Technische Universität München

Fakultät für Informatik

Bachelor's Thesis in Informatik

MoTraS : A Tool for Modal Transition Systems

Martin Stoll

Technische Universität München

Fakultät für Informatik

Bachelor's Thesis in Informatik

MoTraS : A Tool for Modal Transition Systems

MoTraS : Ein Werkzeug für Modale Transitionssysteme

Author: Martin Stoll
Supervisor: Univ.-Prof. Dr. Dr. h.c. Javier Esparza
Advisor: M. Sc. Jan Kretinsky
Submission Date: 15.08.2011

I assure the single handed composition of this bachelor's thesis only supported by declared resources.

Munich, August 10, 2011

Contents

Chapter 1

Introduction

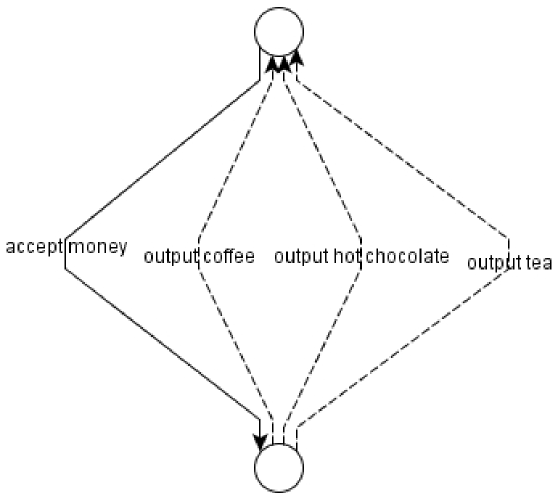
MoTraS - A tool for Modal Transition Systems. A tool that provides functionalities for displaying, drawing, importing and exporting Modal Transition Systems as well as executing the yet-to-be investigated algorithms on them. A tool with a convenient and intuitive graphical user interface which can be of great help for every scientist. And finally a tool that is ready for further extensions, as research never comes to an end.

The Modal Transition Systems concept is very “useful as a specification formalism of component-based systems as it supports compositional verification and stepwise refinement.” [BKLS09b]. The idea is to start from an initial specification and to make “a series of small and successive refinements until eventually a specification is reached from which an implementation can be extracted directly.” [BKLS09b]

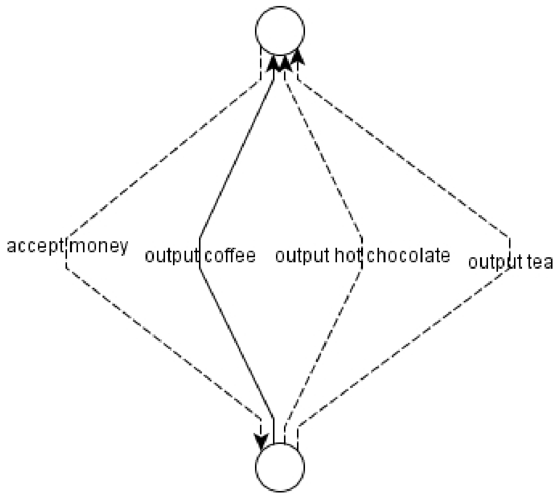
What basically are Modal Transition Systems (MTSs)? They are an extension of the concept of Labeled Transition Systems (LTS), as they allow two different types of transitions: may-transitions and must-transitions. May-transitions are allowed to be in each refining MTS, but do not need to be realized necessarily. Must-transitions are obliged to be in every refinement of the given MTS. It is intuitively clear that every must-transition has a corresponding may-transition (as everything that is required must be allowed), and so it is a common convention to draw only the must-transition for the graphical representation.

Graphically, an MTS is a directed graph with nodes and two different types of labeled edges. The may-transitions are drawn as dashed arrows, while the must-transitions are represented as solid arrows.

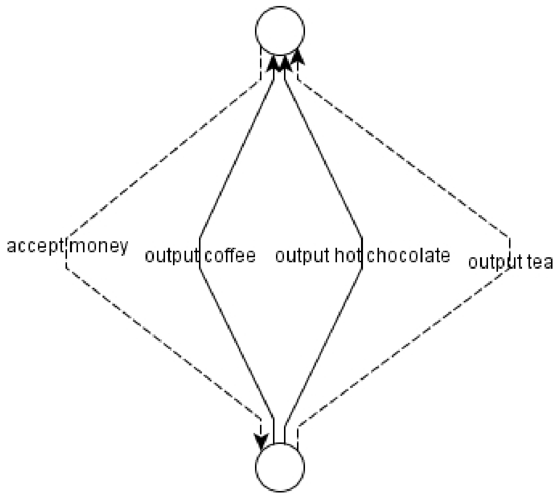
At this point, the following example illustrates a first application of this concept. Imagine a boss of a small company who wants to buy a coffee machine for his two employees.



(a) The requirements of the boss



(b) The requirements of the first employee



(c) The requirements of the second employee

Figure 1.1: The 3 different requirements on the coffee machine

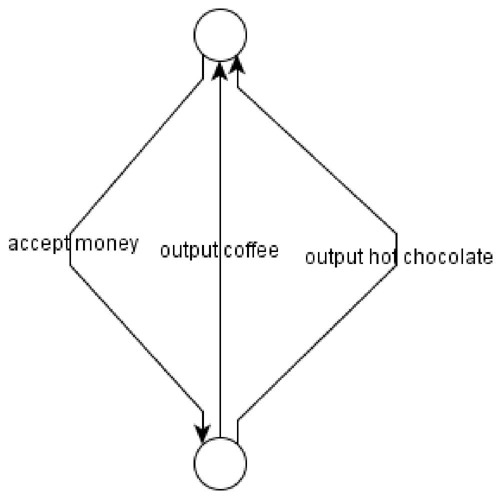


Figure 1.2: The common implementation - or the system everybody's happy with

The machine should produce coffee and other drinks after inserting some money. He himself and his employees have different requirements on the machine. The boss wants the machine to accept money. He doesn't really care about the output of the machine, but he determines for financial reasons the machine producing only the following three types of drinks: coffee, hot chocolate and tea. The employees are not really happy that they have to pay for the drinks, but they accept it. The first employee is a coffee drinker, who would be satisfied if the machine could produce coffee and has no interest in other drinks. The second employee wants it to produce coffee and hot chocolate and doesn't care for the tea. Figure 1.1 shows the three different requirement specifications as MTSs.

Now the boss tries to construct a system fulfilling all requirements together. The system should provide as little options as possible, as the boss doesn't want to spend more money than necessary. The system shown in Figure 1.2 is the best solution for this problem, the so called "smallest common implementation".

This concept - and more computational problems are introduced in the next chapter after some basic definitions on MTSs.

Chapter 2

Theory

2.1 Modal Transition Systems

A *Modal Transition System* (MTS) is a triple $(P, \dashrightarrow, \longrightarrow)$, defined as follows:

- P is a set of processes
- $\dashrightarrow$ is a may-transition relation and $\longrightarrow$ is a must-transition relation with $\longrightarrow \subseteq \dashrightarrow \subseteq P \times \Sigma \times P$, whereby Σ is a set of actions, the action alphabet

Furthermore, we want to define a special case of an MTS:
An implementation is an MTS, where every may-transition is either realized (there is a corresponding must-transition for it), or omitted. Formally:

An *Implementation* is an MTS with $\longrightarrow = \dashrightarrow$.

Remark: When speaking of processes in the context of algorithms, we semantically do not only mean the single state, but the whole reachable part of the underlying MTS of the process. That's the reason why e.g. the notion "implementation" above can also be defined for processes.

In most of the computational problems that are described later, the runtime of the algorithms is heavily affected not only by the size of the input processes, but also by an important property of processes - determinism:

A process D is *deterministic*, if the actions of every outgoing may-transition of every reachable process of D are pairwise distinct.

From now on, S and T always denote processes, I and J always denote implementations and D is always a deterministic process.

As MTS are an extension of Labeled Transition Systems (LTS), there is also an extension for MTSs, the "Disjunctive Modal Transition Systems". It allows to define sets of outgoing must-transition of a process, from which only one has to be realized. The former must-transitions are now the sets containing only one transition. The formal definition is not so different from the definition of MTSs, just the must-transition relation is defined in a more complex way:

A *Disjunctive Modal Transition System* (DMTS) is a triple $(P, \dashrightarrow, \longrightarrow)$, defined as follows:

- P is a set of processes
- $\dashrightarrow$ is a may-transition relation and $\longrightarrow$ is a must-transition relation with $\dashrightarrow \subseteq P \times \Sigma \times P$ and $\longrightarrow \subseteq P \times 2^{\Sigma \times P}$, whereby Σ is a set of actions, the action alphabet

A disjunctive transition is drawn as an arrow that splits up to two or more arrows in the middle. An example of a DMTS is shown in Figure 2.3 (c).

2.2 Computational problems

2.2.1 Modal and Thorough Refinement

In the introduction, the refinement problem has already been mentioned. The question is now the following: Given two MTS, one wants to check whether a certain process of the first MTS refines a specified process of the second MTS. There are actually two different refinement problems concerning MTS, the **modal refinement** (MR) and the **thorough refinement** (TR).

