Formal Modelling of Communication Systems: A Practical Approach

Mauricio Alejandro Cano
(with modifications by Jorge A. Pérez)

March 27, 2017

1 Introduction

The Calculus of Communicating Systems (CCS) is a formalism for dealing with the interaction and communication of concurrent systems. However, modelling real-life complex systems in CCS can be convoluted. One main reason for this is that "pure" CCS alone does not provide native support for data. Data and data structures are at the heart of most real-life systems; from lists to graphs and trees. The previous situation has provided good ground for the development of alternative models that include built-in data types that can be easily used to model more realistic systems.

In this brief tutorial we will present mCRL2 as a formalism that allow the natural inclusion of data in our models. One of the most important aspects of mCRL2 is that it comes with its own framework of software tools that allow the complete verification of the behavior of the model. In the following sections we will provide a brief overview of some of the tools included in mCRL2 and how they can be used in the context of model analysis.

This tutorial is based on the examples provided online[1], as well as the developments in [?]. We focus on the GUI of the mCRL2 toolset; it is also possible to invoke the tools from terminal commands, but the easiest way is using the GUI and we will present this way only.

2 A brief mCRL2 tutorial

2.1 A guiding example

Let us consider a vending machine that only sells one single item. We also know that there is a client that wants to buy that chocolate. Using CCS, we know that one possible model for this (simple) system is as follows:

$$\begin{aligned} M &= coin.\overline{item}.M \\ U &= \overline{coin}.item \\ S &= M \mid U \end{aligned}$$

where both *item* and *coin* are co-actions that can interact with each other. When moving from one modeling formalism to another, we should to take into account what can be done in one that

[1]See `http://www.mcrl2.org/release/user_manual/index.html`

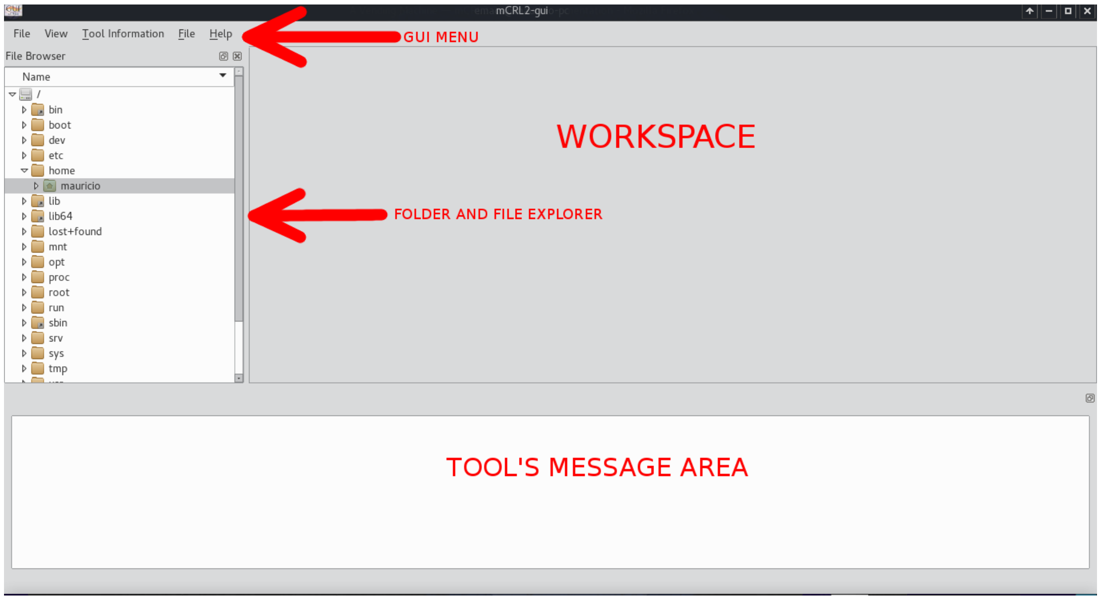


Figure 1: Main mCRL2 window

cannot be done in the other. In the following we will show, via this simple example that mCRL2 can model the same kind of systems as CCS, while extending it with data types and time.

