Formal Methods Technical Report

Jane Jovanovski, Maja Siljanoska, Vladimir Carevski,
Dragan Sahpaski, Petar Gjorcevski, Metodi Micev,
Bojan Ilijoski, and Vlado Georgiev

Institute of Informatics,
Faculty of Natural Sciences and Mathematics,
University "Ss. Cyril and Methodius",
Skopje, Macedonia
jane.jovanovski@gmail.com,maja.siljanoska@gmail.com,carevski@gmail.com,
dragan.sahpaski@gmail.com,pgorcevski@t-home.mk,metodi.micev@gmail.com,b.
ilijoski@gmail.com,vlatko.gmk@gmail.com

Abstract. The Calculus of Communicating Systems (CCS) is a young branch in process calculus introduced by Robin Milner. CCS's theory has promising results in the field of modeling and analysing of reactive systems. Despite CCS's success, there are only few computer tools that are available for modeling and analysis of CCS's theory. This paper is a technical report of an effort to build such a tool. The tool is able to parse CCS's expressions, it can build LTSs (Labelled Transition Systems) from expressions and it can calculate if two LTS are bisimular by using naive method or by using Fernandez's method. The tool is simple, but it has the functionality needed to perform modeling, specification and verification of certain system.

Keywords: Reactive Systems, CCS, Formal Methods, Process Algebra

Acknowledgment

The authors would like to thank Jasen Markovski for being their mentor...

Table of Contents

List of Figures

List of Tables

1 Introduction

This paper is organized in four chapters. The first chapter makes an introduction to the basic terminology used throughout the report. The second chapter is devoted to the problem of parsing CCS's expressions and generation of Labelled Transition System (LTS) and also discusses some of the choices made during the implementaion. The third chapter discusses the problem of saturation of LTS and implementation details. The fourth chapter discusses the problem of implementing the two methods for determining bisimilarity between two LTS graphs, the first method is the so called naive method and the second is the method of Fernandez. The fifth chapter shows typical examples of modeling, specification and verification of example systems. The first example system is Alternating bit protocol and the second one is mutex algorithm defined by Petersen and Hamilton.

The tool is a Java executable jar library with very simple user interface.

2 Parsing and LTS generation

Given a set of action names, the set of CCS processes is defined by the following BNF grammar:

$$P ::= \emptyset \mid a.P_1 \mid A \mid P_1 + P_2 \mid P_1|P_2 \mid P_1[b/a] \mid P_1 \backslash a \tag{1}$$

Two choices were considered for the problem of building a parser. The first choice was to build a new parser, which required much more resources and the second choice was to define a grammar and to use a parser generator (compiler-compiler, compiler generator) software to generate the parser source code. In formal language theory, a context-free grammar (CFG) is a grammar in which every production rule has the form:

$$V \rightarrow w$$

where V e is a nonterminal symbol, and w is a string of terminal and/or nonterminal symbols (w can be empty). Obviously, the defined BNF grammar for describing the CCS process is a CFG. Deterministic context-free grammars (DCFGs) are grammars that can be recognized by a deterministic pushdown automaton or equivalently, grammars that can be recognized by a LR (left to right) parser. DCFGs are a proper subset of the context-free grammar family. Although, the defined grammar is a non-deterministic context-free grammar, modern parsers have a look ahead feature, which means that the parser will not make a parsing decision until it reads ahead several tokens. This feature alows us to use a parser generator software, that will generate LR or LL parser which is able to recognize the non-deterministic grammar.

ANTLR (ANother Tool for Language Recognition) [Par07] is a parser generator that was used to generate the parser for the CCS grammar. ANTLR uses LL(*) parsing and also allows generating parsers, lexers, tree parsers and

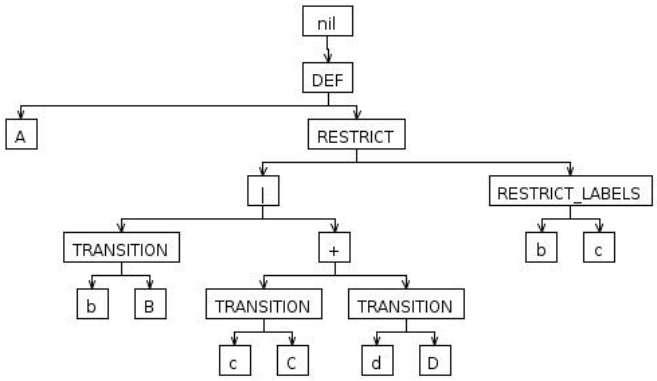

Fig. 1. AST Example

combined lexer parsers. It also automatically generates abstract syntax trees (AST), by providing operators and rewrite rules to guide the tree construction, which can be further processed with a tree parser, to build an abstract model of the AST using some programming language. A language is specified by using a context-free grammar which is expressed using Extended Backus Naur Form (EBNF). In short, an LL parser is a top-down parser which parses the input from left to right and constructs a Leftmost derivation of the sentence. An LL parser is called an LL(*) parser if it is not restricted to a finite k tokens of lookahead, but can make parsing decisions by recognizing whether the following tokens belong to a regular language. This is achieved by building temporary deterministic finite automaton, which are maintained and used until the parser makes a decision. This feature and some more optimization for the LL parser were published in recent years which made this kind of parser generators popular and favorable [NW].

ANTLR was used to generate lexer, parser and a well defined AST which represents a tree representation of the abstract syntactic structure of the parsed sentence. The term abstract is used in a sense that the tree does not represent every single detail that exists in the language syntax, for example, parentheses are implicit in the tree structure. From a development point of view it is much easier to work with trees than with a list of recognized tokens. One example of an AST is shown in Fig. 1 which is the result of parsing the expression:

$$A = (b.B \mid c.D \ + \ d.D) \backslash \{b, \ c\} \tag{2}$$

Although working directly with ASTs and performing all algorithms on them is possible, it causes a limitation in future changes. Even a small change in the grammar and/or on the structure of the generated abstract syntax trees causes a change in the implemented algorithms. Because of this, a specific domain model is built along with a domain builder algorithm. The input of the domain builder algorithm is a AST and the output is a fully built domain model. It is not ex-

pected for the domain model to change, so all algorithms are implemented on the domain model. The domain model also is a tree-like structure, so it is as easy to work with, as with AST. The algorithms for LTS generation is built using the domain model. The LTS generation algorithm is recursive algorithm which traverses the tree structure and performs SOS rule every time it reaches operation. In this fashion all sos transformation are preformed on the domain and as result a new tree structure is created which represents the LTS. This tree structure can be easily exported to file in Aldebaran format.

3 Bisimulation equivalence

Bisimulation equivalence (bisimilarity)[1] is a binary relation between labeled transition systems which associates systems that can simulate each other's behaviour in a stepwise manner. This enables comparison of different transition systems. An alternative perspective is to consider bisimulation equivalence as a relation between states of a single labelled transition system. By considering the quotient transition system under such a relation, smaller models are obtained [BK08].

The bisimulation equivalence finds its extensive application in many areas of computer science such as concurrency theory, formal verification, set theory, etc. For instance, in formal verification minimization with respect to bisimulation equivalence is used to reduce the size of the state space to be analyzed. Also, bisimulation equivalence is of particular interest in model checking, in specific to check the equivalence of an implementation of a certain system with respect to its specification model.

Our tool implements both options: reducing the size of the state space of a given LTS (minimization modulo bisimilarity) and checking the equivalence of two labeled transition systems modulo bisimilarity.