One can describe the **modal refinement problem** in the following way: If S is refining T (written $S \leq_m T$), there must be a corresponding must-transition of S for every must-transition of T as well as a corresponding may-transition of T for every may-transition of S. The idea behind that is very simple: Everything that is required in the abstract system has to be in the refinement system as well, and everything that is allowed in the refinement system must also be allowed in the abstract system. Formally:

Let $M_1 = (P_1, \longrightarrow_1, \dashrightarrow_1), M_2 = (P_2, \longrightarrow_2, \dashrightarrow_2)$ be two MTSs over the same action alphabet Σ and $S \in P_1, T \in P_2$.

$$\begin{array}{lll} S \textit{ modally refines } T & \Leftrightarrow & \text{There exists a relation } R \subseteq P_1 \times P_2 : (S,T) \in R \wedge \\ & & \forall\ (A,B) \in R, a \in \Sigma : \\ & & 1.\ A \overset{a}{\dashrightarrow}_1 A' \Rightarrow B \overset{a}{\dashrightarrow}_2 B' : (A',B') \in R \\ & & 2.\ B \xrightarrow{a}_2 B' \Rightarrow A \xrightarrow{a}_1 A' : (A',B') \in R \end{array}$$

There is also a somewhat better way to understand the modal refinement problem: the **modal refinement game** [BKLS09b]. A modal refinement game on a pair of processes (S,T) of two MTS (as defined in the modal refinement definition) is a two-player game between an **Attacker** and a **Defender**. It is played in rounds, by modifying the current pair of processes (A,B), initially $A := S, B := T$. Each round includes the following steps:

1. Attacker chooses an action a and performs one of the following moves:
   a) $A \overset{a}{\dashrightarrow}_1 A'$ for some A';
   $A := A'$;
   b) $B \xrightarrow{a}_2 B'$ for some B';
   $B := B'$;
2. Defender has to respond with the following moves: If the Attacker has chosen move a (respectively b), he must also choose move a (respectively b):
   a) $B \overset{a}{\dashrightarrow}_2 B'$ for some B';
   $B := B'$;
   b) $A \xrightarrow{a}_1 A'$ for some A';
   $A := A'$;

If a player cannot make a move, he loses the game and the other player wins.
If the game is infinite, then the Defender wins.

One can derive the following results now:
★ *The Defender has a winning strategy* $\Rightarrow S \leq_m T$
★ *The Attacker has a winning strategy* $\Rightarrow S \nleq_m T$

The reason for explaining this game here is that there is an algorithm "MRMartin" for checking modal refinement integrated in the source code of the tool, that implements this Attacker-Defender game. More to this algorithm, which unfortunately could not be integrated in the tool, is shown in chapter 6.1.

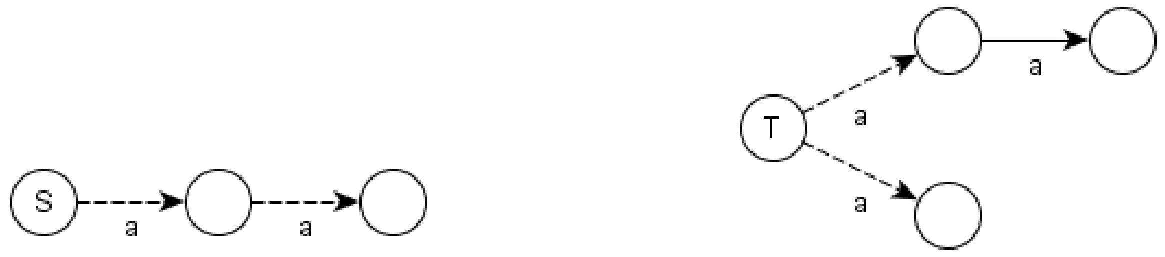


Figure 2.1: $S \leq_t T$, but $S \nleq_m T$

For the **thorough refinement problem**, another construct has to be defined first, the *set of all implementations of a process*:
$[\![S]\!] := \{I \mid I \leq_m S\}$

The condition for thorough refinement is now as follows:
S thoroughly refines T ($S \leq_t T$) $\Leftrightarrow [[S]] \subseteq [[T]]$.

Some resulting theorems are important to know (the proofs are shown in [BKLS09b]):

★ $\leq_t$ *and* $\leq_m$ *are transitive relations.*
★ $I \leq_m J \Leftrightarrow I \leq_t J$
★ $S \leq_m T \Rightarrow S \leq_t T$
★ $S \leq_t D \Rightarrow S \leq_m D$

Figure 2.1 shows an example of modal and thorough refinement.

2.2.2 The Deterministic Hull

Sometimes, one is interested in getting a deterministic system out of a non-deterministic MTS. In this situation, the concept of the **deterministic hull** is very helpful. The deterministic hull of a process is the "smallest (w.r.t. refinement) deterministic system refined by the original system." [BKLS09b]. It is computable for every process of an MTS.

Construction of the *deterministic hull* $\mathcal{D}(S)$ of a process S:

1. Let $(P, \longrightarrow, \dashrightarrow)$ be the underlying MTS of S.
2. Let $\mathcal{T}_a := \{T' \in P \mid \exists T \in \mathcal{T} : T \overset{a}{\dashrightarrow} T'\}$ for $\emptyset \neq \mathcal{T} \subseteq P$
   ($\mathcal{T}_a$ is the set of all may-successors of processes of $\mathcal{T}$ under the action a)

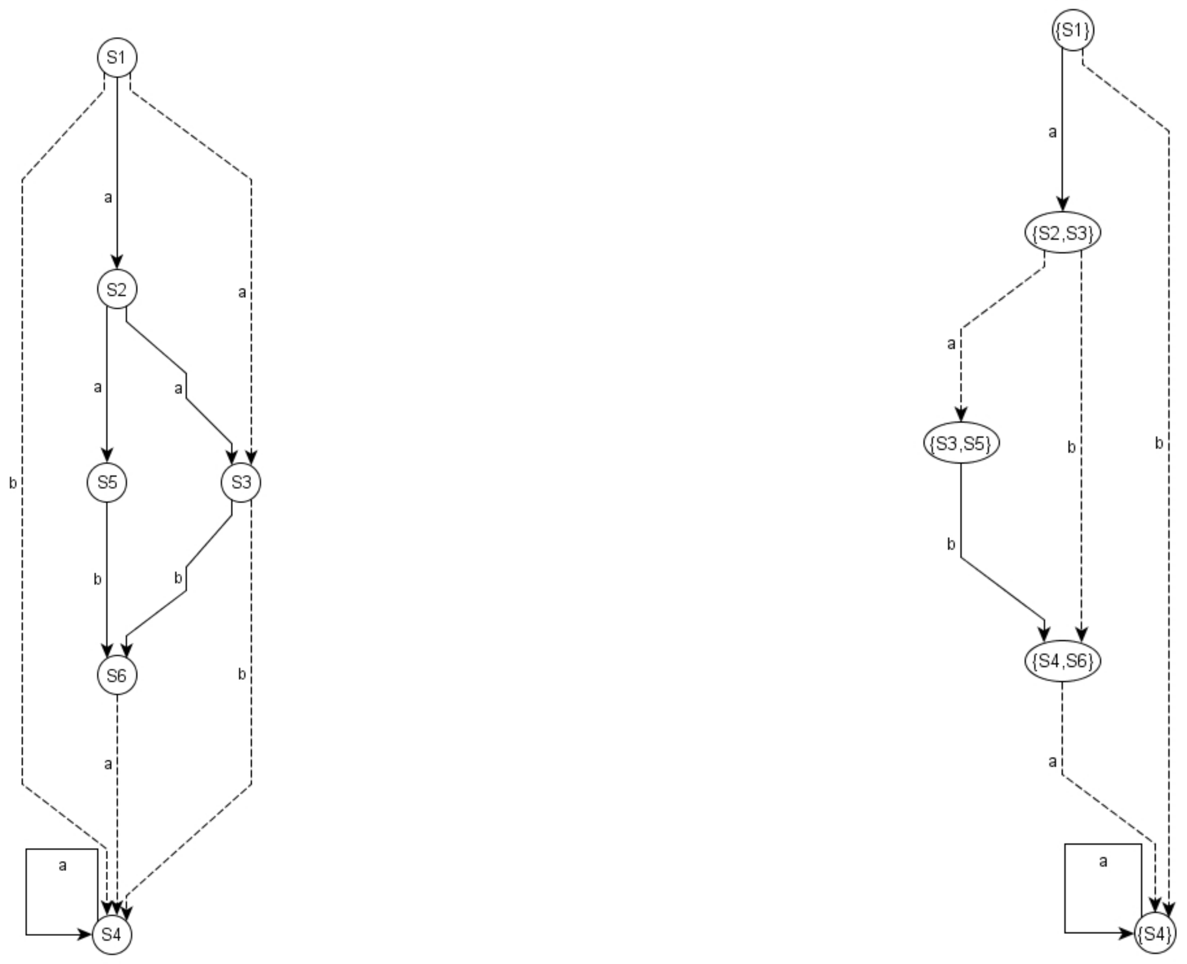


Figure 2.2: A process $S1$ and its deterministic hull $\mathcal{D}(S1) = \{S1\}$

3. The deterministic hull is defined as the MTS $M := (\mathcal{P}(P) \backslash \emptyset, \longrightarrow_D, \dashrightarrow_D)$, where the transitions are defined as follows:

   (i) $\mathcal{T} \overset{a}{\dashrightarrow}_D \mathcal{T}_a \Leftrightarrow \mathcal{T}_a \neq \emptyset$

   (ii) $\mathcal{T} \overset{a}{\longrightarrow}_D \mathcal{T}_a \Leftrightarrow \forall T \in \mathcal{T} : \exists T' \in \mathcal{T}_a : T \overset{a}{\longrightarrow} T'$

4. $\mathcal{D}(S) = \{S\}$

An example of the deterministic hull of a process is given in Figure 2.2.

