modelling and Analysis of Communicating Systems.* URL: `http://www.win.tue.nl/~jfg/educ/2IW25/herfst2009/mcrl2-book.pdf` (visited on 02/22/2017).

[4] *Bisimulation — Wikipedia.* URL: `https://en.wikipedia.org/wiki/Bisimulation` (visited on 02/22/2017).

[5] *mCRL2 User Documentation.* URL: `http://www.mcrl2.org/web/user_manual/user.html` (visited on 02/22/2017).