3.1 Minimization of an LTS modulo bisimilarity.

The process of reducing the size of the state space of a certain labeled transition system was implemented using an approach which consists of two steps:

1. Computing strong bisimulation equivalence (strong bisimilarity) for the LTS;
2. Minimizing the LTS to its canonical form using the strong bisimilarity obtained in the first step;

The first step, computing strong bisimulation equivalence, was implemented with two different methods: the so called naive method and a more efficient method due to Fernandez, both of which can serve as minimization procedures.

The naive algorithm [AILS07a] for computing bisimulation equivalence stems from the theory underlying Tarski's fixed point theorem [AILS07b]. It has been

[1] The notion of bisimulation equivalence (bisimilarity) in this chapter refers to strong bisimulation equivalence (strong bisimilarity)

proven that the strong bisimulation equivalence is the largest fixed point of the monotic function F as defined in [AILS07a] given by Tarsky's fixed point theorem.

The labeled graph was represented as a list of nodes and the following terminology was used:

- $S_p = \{(a, q)\}$ - set of pairs (a, q) for state p where a is an outgoing action for p and q is a state reachable from p with the action a

Our Java implementation of the algorithm takes as input an LTS in aldebaran format, generates a corresponding labeled graph and then computes the strong bisimulation equivalence as pairs of bisimilar states.

This algorithm has time complexity of $O(mn)$ for a labeled transition system with m transitions and n states.

The algorithm due to Fernandez exploits the idea of the relationship between strong bisimulation equivalence and the relational coarsest partition problem solved by Paige and Tarjan. It represents adaptation of the Paige-Tarjan algorithm of complexity $O(m \log n)$ to minimize labeled transition systems modulo bisimulation equivalence by computing the coarsest partition problem with respect to the family of binary relations $(T_a)_{a \in A}$ instead of one binary relation, where $T_a = \{(p, q) | (p, a, q) \in T\}$ is a transition relation for action $a \in A$ and T is a set of all transitions [PT87][Fer89].

The algorithm due to Fernandez in our Java implementation takes an LTS in aldebaran format as an input, generates a corresponding labeled graph and then partitions the labeled graph into its coarsest blocks where each block represents a set of bisimilar states. Partition is a set of mutually exclusive blocks whose union constitutes the graph universe.

To define graph transitions the following terminology was used:

- $T_a[p] = \{q\}$ - an a-transition from state p to state q
- ${T_a}^{-1}[q] = \{p\}$ - an inverse a-transition from state q to state p
- ${T_a}^{-1}[B] = \cup \left\{{T_a}^{-1}[q], q \in B\right\}$ - inverse transition for block B and action a
- W - set of sets called splitters that are being used to split the partition
- infoB(a, p) - info map for block B, state p and action a

The time complexity of Fernandez's algorithm is $O(m \log n)$ for a labeled transition system with m transitions and n states.

The next step in the reduction of the state space of an LTS uses the bisimulation equivalence computed in the first step in order to minimize the labeled graph. This reduction is implemented as follows:

1. If a pair of states (p, q) is bisimilar, then the two states are merged into one single state k;
2. All incoming transitions $r \xrightarrow{a} p$ and $s \xrightarrow{a} q$ are replaced by transitions $r \xrightarrow{a} k$ and $s \xrightarrow{a} k$;
3. All outgoing transitions $p \xrightarrow{a} r$ and $q \xrightarrow{a} s$ are replaced by transitions $k \xrightarrow{a} r$ and $k \xrightarrow{a} s$;
4. The duplicate transitions are not taken into consideration.

The procedure is repeated for all pairs of bisimilar states.

The process of reducing a given labeled graph module strong bisimilarity is illustrated below for the labeled graph in Fig. 2.

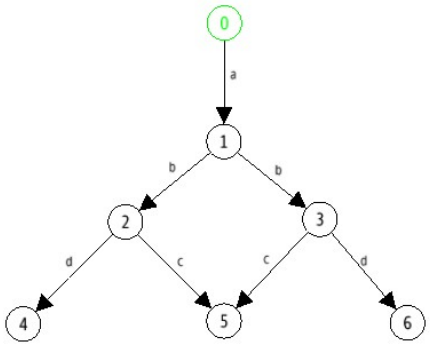

Fig. 2. Graph 1

Applying both the naive and the advanced algorithm due to Fernandez for computing strong bisimulation equivalence for Graph 1, gives the results shown in Table 1.

Algorithm	Graph 1
Naive	(2, 3), (3, 2), (4, 5), (5, 4), (4, 6), (6, 4), (5, 6), (6, 5), (0, 0), (1, 1), (2, 2), (3, 3), (4, 4), (5, 5), (6, 6)
Fernandez	{0}, {1}, {2}, {3}, {4, 5, 6}

Table 1. Computing strong bisimularity for Graph 1

The results obtained with either of the two algorithms for computing strong bisimilarity are then used as a basis for the reduction of the given graph to its minimal form. Namely, all mutually bisimilar states are merged into a single state and their transitions are updated accordingly. The reduction of Graph 1 to its canonical form with respect to the bisimulation equivalence is given in Fig. 3. As it can be seen from the figure, the states 2 and 3 are merged into state 2 in the minimal graph, and states 4, 5 and 6 are merged into state 3 in the minimal graph.

3.2 Comparison of two LTSs modulo bisimilarity.

The idea for the implementation of the equivalence checking of two labeled transition systems modulo strong bisimilarity was based on the following fact: Two

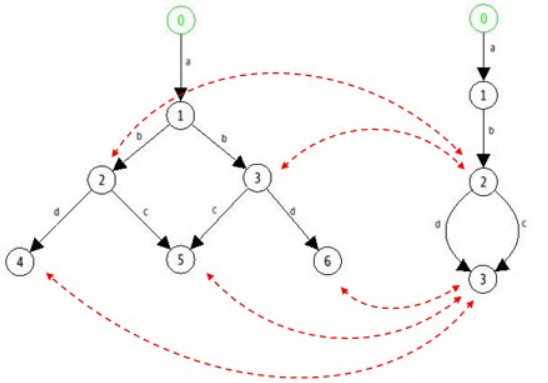

Fig. 3. Minimized Graph 1

labelled transition systems are (strongly) bisimilar iff their initial states are bisimilar [GR09].

That means that in order to check whether two labeled transition systems are bisimilar it is enough to check whether their initial states are bisimilar. This can be done using the following approach:

1. The two labeled transition systems are merged into a single transition system
2. An algorithm for computing the strong bisimilarity is applied to the merged system
3. A check is performed to see if the initial states belong to the same bisimulation equivalence class

The correctness of the implementation was tested with the use of ltsconvert and ltscompare tools of mCRL2, micro Common Representation Language 2, a specification language that can be used to specify and analyse the behaviour of distributed systems and protocols [mCRL2Ref].

4 Saturation and weak bisimulation equivalence

Def: Let P and Q be CCS processes or, more generally, states in a labeled transition system. For each action a, we shall write $P \stackrel{a}{\Rightarrow} Q$ iff:

- Either $a \neq \tau$ and there are processes P' and Q' such that $P \left(\stackrel{\tau}{\rightarrow}\right)^* P' \stackrel{a}{\rightarrow} Q' \left(\stackrel{\tau}{\rightarrow}\right)^* Q$,
- Or $a = \tau$ and $P \left(\stackrel{\tau}{\rightarrow}\right)^* Q$,

where we write $\left(\stackrel{\tau}{\rightarrow}\right)^*$ for he reflexive and transitive closure of the relation $\stackrel{\tau}{\rightarrow}$.