2.2.3 The Smallest Common Implementation Problem

The smallest common implementation problem has already been introduced in the coffee machine example. The input for this problem is a set of processes, and the result (if there is one - as not every set of processes has a smallest common implementation) should be a process that implements all the processes of the set. The computation of the smallest common implementation for general MTSs is very complex. So we confine ourselves to show it for deterministic processes here, as MoTraS cannot compute a smallest common implementation of non-deterministic processes, either (but it can compute the greatest common implementation for DMTSs instead, which will be described later).

Smallest common implementation problem for k deterministic processes - denoted as CI_D^k:
$CI_D^k = \{\langle D_1, \ldots, D_k \rangle \mid \exists\, I : I \in [\![D_1]\!] \cap \cdots \cap [\![D_k]\!] \wedge D_1, \ldots, D_k \in d\mathcal{MTS}\}$
Remark: $d\mathcal{MTS}$ is the set of all deterministic MTSs

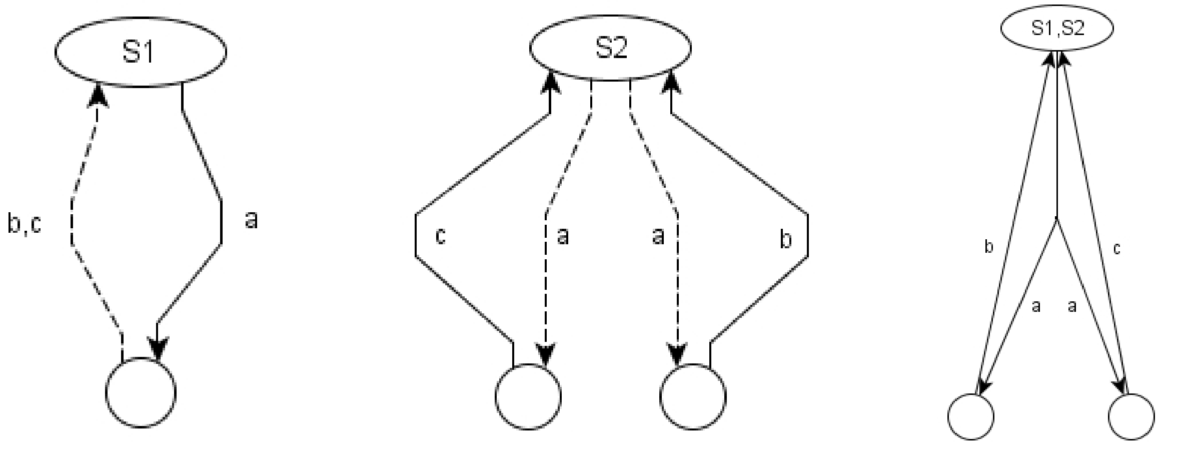
Computation of CI_D^k for $D_1, \ldots, D_k$:

1. Construct a graph, whereby every k-tuple $(E_1, \ldots, E_k)$ with E_i being a process of the underlying MTS of $D_i \,\forall\, i \in \{1 \ldots k\}$ is a node. $(D_1, \ldots, D_k)$ is the initial node.
2. Insert an edge after the following rule:
   $(E_1, \ldots, E_k) \xrightarrow{a} (F_1, \ldots, F_k) \Leftrightarrow \forall\, i : E_i \overset{a}{\dashrightarrow} \wedge \exists\, j : E_j \xrightarrow{a} F_j$
3. Mark every node $(E_1, \ldots E_k)$ that is reachable from the initial node and that satisfies the following condition:
   $\exists\, a, i : E_i \xrightarrow{a} E_i' \wedge \exists\, E_j : E_j \overset{a}{\not\dashrightarrow}$
4. Delete all nodes that are not reachable from the initial node.
5. The graph represents a smallest common implementation of $(D_1, \ldots, D_k) \Leftrightarrow$ the graph contains no marked nodes.

An example is already provided in the first chapter in Figure 1.1 and 1.2.

2.2.4 The Conjunction

A more general concept than the common implementation is the conjunction of a set of processes. The conjunction is a "process that is the greatest lower bound for a given set of processes w.r.t. the modal refinement" [BCK]. In other words, it is the most general process that refines all the processes of the given set. Although "MTSs may not have an MTS conjunction, there is always a conjunction expressible as a DMTS." [BCK] The formal definition of the conjunction is the following:



(a) A process S_1

(b) A process S_2

(c) The conjunction of S_1 and S_2

Figure 2.3: An example of the conjunction

The *conjunction* of a set of DMTSs $A_1, \ldots, A_k$ with $A_i = (P_{A_i}, \longrightarrow_{A_i}, \dashrightarrow_{A_i})\, \forall\, i \in \{1 \ldots k\}$ is a DMTS $C = (P_C, \longrightarrow_C, \dashrightarrow_C)$ with $P_C \subseteq P_{A_1} \times \cdots \times P_{A_k}$ and

- $(A_1, \ldots, A_k) \overset{a}{\dashrightarrow}_C (B_1, \ldots, B_k) \Leftrightarrow \forall\, i \in \{1 \ldots k\} : A_i \overset{a}{\dashrightarrow} B_i$
- $(A_1, \ldots, A_k) \overset{a}{\longrightarrow}_C \mathcal{B}$ with $\mathcal{B} = \{(a, (B_1, \ldots, B_n)) \mid (a, B_j) \in \mathcal{B}_j \wedge$ $(A_1, \ldots, A_k) \overset{a}{\dashrightarrow}_C (B_1, \ldots, B_k)\} \Leftrightarrow \exists\, j : A_j \longrightarrow \mathcal{B}_j$

Figure 2.3 shows an example of the conjunction of two processes.

2.2.5 The Greatest Common Implementation

This concept is a bit different from the common implementation problem described above. It is computable for every MTS / process by first computing the conjunction and then adding a must-transition for every may-transition that has no corresponding must-transition yet. As an abbreviation, we sometimes use the notion “maxCI” for this construct.

2.2.6 The Composition

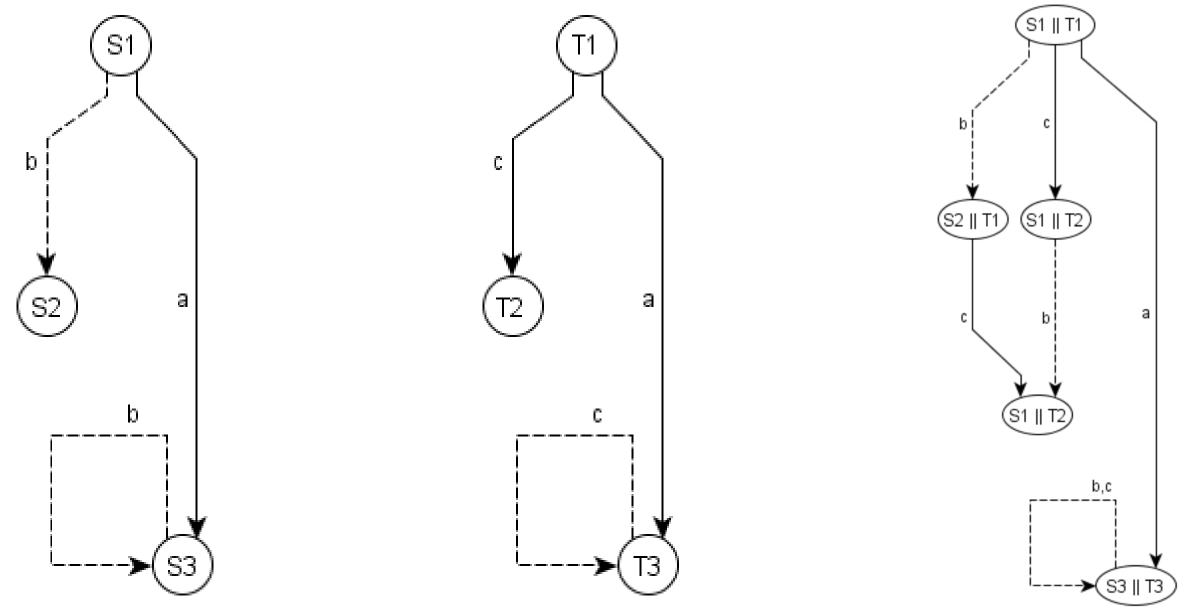
Another concept, quite different from the other problems that were shown above, is the composition. It enriches the MTS theory by providing the possibility of synchronization as well as subsequent executing of two processes. For the composition, one has to define a synchronizing alphabet $\Gamma \subseteq \Sigma$ first. The idea of this concept is the following:

- a transition with an action $a \in \Gamma$ will be added to the composition only in the case that there is a corresponding transition in M_1 AND M_2
- a transition with an action $a \notin \Gamma$ will be added to the composition if there is a corresponding transition either in M_1 OR in M_2

Formally:

- for $a \in \Gamma$:
  $S_1 \parallel S_2 \overset{a}{\dashrightarrow} S_1^{'} \parallel S_2^{'} \Leftrightarrow S_1 \overset{a}{\dashrightarrow} S_1^{'} \wedge S_2 \overset{a}{\dashrightarrow} S_2^{'}$
  (analogously for must-transitions)
- for $a \notin \Gamma$:
  $S_1 \parallel S_2 \overset{a}{\dashrightarrow} S_1^{'} \parallel S_2 \Leftrightarrow S_1 \overset{a}{\dashrightarrow} S_1^{'}$ as well as
  $S_1 \parallel S_2 \overset{a}{\dashrightarrow} S_1 \parallel S_2^{'} \Leftrightarrow S_2 \overset{a}{\dashrightarrow} S_2^{'}$
  (analogously for must-transitions)

Figure 2.4 illustrates an example of the composition.