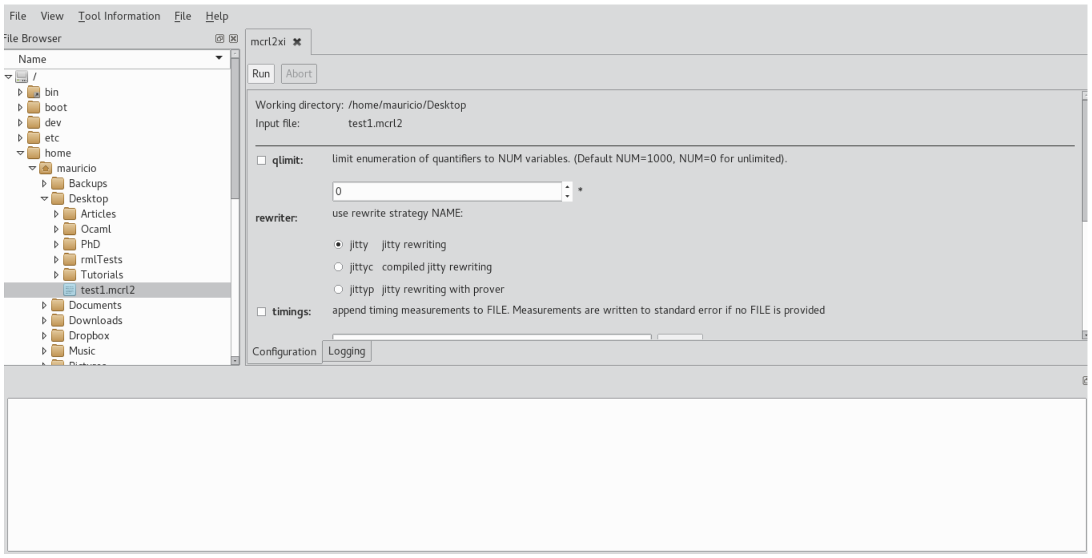
2.2 Starting the environment

Before doing any modelling, it is important to start our tool and to know how to set up the environment.

Action 1 *The first step is to start the tool by double clicking the mCRL2 icon of your local installation. After doing that, you will find yourself looking at Figure 1.*

From the view in Figure 1 we may invoke all the tools that will be used in the tutorial. This view is divided in three sections. The first contains five drop-menus (File, View, Tool Information, File, Help). The first two menus (File and View) provide management of the main GUI, such as resetting the view and closing all the executing processes. The menu Tool Information provides a brief description of all the tools included in mCRL2. The File menu is arguably the most important one, as it allow us to create new files. The second part of the main GUI is the folder and file explorer. It allow us to move through different folders and files, as well as to start the required tools. Lastly, we have a message area, where some general messages will appear when running different tools.

Action 2 *Find a folder where you can create your test files and while having that folder selected in the Folder and File Explorer, click in File and New File. The new file should appear inside the selected folder; name it* `test1.mcrl2`.



```
Working directory: /home/mauricio/Desktop
Input file:        test1.mcrl2

[ ] qlimit:   limit enumeration of quantifiers to NUM variables. (Default NUM=1000, NUM=0 for unlimited).
              0  *
    rewriter: use rewrite strategy NAME:
              (•) jitty   jitty rewriting
              ( ) jittyc  compiled jitty rewriting
              ( ) jittyp  jitty rewriting with prover
[ ] timings:  append timing measurements to FILE. Measurements are written to standard error if no FILE is provided
```

Figure 2: Starting the editor

We will now explain the different file extensions that we will use in mCRL2. In principle, there are three important extensions in this tutorial: `.mcrl2`,`.lps`,`.lts`. The `.mcrl2` is the most important one: it is the one that we will use to create our specifications; the other two extensions are *generated*. In particular, we need to generate an `.lps` file before having an `.lts` file. The `.lts` rule is the one that will allow us to visualize graphically the LTS of the specification we model in the `.mcrl2` file. Now we will create a simple specification of our vending machine example.