Def: (weak bisimulation and observational equivalence) A binary relation R over the set of states of an LTS is a weak bisimulation iff, whenever $s_1 R s_2$ and a is an action (including τ):

– If $s_1 \xrightarrow{a} s_1'$ then there is a transition $s_2 \stackrel{a}{\Rightarrow} s_2'$ such that $s_1' R s_2'$;
– If $s_2 \xrightarrow{a} s_2'$ then there is a transition $s_1 \stackrel{a}{\Rightarrow} s_1'$ such that $s_1' R s_2'$;

Two states s and s' are observationally equivalent (or weakly bisimilar), written $s \approx s'$, iff there is a weak bisimulation equivalence that relates them.

Def: Let $T \subseteq S \times Act \times S$ be an LTS. We shall say that
$T^* = \left\{(p,a,q)\,|p \stackrel{a}{\Rightarrow} q\right\} = T \cup \left(\xrightarrow{a}\right)^* \cup \left\{(p,a,q)\,|a \neq \tau \wedge (\exists p', q' \in A)\, p \left(\xrightarrow{\tau}\right)^* p' \xrightarrow{a} q' \left(\xrightarrow{\tau}\right)^* q\right\}$
is a saturation of T.

Proposition: Two LTSs are weakly bisimilar iff their saturated LTSs are strongly bisimilar.

Proof: Let T and U be two LTSs for which their saturated LTSs are strongly bisimilar by the relation R. Since the strong bisimulation is also a weak bisimulation it follows that T and U are weakly bisimular, since they are weakly bisimular to their respective saturated LTSs. Let T and U be two LTSs that are weakly bisimilar. Let $u \approx t$, $u' \approx t'$. If $t \xrightarrow{a} t' \in T^*$, that means $t \stackrel{a}{\Rightarrow} t'$, or that there exist states t_i, t_j' such that: $t \xrightarrow{\tau} t_1 \xrightarrow{\tau} ... \xrightarrow{\tau} t_k \xrightarrow{a} t_1' \xrightarrow{\tau} ... \xrightarrow{\tau} t_m' \xrightarrow{\tau} t'$. Let $u_i \approx t_i$,...,$u_j' \approx t_j'$,.... It follows that $u \stackrel{\tau}{\Rightarrow} u_1 \stackrel{\tau}{\Rightarrow} ... \stackrel{\tau}{\Rightarrow} u_k \stackrel{a}{\Rightarrow} u_1' \stackrel{\tau}{\Rightarrow} ... \stackrel{\tau}{\Rightarrow} u'$, or that there exist states u'', u''' in U such that $u \left(\xrightarrow{\tau}\right)^* u_k \left(\xrightarrow{\tau}\right)^* u'' \xrightarrow{a} u''' \left(\xrightarrow{\tau}\right)^* u_1' \left(\xrightarrow{\tau}\right)^* u'$ which means $u \stackrel{a}{\Rightarrow} u'$. Since U^* is a saturation of U, it follows that $u \xrightarrow{a} u' \in U^*$. Similarly, for every transition $u \xrightarrow{a} u' \in U^*$, there is a transition in $t \xrightarrow{a} t' \in T^*$, such that $u \approx t$, $u' \approx t'$. Therefore, the weak bisimulation relation $\approx$ for T and U is a strong bisimulation relation for T^* and U^*.

4.1 The algorithm for saturation.

The set of triples R can be partitioned in $2n$ sets with:
$T_{\tau p} = \{(p,\tau,q)\,|\,(p,\tau,q) \in T\}$, and
$T_{ap} = \{(p,\tau,q)\,|\,a \neq \tau \wedge (p,\tau,q) \in T\}$ for every $p \in S$.

By the definition of $T_{\tau p}$ and T_{ap} it can be seen that $\bigcup_{p\in S}(T_{\tau p} \cup T_{ap}) = T$, and also, their pairwise intersection is empty.

The family of sets $T^*_{\tau p}$ can be iteratively constructed with:
$T^0_{\tau p} = T_{\tau p} \cup \{(p,\tau,p)\}$,
$T^i_{\tau p} = T^{i-1}_{\tau p} \cup \left\{(p,\tau,r)\,|\,(\exists q \in S)\,(p,\tau,q) \in T^{i-1}_{\tau p} \wedge (q,\tau,r) \in T_{\tau q}\right\}$ and
$T^*_{\tau p} = T^n_{\tau p}$

(Note: $\left|T^*_{\tau p}\right| \leq |S| = n$; when for some $k < n$, $T^k_{\tau p} = T^{k+1}_{\tau p}$, then $T^*_{\tau p} = T^k_{\tau p} = T^n_{\tau p}$)

With this step a reflexive, transitive closure was constructed: $T^*_\tau = \bigcup_{p\in A} T^*_{\tau p} = \left\{(p,\tau,q)\,|p \left(\xrightarrow{\tau}\right)^* q\right\}$.

The next step is to construct $T'_{ap} = \bigcup_{p\in A} T'_{ap} = \left\{(p,a,q)\,|\,(\exists q' \in S)\,(p,a,q') \in T \wedge q' \left(\xrightarrow{\tau}\right)^* q\right\}$:
$T^0_{ap} = T_{ap'}$,
$T^i_{ap} = T^{i-1}_{ap} \cup \left\{(p,a,r)\,|\,(\exists q \in S)\,(p,a,q) \in T^{i-1}_{ap} \wedge (q,\tau,r) \in T^{i-1}_{\tau q}\right\}$ and
$T'_{ap} = T^{n|Act|}_{ap}$

(Note: $|T'_{ap}| \leq |S||Act| = n|Act|$, when for some $k < n|Act|$, $T^k_{ap} = T^{k+1}_{ap}$, then $T'_{ap} = T^k_{ap} = T^{n|Act|}_{ap}$)

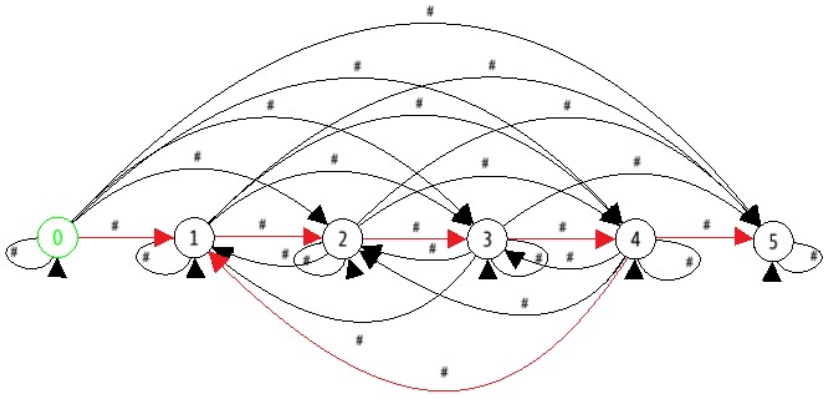

Fig. 4. Reflexive, transitive closure of τ. The original graph is depicted with red lines

For the third step, $T' = \bigcup_{p \in S} \left(T^*_{\tau p} \cup T'_{ap}\right)$ needs to be partitioned again, defined by the destination in the triple:
$T_{\tau q} = \{(p, \tau, q) \,|\, (p, \tau, q) \in T'\}$ and $T_{bq} = \{(p, a, q) \,|a \neq \tau \wedge (p, a, q) \in T'\}$ for every $p \in S$,
and then construct:
$T^0_{bq} = T_{bq}$,
$T^i_{bq} = T^{i-1}_{bq} \cup \left\{(p', a, q) \,|\, (\exists p \in S)\, (p, a, q) \in T^{i-1}_{bq} \wedge (p', \tau, p) \in T^*_{\tau p}\right\}$ and
$T^*_{bq} = T^{n|Act|}_{bq}$