(a) A process S_1 (b) A process T_1 (c) The composition $S1 \,||\, T1$

Figure 2.4: An example of the composition of two processes with the synchronized alphabet $\Gamma = \{a\}$

Chapter 3

MoTraS Documentation

This chapter is the user manual for MoTraS. Described here is the graphical user interface and the functionality of the tool.

As an outline, this is what the tool currently is able to do:

- displaying (disjunctive) MTSs
- importing and exporting (disjunctive) MTSs
- drawing (disjunctive) MTSs

Beneath that, the tool provides the following algorithms:

- checking modal refinement for two MTSs
- computing the deterministic hull for an MTS
- computing the composition for two MTSs
- checking / computing the smallest common implementation for two **deterministic** MTSs
- computing the conjunction for two (disjunctive) MTSs
- computing the greatest common implementation for two (disjunctive) MTSs
- performing LTL model checking
- creating random MTSs and random refinements of MTSs

3.1 Setup and System Requirements

The tool doesn't need to be installed - it works as a stand-alone. As the integrated tool for model checking is written in C, there are two MoTraS versions provided: One for Linux operation systems, and one for Windows.
Remark: The tool requires the Java Runtime Environment to be installed.

3.2 The Start GUI / Creating a new MTS

After starting the tool, one has several choices of creating / displaying an MTS after clicking on the button "New": "Display MTS", "Import MTS", "Draw MTS", "Generate Random MTS" and "Runtime Test for Algorithms". These five options are described in more detail here.

3.2.1 Display MTS

This option lets the user select one of the already created (imported or drawn) MTSs to display it on the GUI. Right after starting the tool, there are already a lot of sample MTSs selectable to display:

1. Figure VM1, VM2 and VMC (taken from [BKLS09b], p. 3)
2. Figure 3a and 3b (taken from [BKLS09b], p. 5)
3. Figure 5 (taken from [BKLS09b], p. 9)
4. Sample DMTS (taken from [BCK], Figure 7 in chapter 4)
5. Random x,y,z (4 different random created MTSs - x is the number of processes, y is the number of transitions and z is the maximum number of different actions)

Remark: The Figure 3a and 3b examples are equivalent to the MTSs of Figure 2.1, and the Figure 5 example is equivalent to the left MTS of Figure 2.2.

It is also possible to have more than one MTS opened, as the GUI is organized in tabs.

Remark: To avoid consistency problems, a copy will be created when selecting an MTS which is already opened.

3.2.2 Import MTS

Inputting an MTS in text format is quite easy, but there are a couple of restrictions concerning the input language:

1. Every line must contain exactly one process definition.
2. A process definition is of the following form:
   $\langle$ process name $\rangle$ = $\langle$ transition definition $\rangle$ + ... + $\langle$ transition definition $\rangle$
3. A transition definition can have three different forms:
   - may-transition: ? $\langle$ action $\rangle$. $\langle$ name of target process $\rangle$
   - must-transition: $\langle$ action $\rangle$. $\langle$ name of target process $\rangle$
   - disjunctive transition: {$\langle$ action $\rangle$. $\langle$ name of target process $\rangle$ | ...
     $\cdots$ | $\langle$ action $\rangle$. $\langle$ name of target process $\rangle$}
4. Process names and actions must not contain any of the following characters:
   '+', '|', '.' or '$'.

This is an example for a DMTS input:

```
S1 = {a.S2 | b.S3} + c.S4
S2 = ?b.S1
```

Importing MTS from file:
The file must be a ".dmts" file and be of the same format as described above.
Remark: When importing a file created by the tool via file export, the process names will contain '$' characters, which is allowed in this case.

3.2.3 Draw MTS

When clicking on "Draw MTS", a new empty panel is shown. Drawing MTSs will work like this:

- Creating a new process: Double-click on a free place on the GUI.
- Drawing a may-/must-transition: Clicking on the button "Type of Transition" lets the user change between may- / must- / and disjunctive transitions. Then click on the source process and drag the mouse to the target process. After that, the tool asks for the action name of this transition.

- Drawing a disjunctive transition: First, select "Disjunctive Transition" via the "Type of Transition"-button. Then double-click on a free place on the GUI, and a small node (fork) will be created. After that, drag an arrow from the source process to the fork. Finally, the user can add transitions starting from the fork to several target processes the same way as drawing may- and must-transitions.
- Adding self-transitions: Right-click on a process and select "Add Self-Transition".
- Removing processes: Right-click on the process and select "Remove Process". The process and all incoming and outgoing transitions will be removed.
- Removing transitions: Right-click on the transition and select "Remove Transition".
- Renaming processes: Right-click on the process and select "Rename Process". Then insert a new name for this process.
- Moving processes: Processes can be moved by selecting them and dragging them around.

3.2.4 Generate Random MTS

Selecting this option will generate a random MTS after inputting some properties. First, it will ask for the number of processes that the MTS should have. After that, input the number of transitions (this is the number of may-transitions the MTS will have - besides that, for every created may-transition there is a 30% chance that a corresonding must-transition will be created, too). Finally, type in the maximum number of different actions allowed. If the input is correct, a generated MTS with the specified properties will be displayed.

3.2.5 Runtime Test on Algorithms

This options runs a selected algorithm for random MTSs and measures the runtime. First, select an algorithm to be tested. Then choose whether MTSs of different sizes (incremental) should be created or whether they should have the same size. Then the tool asks for the lower and upper bound and an increment for the number of processes (or just for the number of processes in the case of a non-incremental test). After that, type in the maximum number of different actions. For the number of transitions, choose a rate first, which can be refined with a factor in the second step (e.g. for $4n$ transitions (with n being

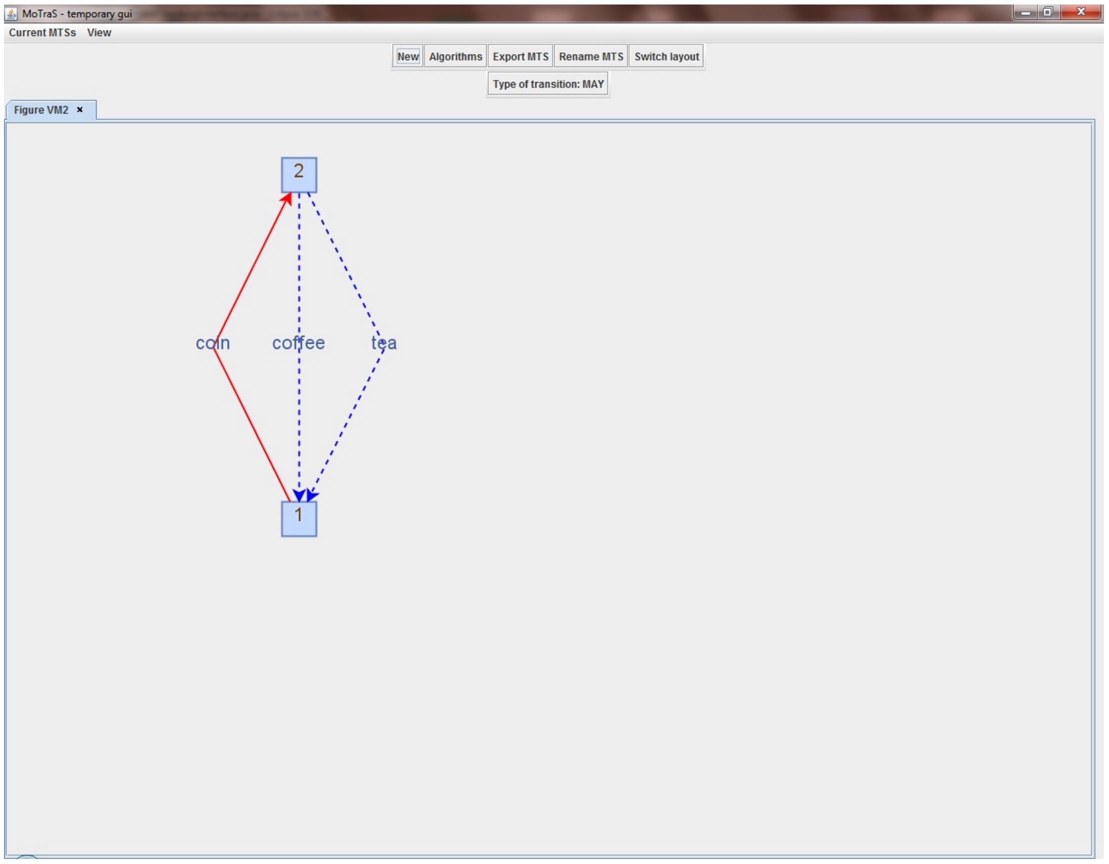


Figure 3.1: A drawn DMTS

the number of processes), choose “LINEAR” and type in “4”). In a last step, insert the number of iterations. The result is a separate line for each run containing the properties of the random MTS(s), the runtime in milliseconds and the size of the computed MTS (in the case of the deterministic hull, the conjunction, the maxCI and the composition) displayed in a text box on a new tab.

3.3 Algorithms and other Operations on a single MTS

3.3.1 Compute Deterministic Hull

Select the option “Compute Deterministic Hull” after clicking on “Algorithms”. Then choose a process of the current active MTS for which the deterministic hull should be computed. The result will be displayed in a new tab.

