A tool for modeling, manipulation and analysis of concurrent systems based on CCS process language and LTS as its semantic model (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. This paper is a technical report of an effort to build a tool for automatic specification and analysis of concurrent systems. The tool we present here has implemented the CCS process language and can build LTSs (Labelled Transition Systems) from CCS expressions. Furthermore, it can be used to reduce the state space of an LTS and to check whether two LTSs exhibit the same behaviour, using strong and weak bisimulation equivalence. It can parse Henessy-Milner logic formulas as well and also implements the U^w and U^s operators that enable recursive definitions. The tool is very simple, but it has the functionality needed to perform modeling, specification and verification. We give an illustration of that process for two classical examples in the concurrency theory: the Alternating Bit Protocol and Peterson's mutual exclusion algorithm.

Keywords: CCS, LTS, Bisimulation equivalence, LTS minimization, LTS comparation, Formal verification, Henessy-Milner logic.

1 Introduction

Due to the increasing complexity of the real-life systems, manual analysis is very hard and often impossible. Therefore, having computer support for modeling, analysing and verifying system behaviour is of an essential importance nowdays. This means that the systems properties and their behaviour need to be described by formal methods with an explicit formal semantics.

* This technical report is a result of a group project work for the course in Formal methods at the MSc studies in Computer Sciences at the Institute of Informatics 2010/2011

One of the basic process languages used to formally describe and analyse any collection of interacting processes is the Calculus of Communication Systems (CCS), process algebra introduced by Robert Milner and based on the message-passing paradigm. As a semantic model of CCS the notion of Labelled Transition System (LTS) is used. CCS, like any process algebra, can be exploited to describe both implementations of processes and specifications of their expected behaviours.

One of the most important behavioral equivalence is the strong bisimulation equivalence (strong bisimilarity) because if two processes and bisimilar, they cannot be distinguished by any realistic form of behavioral observation. Another useful behavioral equivalence is the weak bisimulation equivalence (weak bisimilarity) which relates processes with internal (non-observable) actions. These and other behavioral equivalences can be used to reduce the size of the state space of an LTS and to check the equivalence between two LTSs. This finds an extensive application in model checking and formal verification.

This paper presents a simple tool that can be used to automate the process of modeling, manipulation and analysis of concurrent systems described by the CCS process language. The tool is given as a Java executable jar library with very simple Graphical User Interface (GUI). Even though it is simple, the tool is functional enough to be able to satisfy the basic requirements for modeling, specification and verification. It can parse CCS expressions, generate LTS from CCS in Aldebaran format, reduce an LTS to its canonical form modulo strong or weak bisimilarity and check whether two LTSs are strongly or weakly bisimilar. Additionally, it can parse HML formulas and implements the HML operators for recursive definitions as well.

1.1 Related work

Different tools for modeling, specification and verification of concurrent reactive systems have been developed over the past two decades. Probably the most famous and most commonly used ones, especially in the academic environment, are the Edinburgh Concurrency Workbench (CWB) and micro Common Representation Language 2 (mCRL2).

The Edinburgh Concurrency Workbench (CWB) is a tool for analysis of cuncurrent systems. CWB allows for equivalence, preorder and model checking using a variety of different process semantics. It also allows defining of behaviours in an extended version of CCS and perform various analysis of these behaviours, such as checking various semantic equivalences for examples checking if two processes (agents in CWB) are strongly or weakly bisimilar [CWB]. We have used the CWB many times while testing our tool, for checking CCS validity and also testing algorithms for strong and weak bisimularity of LTS graphs. Although CWB covers much of the functionality of our tool and more, CWB has a command interpreter interface that is more difficult to work with, unlike our tool that has a Graphical User Interface (GUI), and is very intuitive. As far as we know the CWB does not have the functionality for exporting LTS graphs in Aldebaran format, that our tool has.

The micro Common Representation Language 2 (mCRL2), successor of CRL, is a formal specification language that can be used to specify and analyse the behaviour of distrubuted systems and protocols. Its accompanying toolset contains extensive collection of tools to automatically translate any mCRL2 specification to a linear process, manipulate and simulate linear processes, generate the state space associated with a linear process, manipulate and visualize state spaces, generate a PBES from a formula and a linear process, generate a BES from PBES and manipulate and solve (P)BESs [mCRL2]. All mCRL2 tools can be used from the command line, but mCRL2 has an enhanced Graphical User Interface (GUI) as well which makes it very user-friendly and easy to use. We have used mCRL2 many times while testing our tool, for testing the algorithms for minimization and comparison of LTS graphs using strong and weak bisimularity. However, to our knowledge, mCRL2 does not provide the possibility to define systems' behaviour in the CCS process language nor to specify systems' properties using HML logic, functionalities that are implemented in the very initial version of our tool. Also, even though mCRL2 supports minimization modulo strong and weak bisimulation equivalence, it does not output the computed bisimulation, feature that we have implemented in our tool.