Finally the saturated LTS is:
$T^* = \bigcup_{p \in S} \left(T^*_{\tau p} \cup T^*_{bp}\right) = \left(\stackrel{\tau}{\rightarrow}\right)^* \cup \left\{(p, a, q) \,|a \neq \tau \wedge (\exists p', q' \in A)\, p \left(\stackrel{\tau}{\rightarrow}\right)^* p' \stackrel{a}{\rightarrow} q' \left(\stackrel{\tau}{\rightarrow}\right)^* q\right\}$

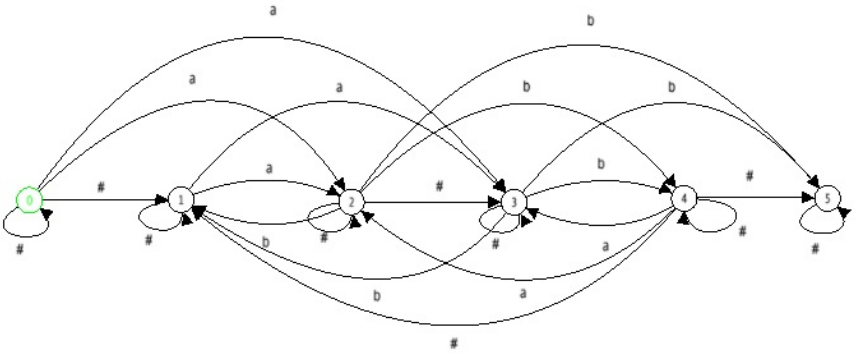

Fig. 5. Example saturated LTS

5 Alternating Bit Protocol - Modelling, Specification and Verification

The alternating bit protocol is a simple yet effective protocol (usually used as a test case), designed to ensure reliable communication through unreliable transmission mediums, and it is used for managing the retransmission of lost messages [AILS07a][Kul94].

The representation of Alternating Bit Protocol (hereby abbreviated as ABP) is shown bellow, and it consists of a *Sender* S, a *Receiver* R and two channels *Transport* T and *Acknowledge* A.

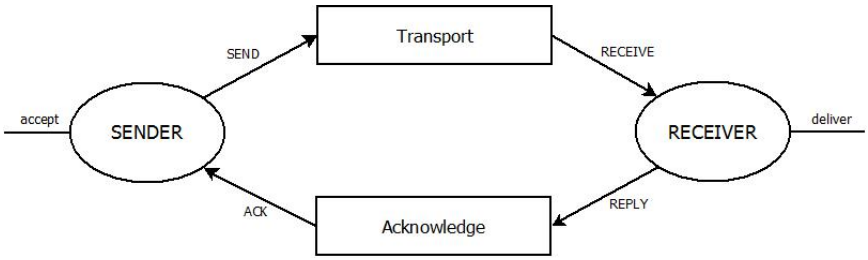

Fig. 6. Alternating bit protocol

All of the transitions in the ABP are internal synchronization and the only visible transitions are deliver and accept, which can occur only sequentially.

The specication of ABP is that it should act as a simple buffer, as follows: $ABP = \overline{deliver}.accept.ABP$

Messages are sent from a sender S to a receiver R. Channel from S to R is initialized and there are no messages in transit. There is no direct communication between the sender S and the receiver R, and all messages must travel trough the medium (transport and acknowledge channel). The ABP works like this:

1. Each message sent by S contains the protocol bit, 0 or 1.
   Here is the implementation:
   $Imp \stackrel{def}{=} (S|T|R|A) \setminus L$,
   where $L = (send0, send1, receive0, receive1, reply0, reply1, ack0, ack1)$ is the set of restricted actions.
2. When a sender S sends a message, it sends it repeatedly (with its corresponding bit) until receiving an acknowledgment ($ack0$ or $ack1$) from a receiver R that contains the same protocol bit as the message being sent.
   $S = \overline{send0}.S + ack0.accept.S_1 + ack1.S$
   $S_1 = \overline{send1}.S_1 + ack1.accept.S + ack0.S_1$
   The transport channel transmits the message to the receiver, but it may lose the message (lossy channel) or transmit it several times (chatty channel).
   $T = send0.\,(T + T_1) + send1.\,(T + T_2)$

$T_1 = \overline{receive0}.\,(T + T_1)$
$T_2 = \overline{receive1}.\,(T + T_2)$

3. When R receives a message, it sends a reply to S that includes the protocol bit of the message received. When a message is received for the first time, the receiver delivers it for processing, while subsequent messages with the same bit are simply acknowledged. $R = receive0.\overline{deliver}.R_1 + \overline{reply1}.R + receive1.R$
$R_1 = receive1.\overline{deliver}.R + \overline{reply0}.R_1 + receive0.R_1$
Again the acknowledgement channel sends the ack to sender, and it also can acknowledge it several times or lose it on the way to the sender.
$A = reply0.\,(A + A_1) + reply1.\,(A + A_2)$
$A_1 = \overline{ack0}.\,(A + A_1)$
$A_2 = \overline{ack1}.\,(A + A_2)$
4. When S receives an acknowledgment containing the same bit as the message it is currently transmitting, it stops transmitting that message, flips the protocol bit, and repeats the protocol for the next message.[Kul94][Kul94]

In order to verify the alternating bit protocol, it needs to be proven that the implementation meets its specification, or more precisely it needs to be shown that $ABP \approx Imp$. Such a bisimulation R can be found as follows:

ABP implementation states	ABP specification states
$(S\|T\|R\|A)\setminus L$, $(S\|\,(T+T1)\,\|R\|A)\setminus L$, $(S\|\,(T+T1)\,\|R\|\,(A+A2))\setminus L$, $(S\|T\|R\|\,(A+A2))\setminus L$, $\big(S\|\,(T+T1)\,\|\overline{deliver}.R1\|A\big)\setminus L$, $\big(S\|\,(T+T1)\,\|\overline{deliver}.R1\|\,(A+A2)\big)\setminus L$, $(S1\|\,(T+T1)\,\|R1\|\,(A+A1))\setminus L$, $\big(S1\|\,(T+T2)\,\|\overline{deliver}.R\|\,(A+A1)\big)\setminus L$, $(S1\|\,(T+T2)\,\|R1\|\,(A+A1))\setminus L$, $(S\|\,(T+T2)\,\|R\|\,(A+A2))\setminus L$	ABP
$(accept.S1\|\,(T+T1)\,\|R1\|\,(A+A1))\setminus L$, $(S\|\,(T+T1)\,\|R1\|\,(A+A1))\setminus L$, $(S\|\,(T+T1)\,\|R1\|\,(A+A2))\setminus L$, $(S\|\,(T+T1)\,\|R1\|A)\setminus L$, $(S1\|\,(T+T2)\,\|R\|\,(A+A2))\setminus L$, $(accept.S\|\,(T+T2)\,\|R\|\,(A+A2))\setminus L$, $(S1\|\,(T+T2)\,\|R\|\,(A+A1))\setminus L$	ABP'

Table 2. Verification of Alternating Bit Protocol

5.1 Alternating Bit Protocol - Tool Usage Example

In this section we present how to use the tool to show that the specification and implementation of the ABP are weakly bisimiliar. As a first step we always need to generate the LTSs from both the specification and implementation of the system that we are trying to model. That can be done in the tab "CCS to LTS" in the tool. As shown in 7 in the upper text area we need to write down the ccs expressions that describe the system or load them from a file. One constraint here is that a general ccs expression that describes the whole system has to be put on the first line. The algorithm for generating the LTS graph starts evaulation and construction of the graph starting from the first line. When we have our ccs expressions put in place we have to click "Generate LTS". This action will make the application parse the expression and if no errors are found will start generating the LTS graph The results from the generated graph will be presented in the lower text area in Aldebaran format. The users can save the results in a file or can click "View LTS Graph" which causes the tool to display a computer generated image of the LTS graph.