3.3.2 Perform LTL Model Checking

With the integration of another tool [ws1], MoTraS is able to do LTL Model Checking on DMTSs. In order to do this, choose a process of the current MTS, insert an LTL-formula into the text field and click on “Run”. The result is shown in a new tab. If the output also contains a DMTS definition (an implementation satisfying the formula), the system will be shown graphically in an extra tab.

3.3.3 Generate Random Refinement

Select the option “Generate Random Refinement” after clicking on “Algorithms”. After defining a refinement rate (a value between 0 and 1), a random refinement of the current active MTS will be displayed. The higher the refinement rate, the more the resulting refinement will differ from the refined MTS. The random refinement is computed very naively: Each may-transition of the refinement MTS is changed with a probability constituted by the refinement rate. The transition is either omitted with a 50% chance, or a corresponding must-transition is added in the other case.

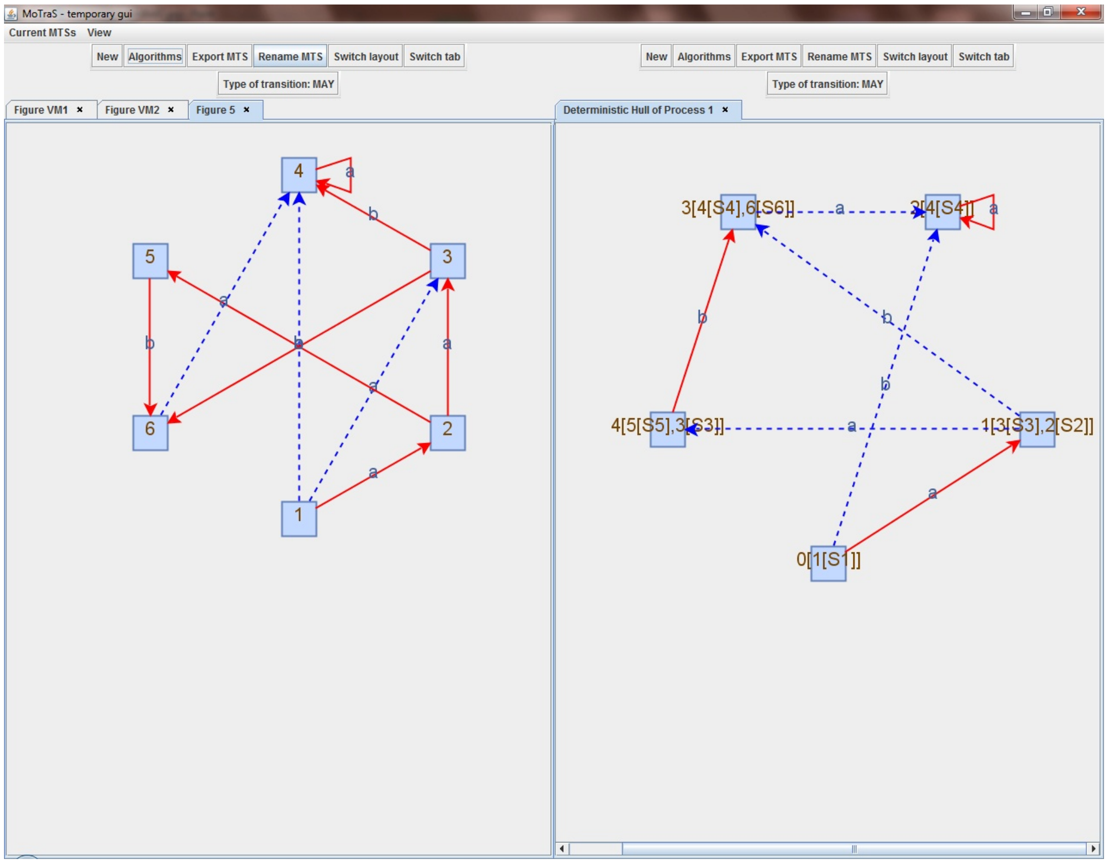


Figure 3.2: The Figure 5 sample and its deterministic hull

3.3.4 Export MTS

Clicking on “Export MTS” opens a tab where the output of the current MTS can be specified. The MTS can be exported to the text field on the tab or to a file that will be saved in the working directory. Moreover, there are three different output formats selectable. If the user wants to use the exported MTS to do model checking by hand with the other MoTraS DMTS-tool [ws1], then the “compatible export” should be chosen. To get a compatible output for the Modal Transition System Analyzer tool [ws4], choose the MTSA-compatible export. In every other case, the “normal export” should be the best option.

3.3.5 Rename MTS

The current active MTS on the GUI can be renamed by clicking on “Rename MTS”.

3.3.6 Switch Layout

The user can switch the layout of the currently displayed MTS by clicking on “Switch Layout”. It will switch between three different layout types. Besides that, there is also the possibility to move the processes by hand.

3.4 Algorithms and other Operations on two MTSs

To use this functionality, switch via the “View” menu to the double GUI first. Switch back to the single GUI mode in the same way. To move tabs to the other side, just click on “Switch Tab”.

Remark: For using the following algorithms, the current active tabs on both sides must be displayed or drawn MTSs. The algorithms are accessible via the menu “Current MTSs”.

3.4.1 Compute Modal Refinement

Choose the processes on the “Modal Refinement” tab. The left process is the refining process, and the right one is the refined. Then click on “Compute”, and the result will be shown in a new tab. If the right process refines the left one, then also the refinement relation will be shown in the text field.

3.4.2 Compute Smallest Common Implementation for dMTS

This algorithm only works for deterministic MTS. Choose two processes and click on “Compute”. If a smallest common implementation was found, it will be displayed in a new tab.

3.4.3 Compute Conjunction for DMTS

This algorithms also works for DMTS. It will show the conjunction of the two selected processes in a new tab.

3.4.4 Compute Greatest Common Implementation

After selecting two processes, the Greatest Common Implementation of them is shown in a new tab.

3.4.5 Compute Composition

After selecting the processes and clicking on “Compute”, the user is asked to add actions to the synchronizing alphabet. If no more actions should be added, just select “NO MORE” and the computation will run.

Chapter 4

Implementation of the Algorithms

In this chapter, we will provide insight into the implementation of the algorithms for the computational problems. After saying a few words to the general ideas, some pseudo code of the algorithms are shown.

4.1 The Modal Refinement Algorithms

There are currently two different algorithms implemented for computing modal refinement: The MRNaive and the MRImproved.

To explain the difference of the algorithms in one sentence: The first algorithm is a pure fixed-point computation, and the second one a mixture between a breadth-first search (BFS) and the first algorithm.

The algorithms 1 and 2 shown below are part of both algorithms.

4.1.1 The MRNaive

This algorithm first creates the Cartesian product of the processes of the refinement MTS and the processes of the refined MTS. Then, the MR-conditions are checked for every pair of processes. If they do not hold for a pair, it will be deleted. This is done until convergence is reached (fixed-point computation). The pseudo code for the MRNaive is shown in Algorithm 3.

Algorithm 1 checkMRConditions((S, T))

```
for all $S \overset{a}{\dashrightarrow} S'$ do
  possibleTransitionFound := FALSE
  for all $T \overset{a}{\dashrightarrow} T'$ do
    if $(S', T') \in R$ then
      possibleTransitionFound := TRUE
      BREAK
    end if
  end for
  if possibleTransitionFound $\equiv$ FALSE then
    return FALSE
  end if
end for
for all $T \xrightarrow{a} T'$ do
  possibleTransitionFound := FALSE
  for all $S \xrightarrow{a} S'$ do
    if $(S', T') \in R$ then
      possibleTransitionFound := TRUE
      BREAK
    end if
  end for
  if possibleTransitionFound $\equiv$ FALSE then
    return FALSE
  end if
end for
return TRUE
```

4.1.2 The MRImproved

The MRImproved first performs a BFS to select all reasonable pairs of processes for the refinement relation, before the fixed-point computation is executed on this smaller set. Algorithm 4 shows the pseudo code for the MRImproved.

4.2 The Deterministic Hull Algorithm

The deterministic hull is computed via a modified BFS that explores the MTS for all reachable $\mathcal{T}$ (the definition of $\mathcal{T}$ is given in the theory chapter). For every $\mathcal{T}$, it looks for possible may-transitions to be added and checks whether a corresponding must-transition

Algorithm 2 fixedPointComputation($R, (S, T)$)

```
convergenceReached := FALSE
while convergenceReached $\equiv$ FALSE do
  convergenceReached := TRUE
  for all $(A, B) \in R$ do
    if checkMRConditions($(A, B)$) $\equiv$ FALSE then
      pop $(A, B)$ from $R$
      convergenceReached := FALSE
    end if
  end for
end while
if $(S, T) \in R$ then
  return TRUE;
else
  return FALSE;
end if
```

Algorithm 3 computeMRNaive((S, T))

```
$R := P_1 \times P_2$
return fixedPointComputation($R, (S, T)$)
```

Algorithm 4 computeMRImproved((S, T))

```
unprocessed := $\{(S, T)\}$
while unprocessed $\not\equiv \emptyset$ do
  pop $(A, B)$ from unprocessed
  add $(A, B)$ to $R$
  for all $A \overset{a}{\dashrightarrow} A'$ do
    for all $B \overset{a}{\dashrightarrow} B'$ do
      if $(A', B') \notin R$ then
        add $(A', B')$ to unprocessed
      end if
    end for
  end for
end while
return fixedPointComputation($R, (S, T)$)
```

could also be inserted. The pseudo code is provided in Algorithm 5.