Action 3 *Right-click the file that you have just created. In the menu that appears select editing and click on* `mcrl2xi`*. The main GUI should now look like Figure 2. At this point we just click* `run`.

After clicking `run` we will obtain the window in Figure 3. This window has two parts: the first one is the common writing and editing part, while in the other we can check syntax and the correct typing of our program. In this last part it is also possible to test the functions and operations that we define over our data types and data structures. In the next section we will introduce the language and how the code blocks and structure should look.

2.3 Syntax and coding in mCRL2

We now introduce the syntax and code block structure of a *basic* mCRL2 specification. Throughout the tutorial we will only use built-in data types, so we will not go to create user defined data structures, as this goes beyond the scope of the course. A basic specification in mCRL2 looks as follows:

```
%This is a comment

%Action definitions
```

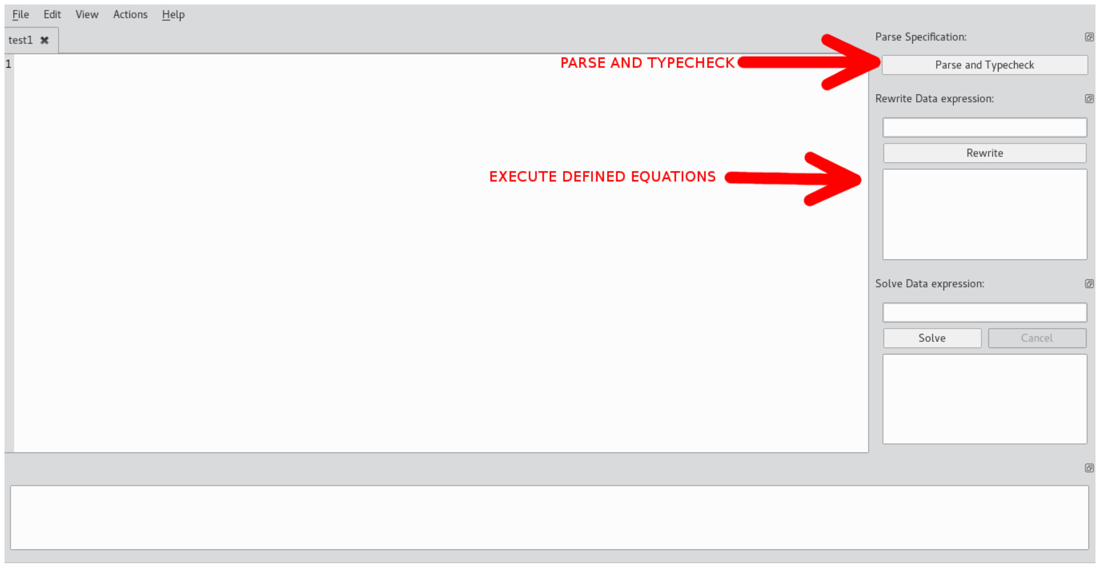


Figure 3: Editor window

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

%Base processes for our system
proc P = a.b.c.P + b.a.c.P;
     Q = d.e.Q + e.d.Q;

%Initialization of the system definition, communicating and parallel processes.
init
    hide({ad},
        allow({be, ad,c},
            comm({a|d -> ad, b|e->be},
                P||Q)
            )
        );
```

We can see in the example code that there are three main programming blocks:

1. **act:** In this block we define our actions and the types that they will have. Actions may not have a type; in this case we have only defined untyped actions, which means that they cannot take data values. In further examples we will show how to define valued actions.

2. **proc:** In this section we define the processes that are the basis of our system. That means that we define all the processes that will execute in parallel in the *complete system*. Here we introduce the complete syntax for the definition of processes without communication:

```
P,Q ::= |P + Q                    (Nondeterministic choice)
        |P || Q                   (Parallel composition)
```