LTS minimization can be done in the "LTS minimization" tab. The minimization is pretty straigthforward process. As shown in 8 and 9 LTS is loaded from file, a method of calculation has to be chosen and in order to perform calculation the user has to click the button "Calculate".

Checking for LTS bisimilarity is performed in the "LTS comparison" tab. This tab is similary to the one for the minimization. As shown in 10 and 11 two LTSs have to loaded from file. In order to check for LTS bisimilarity the user has to chose a method for calculatino and click on the button "Calculate". The results will be displayed on the right side of the button and they give information wheather the LTSs are bisimilar and how long the calculation lasted.

6 Peterson and Hamilton algorithm - Modeling, Specification and Testing

Peterson's algorithm (aka Peterson's solution) is a concurrent programming algorithm for mutual exclusion. Mutual exclusion (often abbreviated to mutex) algorithms are used in concurrent programming to avoid the simultaneous use of a common resource, such as a global variable, by pieces of computer code called critical sections. A critical section is a piece of code in which a process or thread accesses a common resource. The critical section by itself is not a mechanism or algorithm for mutual exclusion. A program, process, or thread can have the critical section in it without any mechanism or algorithm which implements mutual exclusion. Assume that a variable (memory location) can only have one value (never between values) and processes A and B write want to write to the same memory location at the "same time", either the value from A or the value from B will be written rather than some scrambling of bits. Peterson and Hamilton's algorithm is a simple algorithm that can be run by two processes to ensure mutual exclusion for one resource (say one variable or data structure). Shared