Algorithm 5 computeDeterministicHull(S)

```
unprocessed := $\{\{S\}\}$
$P_{\mathcal{D}(S)} := \{\{S\}\}$ {add $\{S\}$ as a new process}
while unprocessed $\not\equiv \emptyset$ do
  $\mathcal{T}$ := pop $\{P_1, \ldots P_k\}$ from unprocessed
  for all $a \in \Sigma$ do
    $\mathcal{T}_a := \{P' \mid \exists i : P_i \overset{a}{\dashrightarrow} P'\}$
    if $\mathcal{T}_a \not\equiv \emptyset$ then
      add $\mathcal{T}_a$ to $P_{\mathcal{D}(S)}$ {add $\mathcal{T}_a$ as a new process}
      add $\mathcal{T}_a$ to unprocessed
      mustIsPossible := FALSE
      for all $P_i \in \mathcal{T}$ do
        mustIsPossible := FALSE
        for all $P_i \overset{a}{\dashrightarrow} P'$ do
          if $P' \in \mathcal{T}_a$ then
            mustIsPossible := TRUE
            BREAK
          end if
        end for
        if mustIsPossible $\equiv$ FALSE then
          BREAK
        end if
      end for
      add $\mathcal{T} \overset{a}{\dashrightarrow} \mathcal{T}_a$ to $\dashrightarrow_{\mathcal{D}(S)}$ {insert a may-transition}
      if mustIsPossible $\equiv$ TRUE then
        add $\mathcal{T} \xrightarrow{a} \mathcal{T}_a$ to $\longrightarrow_{\mathcal{D}(S)}$ {insert a must-transition}
      end if
    end if
  end for
end while
```

4.3 The Smallest Common Implementation Algorithm

In this algorithm, the MTSs for the smallest common implementation are explored parallel. If, at some point, a must-transition is required by one of the current processes, and another current process doesn't allow this transition, the algorithm finishes with the result that there is no smallest common implementation. If all other current processes allow the transition, a must-transition is inserted for the smallest common implementation. The pseudo code is shown in Algorithm 6.

Algorithm 6 computeSmallestCommonImplementation($\{D_1, \ldots, D_k\}$)

```
unprocessed := $\{\{D_1, \ldots, D_k\}\}$
$P_{CI}$ := $\{\{D_1, \ldots, D_i\}\}$ {add $\{D_1, \ldots, D_i\}$ as a new process}
while unprocessed $\not\equiv \emptyset$ do
  pop $\{E_1, \ldots, E_k\}$ from unprocessed
  for all $E_i \xrightarrow{a} E_i', i \in \{1, \ldots, k\}$ do
    if $\exists\, j \in \{1, \ldots, k\} : E_j \overset{a}{\not\dashrightarrow} F_j$ then
      return FALSE
    end if
    add $\{F_1, \ldots, F_k\}$ to $P_{CI}$ {add $\{F_1, \ldots, F_k\}$ as a new process}
    pop $\{F_1, \ldots, F_k\}$ from unprocessed
    add $\{E_1, \ldots, E_k\} \overset{a}{\dashrightarrow} \{F_1, \ldots, F_k\}$ to $\dashrightarrow_{\mathrm{CI}}$
    add $\{E_1, \ldots, E_k\} \xrightarrow{a} \{F_1, \ldots, F_k\}$ to $\longrightarrow_{\mathrm{CI}}$
  end for
end while
return TRUE
```

4.4 The Conjunction Algorithm

Like the algorithms before, the conjunction is computed via an extended BFS. Algorithm 7 provides the pseudo code.

4.5 The Greatest Common Implementation Algorithm

With the already existing algorithms, the maxCI can be computed very easily: First, the conjunction algorithm is executed to get the conjunction of the two processes. Then, a must-transition is inserted for every may-transition that is not already realized.

4.6 The Composition Algorithm

The composition is again computed via an extended BFS. First, it is checked if moves for actions from Γ are possible (left and right MTS can move). If positive, a transition is

Algorithm 7 computeConjunction((S, T))

```
unprocessed := $\{(S, T)\}$
$P_{\text{Con}} := \{(S, T)\}$ {add $(S, T)$ as a new process}
while unprocessed $\not\equiv \emptyset$ do
  pop $(A, B)$ from unprocessed
  for all $A \overset{a}{\dashrightarrow} A', a \in \Sigma$ do
    for all $B \overset{a}{\dashrightarrow} B'$ do
      add $(A', B')$ to $P_{\text{Con}}$ {add $(A', B')$ as a new process}
      add $((A, B), a, (A', B'))$ to $\dashrightarrow_{\text{Con}}$
    end for
  end for
  for all $A \longrightarrow \mathcal{A}'$ do
    for all $(a, A') \in \mathcal{A}'$ do
      $\mathcal{C}_{\text{Con}} := \emptyset$
      for all $B \overset{a}{\dashrightarrow} B'$ do
        add $(a, (A', B'))$ to $\mathcal{C}_{\text{Con}}$
        add $(A', B')$ to $P_{\text{Con}}$ {add $(A', B')$ as a new process}
      end for
      add $((A, B), a, \mathcal{C}_{Con})$ to $\longrightarrow_{\text{Con}}$
    end for
  end for
  for all $B \longrightarrow \mathcal{B}'$ do
    for all $(a, B') \in \mathcal{B}'$ do
      $\mathcal{C}_{\text{Con}} := \emptyset$
      for all $A \overset{a}{\dashrightarrow} A'$ do
        add $(a, (A', B'))$ to $\mathcal{C}_{\text{Con}}$
        add $(A', B')$ to $P_{\text{Con}}$ {add $(A', B')$ as a new process}
      end for
      add $((A, B), a, \mathcal{C}_{\text{Con}})$ to $\longrightarrow_{\text{Con}}$
    end for
  end for
end while
```

added to the composition. In a second step, all possible moves from the left and afterwards from the right MTS are checked (for actions $a \in \Sigma \backslash \Gamma$, and so a transition is added if at least one of the processes can move). Algorithm 9 shows the pseudo code.

Algorithm 8 computeMaxCI((S, T))

```
$(P_{\text{maxCI}}, \dashrightarrow_{\text{maxCI}}, \longrightarrow_{\text{maxCI}})$ := computeConjunction($(S, T)$)
**for all** $A \overset{a}{\dashrightarrow} B \in \dashrightarrow_{\text{maxCI}}$ **do**
  **if** $A \xrightarrow{a} B \notin \longrightarrow_{\text{maxCI}}$ **then**
    add $A \xrightarrow{a} B$ to $\longrightarrow_{\text{maxCI}}$
  **end if**
**end for**
```

Algorithm 9 computeComposition((S, T))

```
unprocessed := $\{(S, T)\}$
$P_{Com} := \{(S, T)\}$ {add $(S, T)$ as a new process}
while unprocessed $\not\equiv \emptyset$ do
  $(A, B) \in$ unprocessed
  unprocessed := unprocessed $\setminus\{(A, B)\}$
  for all $A \xrightarrow{a} A', a \in \Gamma$ do
    for all $B \xrightarrow{a} B'$ do
      add $(A', B')$ to $P_{\text{Com}}$ {add $(A', B')$ as a new process}
      add $((A, B), a, (A', B'))$ to $\longrightarrow_{\text{Com}}$
    end for
  end for
  for all $A \xrightarrow{a} A', a \in \Sigma\setminus\Gamma$ do
    add $(A', B)$ to $P_{\text{Com}}$ {add $(A', B)$ as a new process}
    add $((A, B), a, (A', B))$ to $\longrightarrow_{\text{Com}}$
  end for
  for all $B \xrightarrow{a} B', a \in \Sigma\setminus\Gamma$ do
    add $(A, B')$ to $P_{\text{Com}}$ {add $(A, B')$ as a new process}
    add $((A, B), a, (A, B'))$ to $\longrightarrow_{\text{Com}}$
  end for
  for all $A \overset{a}{\dashrightarrow} A', a \in \Gamma$ do
    for all $B \overset{a}{\dashrightarrow} B'$ do
      add $(A', B')$ to $P_{\text{Com}}$ {add $(A', B')$ as a new process}
      add $((A, B), a, (A', B'))$ to $\dashrightarrow_{\text{Com}}$
    end for
  end for
  for all $A \overset{a}{\dashrightarrow} A', a \in \Sigma\setminus\Gamma$ do
    add $(A', B)$ to $P_{\text{Com}}$ {add $(A', B)$ as a new process}
    add $((A, B), a, (A', B))$ to $\dashrightarrow_{\text{Com}}$
  end for
  for all $B \overset{a}{\dashrightarrow} B', a \in \Sigma\setminus\Gamma$ do
    add $(A, B')$ to $P_{\text{Com}}$ {add $(A, B')$ as a new process}
    add $((A, B), a, (A, B'))$ to $\dashrightarrow_{\text{Com}}$
  end for
end while
```

Chapter 5

Runtime Tests on the Algorithms

As already mentioned before, the size of an MTS has a very big influence on the runtime of the algorithms. In this chapter, this will be analyzed in more detail by testing the algorithms on various random MTSs. The variable parameters are the total number of processes and the total number of transitions of an MTS. In concrete, the algorithms are tested with different numbers of processes (incremental, with an increment of 5) and with three different transitions-processes ratios: $1.5n$, $2n$ and $0.25n^2$, whereby n is the number of processes. This means that e.g. for the $0.5n^2$ ratio, an MTS with 50 processes will have $0.25 \cdot 50^2 = 625$ transitions altogether. In almost all tests, the randomly generated MTSs have an alphabet size of 3. For each combination of the number of processes and the number of transitions, the algorithm is run on 10 different random MTSs to ensure quite significant results. For the final value, the median of the runtime for each combination is taken.