```
|(condition) -> P <> Q      (If condition then P else Q)
|(Condition) -> P           (If condition then P)
| a.P                       (Prefixing operation)
```

3. **init:** Lastly we find the initialization block, where we define our complete system and set which actions are hidden, allowed (not restricted) and communicating. It is important to note that the order for these operations should be as mentioned here; otherwise, problems might occur during verification. The hiding operator `hide({act1,act2,act3}, P);` works exactly as the restriction operator in CCS: it does not allow the mentioned actions to execute. The allow operator `allow({act1,act2}, P);` *enforces the communication* to occur in `act1` and `act2`. Lastly the communication operator `comm({act1|act2 -> act3},P)` permits the communication between `act1` and `act2` to happen.

   To clarify this point better we have to consider that mCRL2 does not have co-actions as CCS. This fact, coupled with the ability that mCRL2 has of executing multiple actions at the same time (*multi-actions*), gives a different communication mechanism than that of CCS. Communication in mCRL2 is interpreted as the execution of two or more multi-actions renamed under a name that synchronizes them. That is why the instruction `comm({act1|act2 -> act3},P)` could be also read as "let `act1` and `act2` synchronize on `act3` in `P`". Even then, the communication operator is not enough to enforce the communication, it only makes it possible. The operator that actually enforces the communication is the allow operator. Which is why in our code example we are only allowing actions `ad, be, c`.

Action 4 *After having explained and defined the previous code blocks, we can now proceed to implement our vending machine example. The code will be the following:*

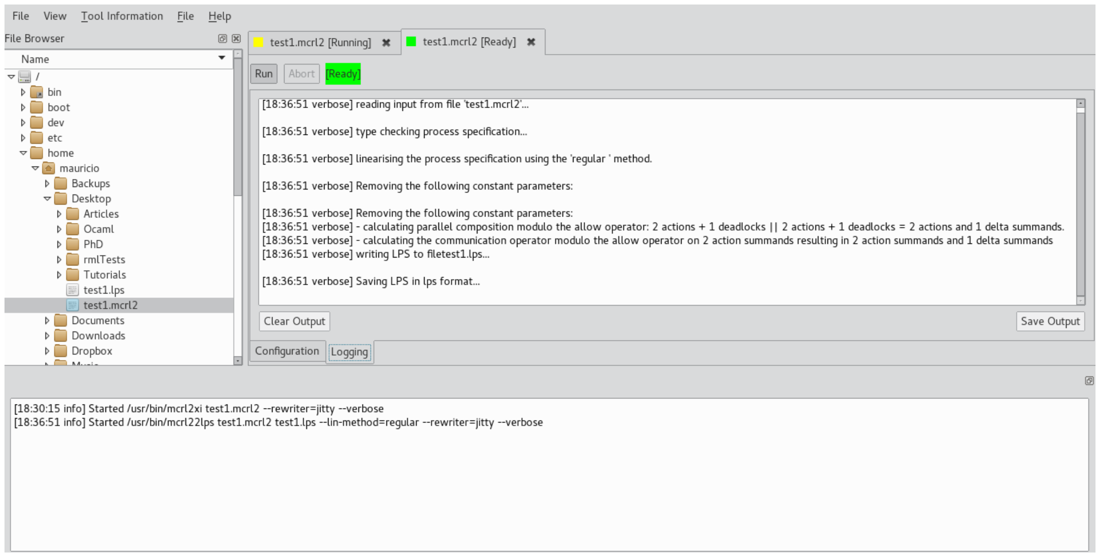
```
act
   ins, acc, giveItem, takeItem, coin, ready ;
proc
   U = ins.takeItem.U;
  M = acc.giveItem.M;
init
   allow({coin, ready},
          comm({ ins|acc -> coin, giveItem|takeItem -> ready },
                U || M
                )
          );
```

Notice that we are not using the hiding operator for this example as we do not want to hide the multi-actions so that they can be seen in the LTS we will generate.

2.4 Generating an LTS in mCRL2

It is very easy to generate an LTS in mCRL2.

Action 5 *Return to the main GUI, and right click on the file that contains our vending machine example. Once in this menu, select transformation and then click* `Run` *in the new tab that appears in the workspace area. This is shown in Figure 4.*



```
[18:36:51 verbose] reading input from file 'test1.mcrl2'...