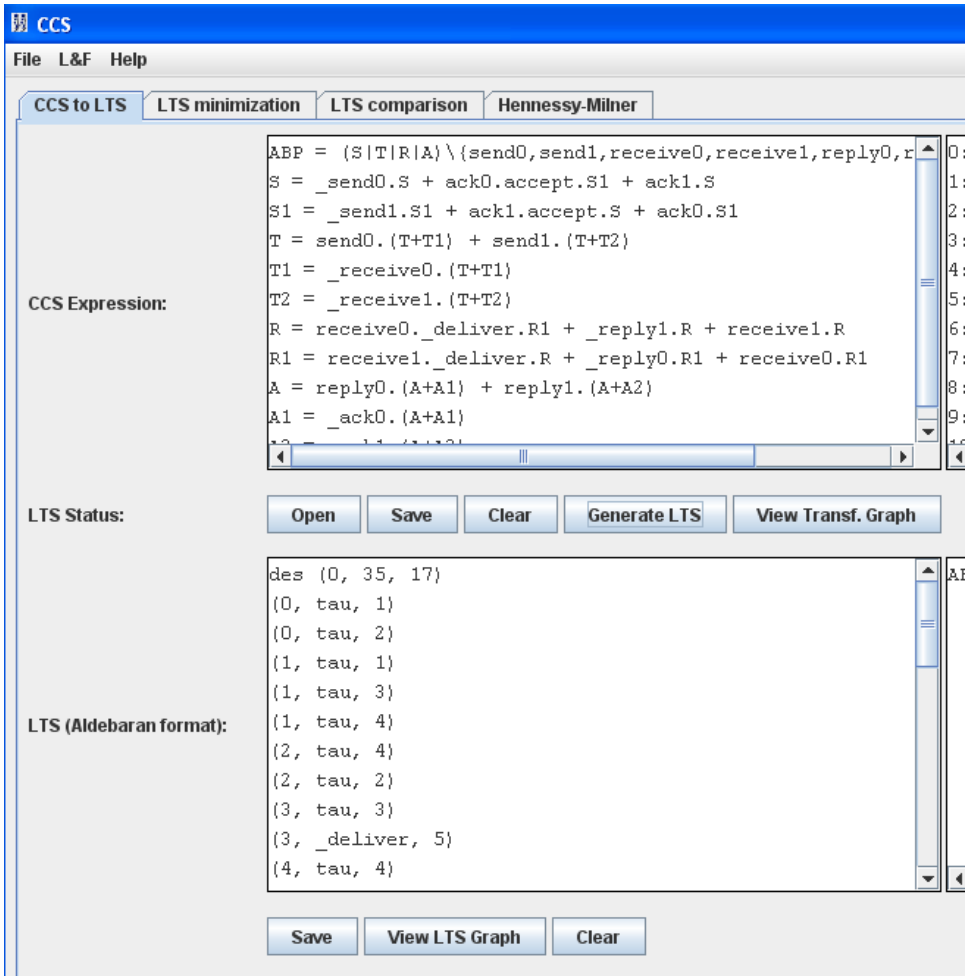

Fig. 7. Tool Usage: ABP Modeling and Generating LTS

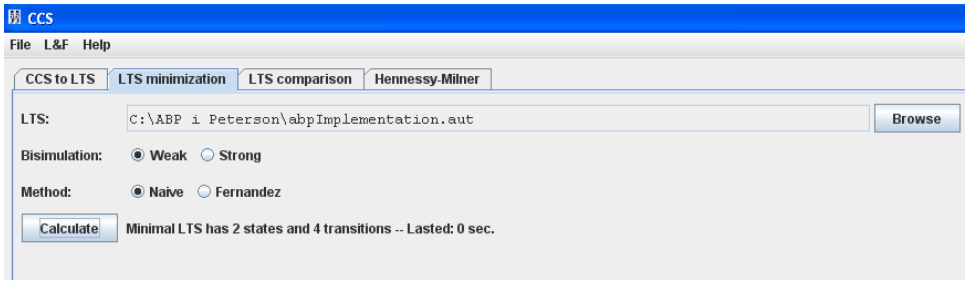

Fig. 8. Tool Usage: ABP Minimization, Weak, Naive

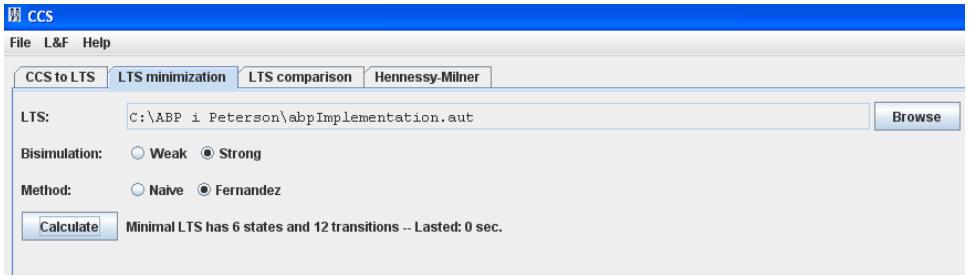

Fig. 9. Tool Usage: ABP Minimization, Strong, Fernandez

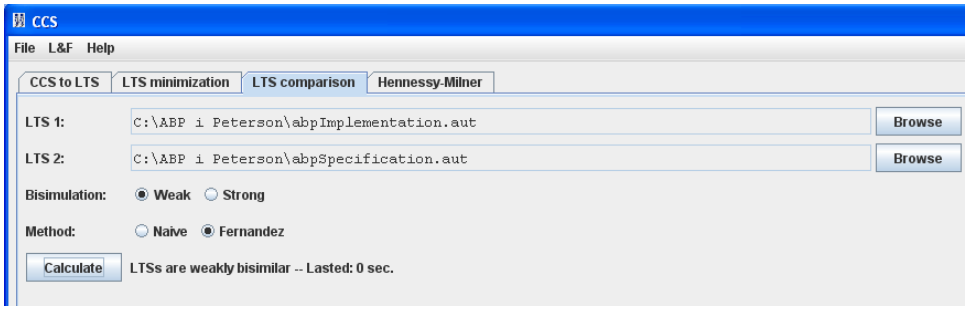

Fig. 10. Tool Usage: ABP Bisimilarity, Weak, Fernandez

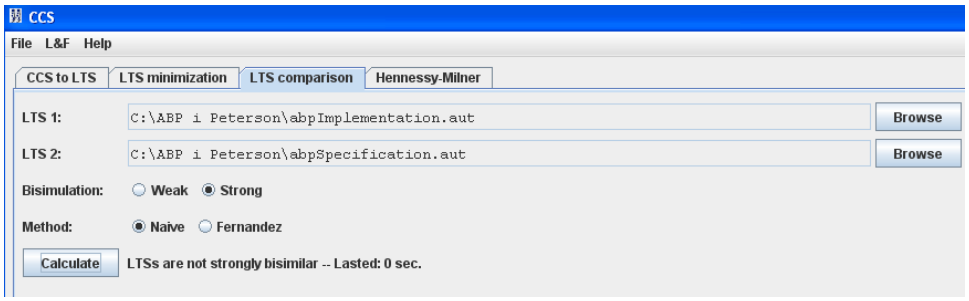

Fig. 11. Tool Usage: ABP Bisimilarity, String, Naive

variables are created and initialized before either process starts. The shared variables flag[0] and flag[1] are initialized to FALSE because neither process is yet interested in the critical section. The shared variable turn is set to either 0 or 1 randomly (or it can always be set to say 0). The next figure shows the algorithm.

There are two different processes which want to enter at critical section. This means that the processes are fighting for one resource (ex. Say one variable or data structure). flag[i]= true means that process I wants to enter the critical section and the turn=i means that process i is next to enter the critical section. At first the shared variables flag[0] and flag[1] are initialized to false because

```
var flag: array [0..1] of boolean;
turn: 0..1;
//flag[k] means that process[k] is interested in the critical section
flag[0] := FALSE;
flag[1] := FALSE;
turn := random(0..1)
//After initialization, each process, which is called process i in the code, runs this
code:
repeat
 flag[i] := TRUE;
 turn := j;
 while (flag[j] and turn=j) do no-op;
 //CRITICAL SECTION
 flag[i] := FALSE;
 //REMAINDER SECTION
until FALSE;
```

Fig. 12. Peterson-Hamilton's algorithm

neither process is yet interested in the critical section. The shared variable turn is set to either 0 or 1 randomly (or it can always be set to say 0). If the process cant enter the critical section it waits in while loop.
Legend: turn 0, 1 flag0, flag1 true, false = t, f w write r read enter enter the critical section exit exit the critical section

Initialization:
turn=1
flag1 = flag2 = f

CCS Specification:
Peterson = (P1 — P2 — flag1f — flag2f — turn1) L
P1 = flag1wt.turnw2.P11
P11 = flag2rf.P12 + flag2rt.(turnr2. .P11 + turnr1.P12)
P12 = enter1.exit1.flag1wf.P1

P2 = flag2wt.turnw1.P21
P21 = flag1rf.P22 + flag1rt.(turnr1. .P21 + turnr2.P22)
P22 = enter2.exit2.flag2wf.P2

FLAG1f = flag1rf.flag1f + flag1wf.flag1f + flag1wt.flag1t
FLAG1t = flag1rt.flag1t + flag1wt.flag1t + flag1wf.flag1f

FLAG2f = flag2rf.flag2f + flag2wf.flag2f + flag2wt.flag2t
FLAG2t = flag2rt.flag2t + flag2wt.flag2t + flag2wf.flag2f
TURN1 = turnr1.turn1 + turnw1.turn1 + turnw2.turn2

TURN2 = turnr2.turn2 + turnw2.turn2 + turnw1.turn1

L = flag1wt, flag1rt, turnw2, union of the access sorts (r,w) of the variables

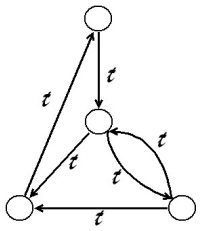

Fig. 13. Live cycle of one process

The previous figures were shown pictures of life cycle of a one process and Model of Peterson - Hamilton algorithm.

The mutual exclusion requirement is assured. Suppose instead that both processes are in their critical section.

Only one can have the turn, so the other must have reached the while test before the process with the turn set its flag. But after setting its flag, the other process had to give away the turn. Contradicting our assumption, the process at the while test has already changed the turn and will not change it again. The progress requirement is assured.

Again, the turn variable is only considered when both processes are using, or trying to use, the resource. Deadlock is not possible. One of the processes must have the turn if both processes are testing the while condition. That process will

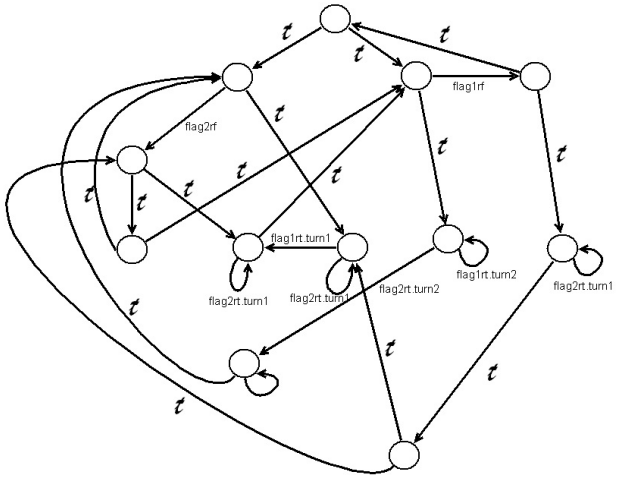

Fig. 14. Model of Peterson-Hamilton algorithm

proceed. Finally, bounded waiting is assured. When a process that has exited the CS reenters, it will give away the turn. If the other process is already waiting, it will be the next to proceed.

Test and set algorithm
repeat
while Test-and-Set(lock) do no-op;
critical section
lock := false;
remainder section
until false

Test-and-Set(target)
result := target;
target := true;
return result

Process 1	Process 2
Wants to set target to true	
Target is changed to true	
Result comes back false so no busy waiting	
	Wants to set target to true
	It receives the result true
	Busy waits as long as Process 1 is in critical section
Leaves critical section	
Set target to false	

7 U^w and U^s implementation

The U^w and U^s operators are added to Hennessy-Milner logic in order to enable and describe the recursive definitions. The addition of the recursive definitions increases the power for specifying the features of the processes of the language.
F U^s G $\stackrel{min}{=}$ G $\vee$ (F $\wedge$ $\langle$Act$\rangle$tt $\wedge$ [Act](F U^s G))
F U^w G $\stackrel{max}{=}$ G $\vee$ (F $\wedge$ [Act](F U^w G)).

This part explains how implementation of Hennessy-Milner logic works. For a given Hennessy-Milner expression and a graph, we should get an output whether that expression is valid for that graph. One Hennessy-Milner expression can be made of these set of tokens {"AND", "OR", "UW", "US", "NOT", "[", "]", "$\langle$", "$\rangle$", "TT", "FF", "(", ")", "", "", ","}. The first part of this process is called tokenization. Tokenization is the process of demarcating and possibly classifying sections of a string of input characters. The resulting tokens are then passed on to some other form of processing. The process can be considered as a sub-task of parsing input. As the tokens are being read they are proceeded to the parser. The parser is a LR top-down parser. This parser is able to recognize a non-deterministic grammar which is essential for the evaluation of the Hennessy-Milner expression. As the parser is reading the tokens, thus in a way we move through the graph and determine if some of the following steps are possible.