Remark: In general, taking the mean value instead of the median shouldn't make a big difference, although the algorithm for the deterministic hull is an exception for that, which will be described later.

5.1 The MR Algorithms

For each of the 10 random MTSs for each combination, a refining MTS is created with the help of the MTSGenerator (refinement rate 0.3). The three diagrams in Figure 5.1 show the test results for the MRNaive and the MRImproved algorithm for the tree different transitions-processes ratios. Moreover, one pair of MTSs for each combination (but only for every other increment of the number of processes) is also tested externally on the "strong refinement"-algorithm of the MTSA tool [ws4] (the strong refinement is equal to the modal refinement). As MTSA doesn't allow processes that have no outgoing transitions, a self-transition has been added to every processes (which doesn't affect the result and shouldn't effect a significantly longer computation time).

Remark: Analyzing the exact runtime results given in milliseconds is not the aim of this paper (as this depends on the hardware configuration of the computer on the one hand and could not be determined correctly for small MTSs on the other hand). The significant results are visible by combining just the trends of the curves for the two algorithms.

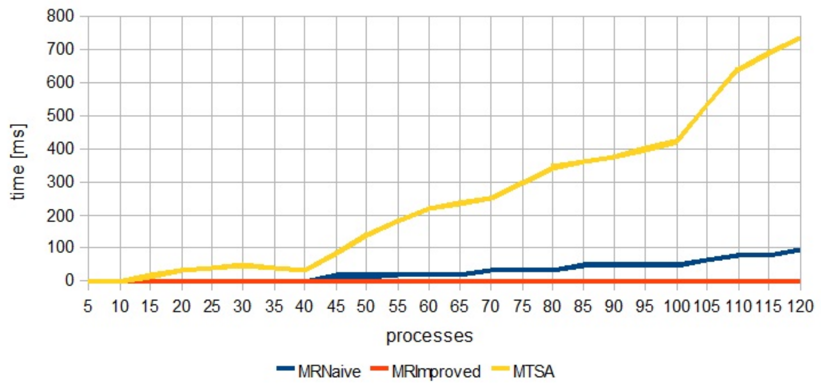
The main result that can be derived from the diagrams: In the case of the linear transition-processes ratio, the MRImproved is faster than the MRNaive, as the BFS on a linear number of transitions together with the fixed-point computation on the remaining reasonable pairs is faster than the fixed-point computation on the whole Cartesian product of the processes. But as in general the BFS is slower than the fixed-point computation, the MRNaive works better for MTSs with a high transitions-processes ratio. The MTSA algorithm isn't able to compete at all.

Testing the algorithms on negative examples will show another advantage of the MRImproved. As in this case, the runtime of the MRImproved is greatly affected by the structure and the similarity of the two MTS, a test like the one above doesn't make much sense here. But it is intuitively clear (when having the implementation of the two algorithms in mind), that the number of reasonable pairs of processes is strongly decreased, if the MTSs are very different. By contrast, the MRNaive algorithm is not affected by this (as it always checks the whole Cartesian product of the processes). Lets make an extreme example for this:
Let M_1, M_2 be two (nearly identical) MTSs with 100 processes and 500 transitions each and let S_i, T_i be processes of M_i for $i \in \{1, 2\}$ with the following properties:

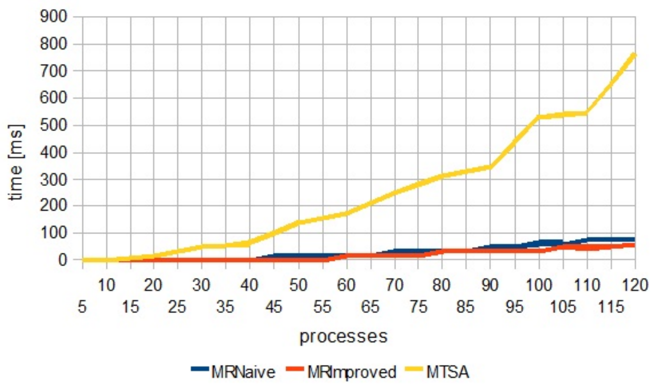
- S_i has no incoming transitions
- T_i has only one incoming transition, which is $S_i \xrightarrow{a_i}_i T_i$ (let this be the only structural difference between M_1 and M_2

Due to construction, it is clear that $S_1 \leq_m S_2$ does not hold. Whereby the MRNaive will obstinately compute the Cartesian product and check each pair, the MRImproved comes to the correct result within two steps (one for the BFS and one for the MR-conditions test), as the only reasonable pair is (S_1, S_2). This can be stated with the result of a confirming run for the MTSs defined above.

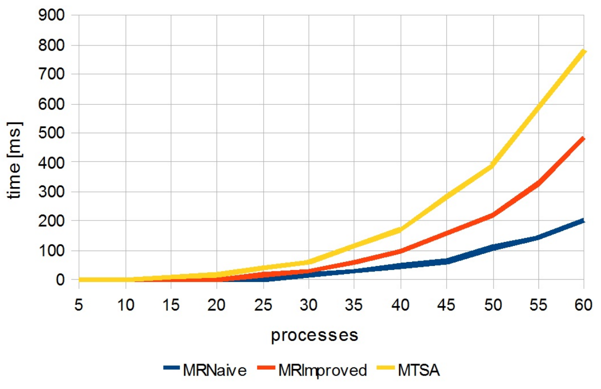
```
Result for MRNaive: false; steps: 12082; time taken: 125ms
Result for MRImproved: false; steps: 1 + 1; time taken: 0ms
```



(a) transitions-processes ratio $1.5n$



(b) transitions-processes ratio $2n$



(c) transitions-processes ratio $0.25n^2$

Figure 5.1: The test result for the three MR algorithms

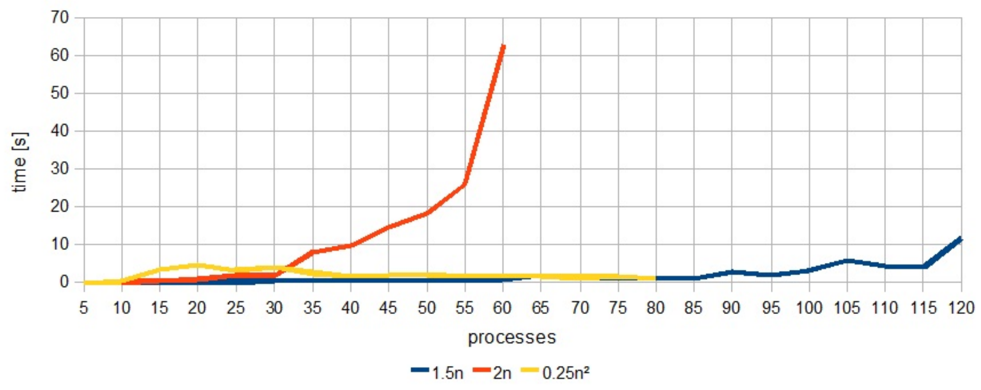


Figure 5.2: The test result for the deterministic hull algorithm

5.2 The Algorithm for computing the Deterministic Hull

The total number of processes for this test is ranging from 10 to 120 for the $1.5n$ ratio, and 5 to 60 with increment 5 for the $2n$ and the $0.25n^2$ ratio, as the computation for bigger MTS would take too much time. The result is shown in Figure 5.2.

Especially the curve for the 0.25-ratio is quite inexplicable. But we have found out that the runtime corresponds to the size of the resulting deterministic hull. This leads to the question why the algorithm produces smaller MTSs for larger input MTSs. Finding an answer for this problem remains open in this paper and could be an approach for later research and work with the tool.

5.3 The Algorithm for computing the Smallest Common Implementation

This algorithm is tested with a random dMTS as the first and a 0.3-refinement as the second MTS. The two test series with linear ratio are performed with up to 6 different actions, but for the quadratic ratio a very huge set of different actions is required to get a deterministic MTS with so many transition, so we allowed 120 different actions for the random MTSs. The result is shown in Figure 5.3.

The reason for the very fast computation in the linear case derives from the fact that the resulting composition has a very constant number of processes and transitions, as only

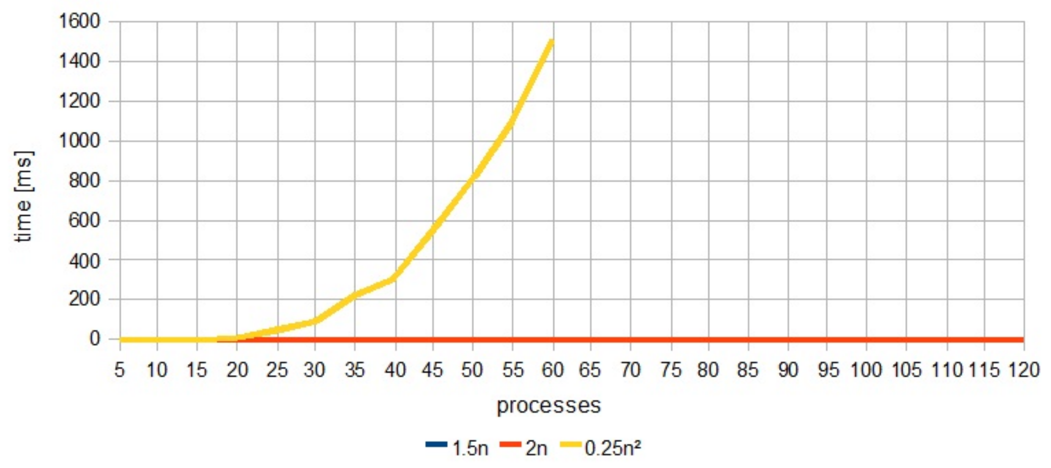


Figure 5.3: The test result for the smallest common implementation algorithm - the curve for $1.5n$ is not visible, as it is the same as the $2.0n$ curve

those transitions are inserted that are required by must-transitions. So there are only a few of those that are reachable from the initial pair of processes during the BFS.