[18:36:51 verbose] type checking process specification...

[18:36:51 verbose] linearising the process specification using the 'regular ' method.

[18:36:51 verbose] Removing the following constant parameters:

[18:36:51 verbose] Removing the following constant parameters:
[18:36:51 verbose] - calculating parallel composition modulo the allow operator: 2 actions + 1 deadlocks || 2 actions + 1 deadlocks = 2 actions and 1 delta summands.
[18:36:51 verbose] - calculating the communication operator modulo the allow operator on 2 action summands resulting in 2 action summands and 1 delta summands
[18:36:51 verbose] writing LPS to filetest1.lps...

[18:36:51 verbose] Saving LPS in lps format...
```

```
[18:30:15 info] Started /usr/bin/mcrl2xi test1.mcrl2 --rewriter=jitty --verbose
[18:36:51 info] Started /usr/bin/mcrl22lps test1.mcrl2 test1.lps --lin-method=regular --rewriter=jitty --verbose
```

Figure 4: LPS generation

We will see that a new file was created. We cannot directly see this file, as it is binary. However, it is possible to use the file to generate an LTS, as explained in the following.

Action 6 *Right-click on the new file, a new set of options, different from the ones that the* `.mcrl2` *file should appear. Select the option transformation and select* `lps2lts`*. At this point, a new tab should appear in the workspace. In this new tab, you can configure different options for your LTS. Once you are ready, click in run and a new file should appear in the explorer. This new file should have extension* `.lts`*.*

This `.lts` file enables two important operations that we will cover: *visualization* and *bisimulation verification.*

2.5 LTS Visualization

To visualize an LTS we just have to right-click on the file and select the Analysis menu.

Action 7 *After selecting the Analysis menu, click on the ltsgraph tool. Figure 5 shows the LTS.*

The produced LTS, however, is not really well organized. You may manually accommodate the states and labels by moving the states and labels with your mouse, or by clicking Start in the left panel and tweaking the different options you have in that panel. After tweaking the LTS, you should have something similar to what it is presented in Figure 6. Now we will show how to compare two LTS.

2.6 Comparing LTS

To compare LTS we need to use the tool `ltscompare`, which can be found in the menu Analysis when right-clicking an `.lts` file.

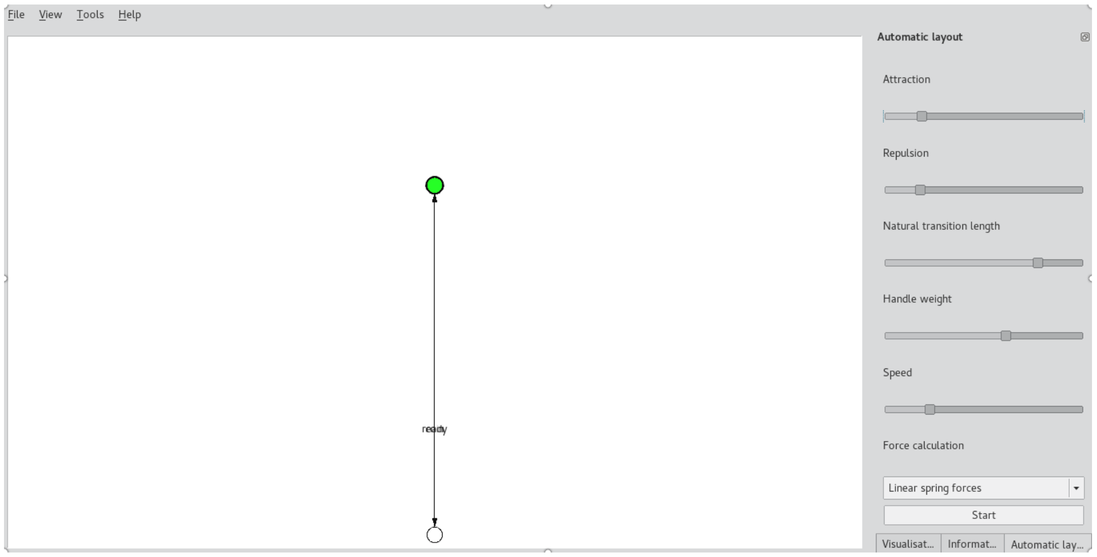


Figure 5: LTS Visualization tool

Action 8 *Select and click* `ltscompare`*: you should obtain in the workspace the tab that is shown in Figure 7.*

In this window you should be able to select another `.lts` file to compare the file that you first selected, as well as to choose the type of equivalence you want to use. You can also select an option to generate a counter-example in the case that you want to know where the equivalence comparison fails.

In the rest of this document we will briefly cover the basics of the built-in data types, and illustrate how to specify systems that use data.

2.7 Data types and mCRL2

As mentioned earlier, one of the motivations to use mCRL2 is to overcome the limitations of CCS for dealing with data. Note that mCRL2 is a typed specification language, which means that actions can carry different types of values, so it is necessary to define the type of data that the action will carry. For example, we could define the actions `a,b,c` that will carry natural numbers and the action `d` that will carry a list of natural numbers. The previous actions can be defined as follows:

```
act  a,b,c: Nat;
     d: List(Nat);
```

We can see that each set of actions of a certain type has to be defined each in one line. Note that both natural numbers and lists have some built-in operations which can be tested in the rewrite window shown in Figure 3. For example, you could write $3 + 1$ and click rewrite and the text box below the button would reply 4. For lists, there are the expected operations for concatenation, removal and getting the element in a certain position. For a complete account of all the operations

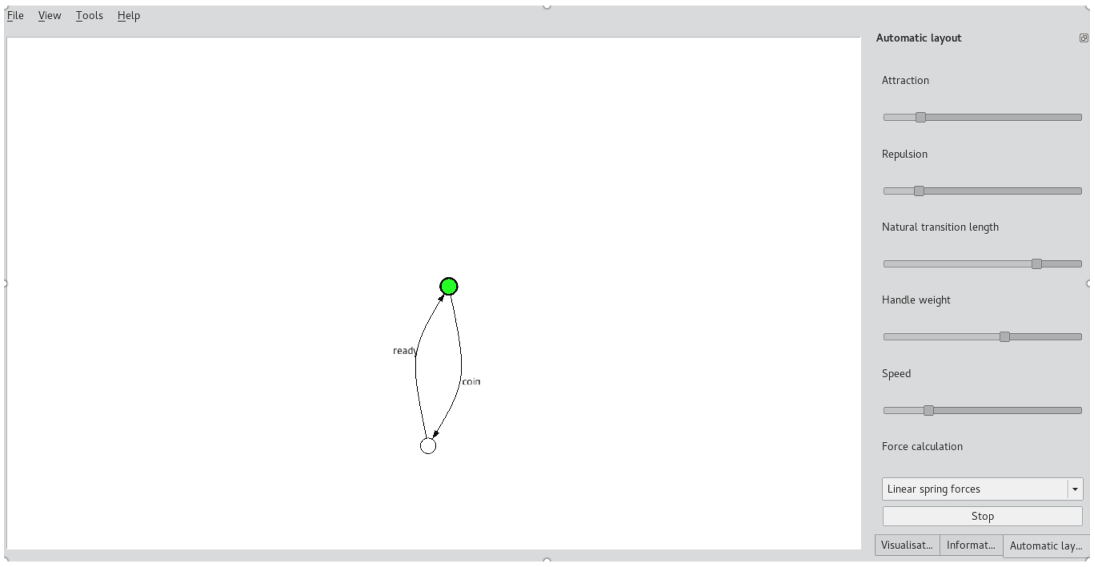


Figure 6: LTS organized

available over lists you can check the online mCRL2 description[2].

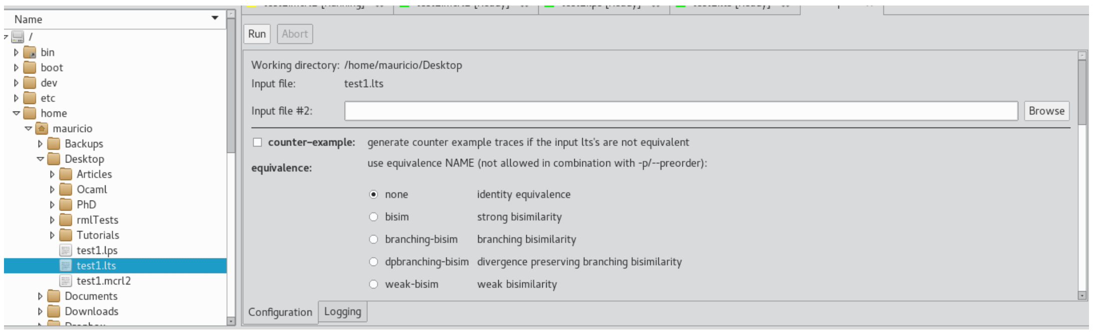
Now, consider the vending machine that has been our example. Now imagine that it is possible to put any any money value there. The code would be as follows:

```
act
  ins, acc, coin: Nat;
  giveItem, takeItem, ready ;
proc
  U(n:Nat) = ins(n).takeItem.U(n);
  M = sum n:Nat.(n<=10) -> acc(n).giveItem.M;
init
  allow({coin, ready},
       comm({ ins|acc -> coin, giveItem|takeItem -> ready },
            U(10) || M
            )
        );
```

Note that actions **ins, acc, coin** are now of type `Nat`. This means that all the actions that will carry natural numbers should be correctly assigned to natural number variables. Another interesting feature of adding data to our models is that we can now define *parametric processes*. The process `U(n:Nat)` will take a natural number `n` and use it inside its definition. One can imagine this as defining functions in your favorite programming language.

A new and important construct that we see is the sum process `sum n: Nat.P`, which intuitively specifies that we will consider the sum of all natural numbers and for each number `n`, we will use it in `P`. This is useful to simulate receiving a number, as in our example. However, it has its drawbacks.

[2]See `http://www.mcrl2.org/release/user_manual/language_reference/data.html`



```
Working directory: /home/mauricio/Desktop
Input file:        test1.lts
Input file #2:

[ ] counter-example:  generate counter example traces if the input lts's are not equivalent
equivalence:          use equivalence NAME (not allowed in combination with -p/--preorder):

                      (●) none               identity equivalence
                      ( ) bisim              strong bisimilarity
                      ( ) branching-bisim    branching bisimilarity
                      ( ) dpbranching-bisim  divergence preserving branching bisimilarity
                      ( ) weak-bisim         weak bisimilarity
```

Figure 7: Comparing LTS

For instance, if you do not limit the amount of numbers with a conditional as in the example, the generation of the LTS can take too much time, or even not finish.

In the following we will give some exercises to think about during the tutorial

3 Exercises (Homework No. 3)

1. Implement the two presented examples in mCRL2 and generate their respective LTS files. Compare the generated LTS using `ltscompare`. Are they behaviorally equivalent? Argue for your answer.

2. How would you extend the vending machine example with data so that according to the number of coins you receive a different item? Implement this extension in mCRL2 and generate its corresponding LTS.

3. Recall the "small university" process *SmUni*, given in Aceto et al.'s book (Equation 2.4, Section 2.1.). This process describes the interaction of a coffee machine (CM) and a computer scientist (CS) so as to produce infinitely many (single-authored) publications.

   a) How would you represent the *SmUni* process in mCRL2?

   b) Starting from your previous answer, propose an extended mCRL2 model that uses multi-actions to represent publications which have three or more computer scientists as authors. Discuss the design choices embedded in your solution.