Every condition is in a way kept on stack and that enables later return to any of the previous conditions. The process is finished when the expression is processed or when it is in a condition from which none of the following actions can be taken over.

The implementing of these operators is made in this way: we get the current state. Current state is the state which contains all the nodes that satisfied the previous actions. For example if we have expression ⟨a⟩ ⟨a⟩TT U^s ⟨b⟩TT, than we get all the nodes that we can reach, starting from the start node, with two actions a. When we get to the Until operation, in this case U^s, we take the start state and check if we can make an action b from some nod. If we cant do action b, than we repeat the process, do a again and check if can we do b. This process is being repeated until we come to a state in which we can't do action a from all nodes in that state. In the end we check the operators type (strong or weak). If the operator is strong, than only one node in its state is enough. If it is weak, we check whether in its new state there arent any nodes or if from the starting node we can get to some of its neighbors through action b.

BNF

HML =⟩ TT | FF | HML UW HML | HML US HML | HML AND HML | HML OR HML | ⟨a⟩HML | [a]HML | (HML) | NOT HML

Grammar

1. HML =⟩ Term Expression
2. Expression =⟩ AND Term Expression
3. Expression =⟩ OR Term Expression
4. Expression =⟩ Uw Term Expression
5. Expression =⟩ Us Term Expression
6. Expression =⟩ λ
7. Term =⟩ NOT Term
8. Term =⟩ [Actions] Term
9. Term =⟩ ⟨ Actions ⟩ Term
10. Term =⟩ TT
11. Term =⟩ FF
12. Term =⟩ (HML)
13. Actions =⟩ Action
14. Actions =⟩ Action ActionsList
15. ActionsList =⟩ , Action ActionsList
16. ActionsList =⟩ λ
17. Action =⟩ Literal Name
18. Name =⟩ Literal
19. Name =⟩ Number
20. Name =⟩ λ
21. Literal =⟩ a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z
22. Number =⟩ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

8 Conclusion

The conclusion goes here.

Appendix A: Comparison and Analysis of the bisimulation equivalence algorithms

There is no official set of benchmarks for testing algorithms for computing bisimulation equivalence [DPP04]. And it is also very difficult to randomly generate suitable labeled transition systems. Therefore, we used the academical examples from mCRL2 as experimental test models.

The experiments were conducted on a laptop with the following specifications:

1. CPU: Intel Pentium P6100 2.0 GHz
2. Memory: 3 GB
3. ??? sth else???

Each of the experiments carried out on our Java implementations of the two algorithms for computing bisimulation equivalence consisted of 10 repeated runs of each of the algorithms on each of the aldebaran test files and measuring the average running time in miliseconds. The results from these experiments are presented in Table 3. The table includes the running times in miliseconds and the absolute error in miliseconds for both of the algorithms, the number of pairs of bisimilar states obtained with the naive algorithm (excluding the reflexive and symmetric pairs) and the number of classes of bisimulation equivalence obtained with the algorithm due to Fernandez (excluding the one-element classes), as well as the ratio of the running time of the naive algorithm with respect to the running time of Fernandez's algorithm.

			Naive		Fernandez				Ratio
aut	nr.states	nr.transitions	$t(ms)$	$\Delta t(ms)$	$t(ms)$	$\Delta t(ms)$	nr.bisimilar pairs	nr.bisimilar classes	$tntf$
scheduler	13	19	17.4	1.18	4.4	0.96	1	1	3.95
trains	32	52	64.5	7.3	7.6	1.76	6	6	8.48
mpsu	52	150	215.4	1.16	27.5	0.5	4	4	7.83
par	94	121	5909.4	22.68	28.5	0.7	170	27	207.34
abp	74	92	162.2	1.44	69.6	3.92	6	6	2.33
abpbw	97	122	287.9	0.94	173.3	5.22	32	27	1.661
leader	392	1128	/	/	841.5	41.5	/	20	/
tree	1025	1024	/	/	1858.1	55.16	/	8	/
dining3	93	431	16.8	0.8	250	2	1	2	0.067
cabp	672	2352	/	/	16184.3	198.94	/	90	/

Table 3. Results of the comperisons

The results from the experiments carried out are presented graphically in Fig. 15 and Fig. 16 for the naive algorithm and the algorithm of Fernandez, respectively.

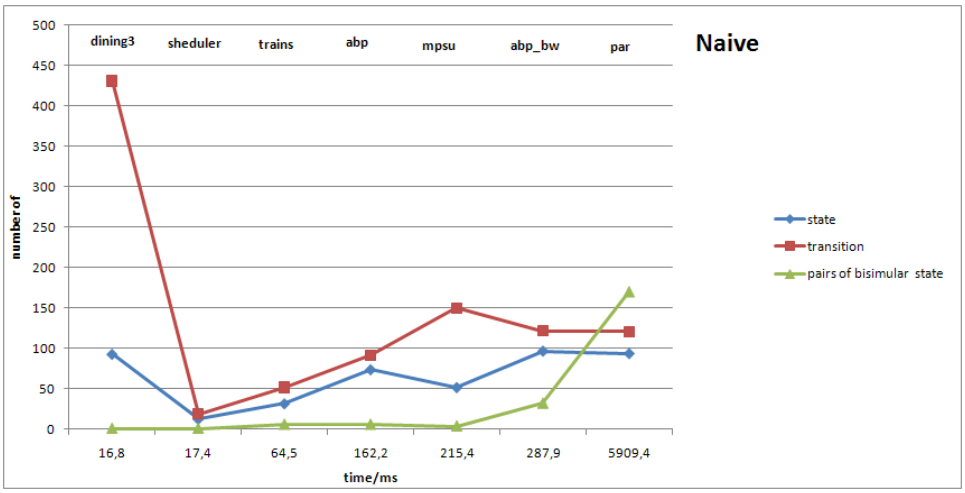

Fig. 15. Analysis of the running time of the naive bisimulation algorithm

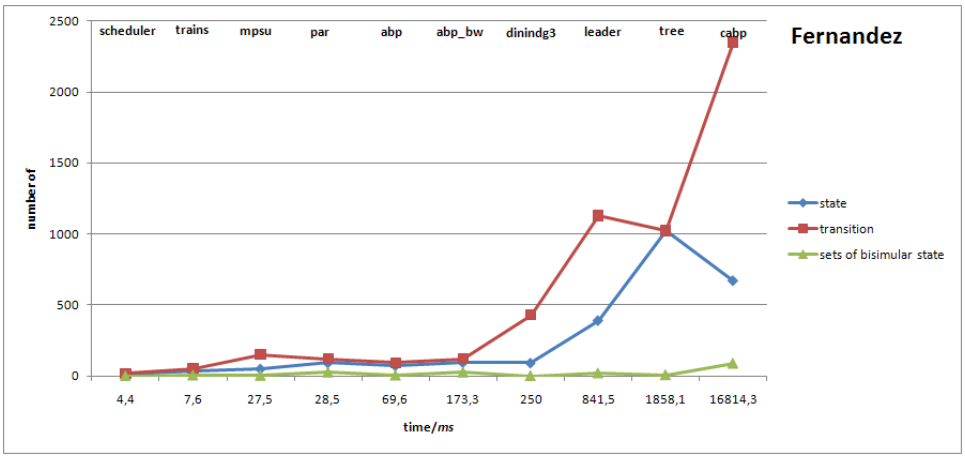

Fig. 16. Analysis of the running time of the bisimulation algorithm due to Fernandez

It can be seen that for all test models, with the exception of the first one, the running time of the naive algorithm is proportional to the number of states, the

number of transitions and the number of resulting bisimilar pairs. However, we need to note that this is not a general conclusion and that in general the running time of the naive algorithm for computing bisimulation equivalence depends on the nature of the monotonic function used by this algorithm, which is strongly related with the form of the labeled transition system. As an example, dining3 is a labeled transition system that has big number of transitions, however in this test model the algorithm determines that the monotony condition is not fulfilled in the second development of the monotonic function and therefore the algorithm stops there.