5.4 The Algorithm for computing the Conjunction

This algorithm is tested on random DMTSs. Due to the fact that the computation time for the conjunction is very high (e.g. when combining it to the smallest common implementation), it could only be performed on small MTSs for the higher transitions-processes ratios.

The reason for those small ups and downs in the 1.5-graph is the instability of the algorithm. Different MTSs of the same size with a different structure can cause a runtime that differs with a factor of 100 or even more. Those differences in structure could be the number and the location of circles in the MTS or even just the ratio of may- and must-transitions, for example.

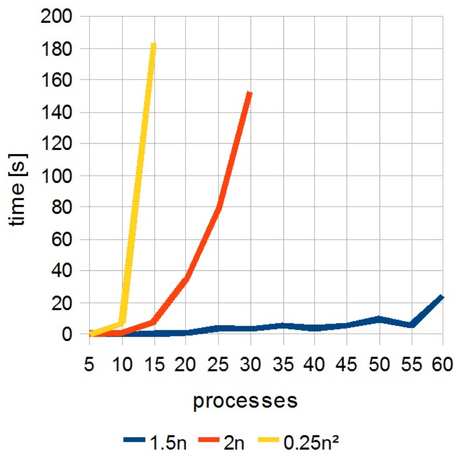


Figure 5.4: The test result for the conjunction algorithm

5.5 The Algorithm for computing the Greatest Common Implementation

The computation of the greatest common implementation consists of to steps. The first (big) one is the computation of the conjunction and the second (small) step is solely the addition of several must-transitions. The runtime for this algorithm is almost the same as for the conjunction with the exception of a plus of 10 to 20 percent for this additional step. Hence, we do not show the runtime test results in a graph here.
A more interesting result can be obtained by comparing this algorithm to the smallest common implementation algorithm, when both algorithms are executed on the same (d)MTSs. Figure 5.5 shows the result for the 1.5-transitions-processes ratio. Here, one can see the great advantage of having an algorithm that indeed works for deterministic MTSs only, but is therefore much faster than the general one.

Remark: Although the two algorithms do not compute exactly the same thing, both results are common implementations and not so different in structure and size. Therefore, these two algorithms can be compared here.

5.6 The Algorithm for computing the Composition

This time, two different MTSs (with the same parameters) are created for each test. The synchronizing alphabet is always $\Gamma = \{a0\}$, whereby $\Sigma = \{a0, a1, a2\}$. Figure 5.6 shows

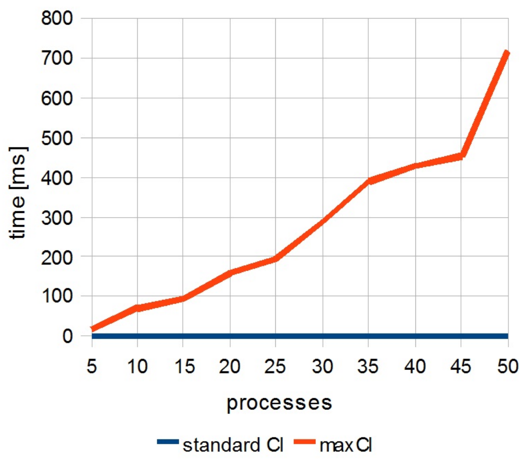


Figure 5.5: The comparison of the smallest and the greatest common implementation algorithm

the (expected) result.

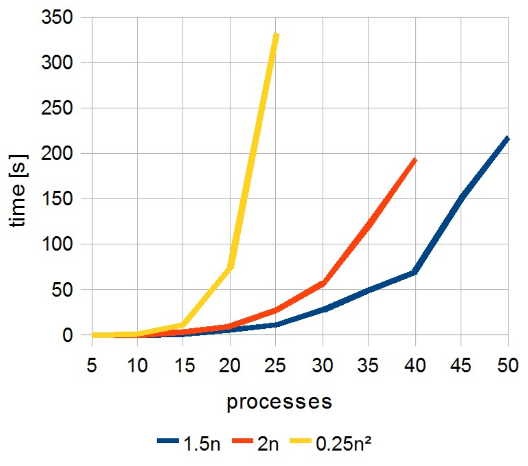


Figure 5.6: The test result for the composition algorithm

Chapter 6

Conclusion and Possible Future Work

"In theory, there is no difference between theory and practice. But in practice, there is." (Yoga Berra)

The intention of this work is to make the still quite unknown concept of MTSs more public and also to provide a tool for scientists and project managers to use this powerful theory in practice, e.g. to administrate software specifications or to control workflow. The main advantage of the MTS theory is the fact that it is a simple extension of the well-known basic graph theory that a lot of people use in practice, and therefore this enhanced concept can also be used without the need of learning and understanding much more theoretical stuff.

Moreover, it shall be a work that can be easily extended by scientists to enrich the already huge functionality of the tool with more concepts, algorithms and improvements. Hereinafter, a suggestion for possible future work on the tool is given.

6.1 The MRMartin

As already mentioned in chapter 2.2.1, the MRMartin is a third implemented algorithm to compute modal refinement, which unfortunately still produces wrong results in a couple of cases. The MRMartin has been programmed to have a way for computing MR without the need of a fixed-point computation, but solely by a recursive extended BSF. We originally wanted to test the MRMartin to the MRImproved and see whether this algorithm is faster for MTSs with many processes and a low processes-transitions ratio. A future work on this algorithm can now either be the correction of the mistake(s) in the code or even a proof that the approach of the algorithm cannot lead to a correct MR computation.

6.2 Extension of Algorithms for DMTSs

At the moment, all algorithms except for the conjunction work for normal MTSs only. This tool already provides a data structure and basic operations for disjunctive transitions, so it should be not so difficult for later developers to extend the algorithms for DMTSs. A prior step could be the extension of the smallest common implementation algorithm for normal MTSs or the implementation of an algorithm which computes a dCI, a deterministic common implementation of a set of processes.

6.3 Algorithms for checking Thorough Refinement

Analogously to the MRNaive, there have also been plans for a TRNaive algorithm. Some necessary data structures and the basic framework for this algorithm are already part of the source code of the tool, but unfortunately the work on this part of MoTraS could not be finished by now. The TRNaive was planned as an algorithm to compute thorough refinement in a very naive way, so there are also many improvements and speedups imaginable to be realized in a "TRImproved".

6.4 Extension and Improvement of the Random MTS Generator

As the current implementation of the MTSGenerator is very naive, several extensions can be taken into account. An additional feature could be the possibility to create more customized MTSs, e.g. with a given number of loops or a given probability distribution for the set of actions. Moreover, the refinement generator could also be extended. An example for this is the possibility for a may-transition to be refined into several successive must-transition steps or splitting up a may-transition to several may-transitions pointing to different targets.

6.5 LTL Model Checking

Although MoTraS already provides a basic functionality for model checking with the help of an integrated C program [ws1], the model checking theory contains a few more concepts than the tool currently is able to handle. A direct realization in the tool would also have the advantage that MoTraS could be provided as a platform-independent software without the need of different versions.

6.6 Other Possible Features and Extensions

While working with the tool, a few suggestions for improvements and useful features have shown up that couldn't be realized in the end, which are mentioned in the following list:

- a button to exchange the tabs on both sides of the GUI
- deleting processes and transitions via typing the delete key
- displaying the refinement relation directly in the systems (e.g. as differently colored nodes)

Bibliography

[BCK] Nikola Benes, Ivana Cerna, and Jan Kretinsky. Modal Transition Systems: Composition and LTL Model Checkings.

[BKLS09a] Nikola Benes, Jan Kretinsky, Kim Guldstrand Larsen, and Jiri Srba. Checking thorough refinement on modal transition systems is exptime-complete. Technical report FIMU-RS-2009-03, Faculty of Informatics, Masaryk University, Brno, 2009.

[BKLS09b] Nikola Benes, Jan Kretinsky, Kim Guldstrand Larsen, and Jiri Srba. On determinism in modal transition systems. *Theor. Comput. Sci.*, 410(41):4026–4043, 2009.

[soc] sockeqwe. Extended JCloseTabbedPane. ”http://www.java-forum.org/awt-swing-swt/76647-extended-jtabbedpane.html”.

[ws1] MoTraS - (D)MTS tool. ”http://anna.fi.muni.cz/∼xbenes3/MoTraS/”.

[ws2] ltl2dstar - LTL to deterministic Streett and Rabin automata. ”http://www.ltl2dstar.de/”.

[ws4] MTSA Modal Transition System Analyser. ”http://lafhis.dc.uba.ar/∼suchitel/MTSA.html”.

[ws5] Javal Graph Drawing Component. ”http://www.jgraph.com/jgraph.html”.

[ws6] yED Graph Editor. ”http://www.yworks.com/en/products_yed_about.html”.

[ws7] Open Office Calc. ”http://www.openoffice.org/product/calc.html”.