Similar conclusions can be drawn for the algorithm due to Fernandez. Its running time is also proportional to the number of transitions, number of states and the number of bisimilar classes.

The comparison of running times of the two algorithms obtained with the experiment is shown in Fig. 17. As it can be clearly seen the algorithm of Fernandez is few times faster than the naive algorithm. The biggest difference can be noticed for the example par.aut where the ratio of the running times is 207.34. An exception is only the example dining3.aut for which the naive algorithm performs faster due to the reasons described earlier. The ration of the running times in this case is 0.0067. Fig. 18 shows the ratios of the running times of the two algorithms for each of the examples.

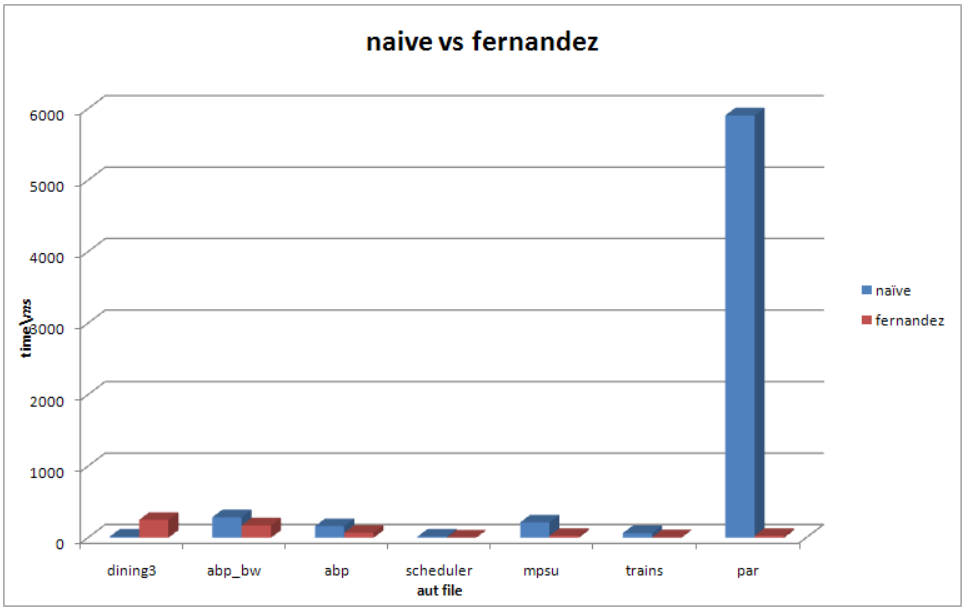

Fig. 17. Comparison of the running time of the two bisimulation algorithms

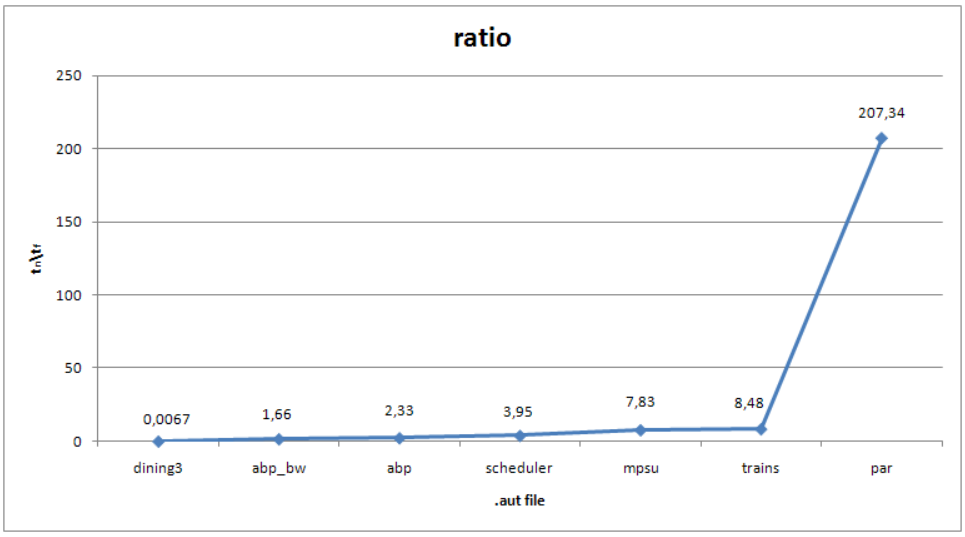

Fig. 18. Ratio of the running times of the two bisimulation algorithm

References

AILS07. Luca Aceto, Anna Ingolfsdottir, Kim G. Larsen and Jiri Srba, *Reactive Systems - Modeling, Specification and Verification*, Cambridge University Press, 2007

Par07. Terence Parr, *The Definitive ANTLR Reference - Building Domain-Specific Languages,* The Pragmatic Bookshelf, 2007

NW. Niklaus Wirth
`http://en.wikipedia.org/wiki/Niklaus_Wirth`

BK08. Christel Baier, Joost-Pieter Katoen, *Principles of model checking*, The MIT Press, pages 456-463, 2008

AILS07a. Luca Aceto, Anna Ingolfsdottir, Kim G. Larsen and Jiri Srba, *Reactive Systems - Modeling, Specification and Verification*, Cambridge University Press, pages 85-88, 2007

AILS07b. Luca Aceto, Anna Ingolfsdottir, Kim G. Larsen and Jiri Srba, *Reactive Systems - Modeling, Specification and Verification*, Cambridge University Press, pages 78-85, 2007

Fer89. J.-C. Fernandez, *An Implementation of an Efficient Algorithm for Bisimulation Equivalence*, Science of Computer Programming, vol. 13, pages 219-236, 1989/1990

PT87. R. Paige and R. Tarjan, *Three partition refinement algorithms,* SIAM J. Comput. 16 (6), 1987

GR09. J.F. Groote and M. Reniers, *Modelling and Analysis of Communicating Systems,* Technical University of Eindhoven, rev. 1478, pages 15-18, 2009

DPP04. A. Dovier, C. Piazza and A. Policriti, *An efficient algorithm for computing bisimulation equivalence,* Theoretical Computer Science 311, pages 221-256, 2004

BPS01a. J.A. Bergstra, A. Ponse and S.A. Smolka, *Handbook of process algebra,* Elsevier Science B.V., pages 9-13, 2001

BPS01b. J.A. Bergstra, A. Ponse and S.A. Smolka, *Handbook of process algebra,* Elsevier Science B.V., pages 47-55, 2001

AILS07a. Luca Aceto, Anna Ingolfsdottir, Kim G. Larsen and Jiri Srba, *Reactive Systems - Modeling, Specification and Verification*, Cambridge University Press, pages 279-280, 2007

Kul94. Seth Kulick, *Process Algebra, CCS, and Bisimulation Decidability*, University of Pennsylvania, pages 8-10, 1994

Kul94. M.Alexander and W.Gardner, *Process Algebra for Parallel and Distributed Processing*, Chapman&Hall/CRC Press, Computational Science Series, pages 114-118, 2009

mCRL2Ref. mCRL2
`http://www.mcrl2.org`