1.2 Outline

The contributions of this paper are organized as follows. In the next section we give introduction to some of the basic terminology used throughout the report. The third section is devoted to the implementation of the CCS process language and generation of Labelled Transition Systems as a semantic model of process expressions, and it also discusses some of the choices made during the implementation. Section four describes the process of reducing the size of the state space of an LTS as well as checking the equivalence between two LTSs with respect to behavioural relations such as strong bisimilarity and observational equivalence (weak bisimilarity). It includes implementation details of two algorithms for computing strong bisimulation equivalence, the naive algorithm and the advanced algorithm due to Fernandez, as well as description of the saturation technique together with a respective algorithm for saturating an LTS with which the problem of computing weak bisimilarity is reduced to computing strong bisimilarity over the saturated systems. Section five explains how the implementation of Hennesy-Milner Logic (HML) works and gives implementation details for the U^w and U^s operators. Next we illustrate the application of the tool for modeling, specification and verification of some classical examples. These classical examples include the Alternating Bit Protocol and Peterson's mutual exclusion algorithm. Finally, we give some conclusions and some directions for future development of the tool. We also include four appendixes. The first appendix contains the experimental results we got with analysis of the running times of our implementations of the two algorithms for computing bisimulation equivalence. Appendix two shows the syntax diagram for the grammar that recognizes CCS expressions, while Appendix three presents the grammar for HML

expressions. The last appendix includes some visual illustration of the tool's usage via screenshots of the graphical application.

2 Preliminaries

In this section, we give some definitions of the basic terminology used throughout the report.

Definition 1. *(Calculus of Communicating Systems): Calculus of Communicating Systems is algebraic theory to formalize the notion of concurrent computation. Commonly known as CCS.*

Definition 2. *(Labelled Transition System): A labelled transition system (LTS) is a five tuple* $A = (S, Act, \rightarrow, s, T)$ *where*

- S *is set of states,*
- Act *is a set of actions, possibly multi-actions,*
- $\rightarrow \subseteq S \times Act \times S$ *is a transitionrelation,*
- $s \in S$ *is the initialstate,*
- $T \subseteq S$ *is the set of terminatingstates.*

Definition 3. *(Strong bisimulation): Let* $A_1 = (S_1, Act, \rightarrow_1, s_1, T_1)$ *and* $A_2 = (S_2, Act, \rightarrow_2, s_2, T_2)$ *be labelled transition systems. A binary relation* $R \subseteq S_1 \times S_2$ *is called a strong bisimulation relation iff for all* $s \in S_1$ *and* $t \in S_2$ *such that* sRt *holds, it also holds for all actions* $a \in Act$ *that:*

1. *if* $s \xrightarrow{a}_1 s'$ *then there is a* $t' \in S_2$ *such that* $t \xrightarrow{a}_2 t'$ *with* $s'Rt'$,
2. *if* $t \xrightarrow{a}_2 t'$ *then there is a* $s' \in S_1$ *such that* $s \xrightarrow{a}_1 s'$ *with* $s'Rt'$, *and*
3. $s \in T_1$ *if and only in* $t \in T_2$.

Definition 4. *Let* P *and* Q *be CCS processes or, more generally, states in a labeled transition system. For each action* a, $P \stackrel{a}{\Rightarrow} Q$ *iff:*

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

where $\left(\xrightarrow{\tau}\right)^*$ *denotes the reflexive and transitive closure of the relation* $\xrightarrow{\tau}$.

Definition 5. *(Weak bisimulation / Observational equivalence): A binary relation* R *over the set of states of an LTS is a weak bisimulation (observational equivalence) 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.

Definition 6. *(Saturation): Let $T \subseteq S \times Act \times S$ be an LTS. Then*

$$T^* = \left\{ (p,a,q) \,|p \stackrel{a}{\Rightarrow} q \right\} =$$
$$= T \cup \left(\stackrel{a}{\rightarrow} \right)^* \cup \left\{ (p,a,q) \,|a \neq \tau \wedge (\exists p', q' \in A)\, p \left(\stackrel{\tau}{\rightarrow} \right)^* p' \stackrel{a}{\rightarrow} q' \left(\stackrel{\tau}{\rightarrow} \right)^* q \right\}$$

is a saturation of T.
Two LTSs are weakly bisimilar iff their saturated LTSs are strongly bisimilar.

Definition 7. *(Hennessy-Milner): The set of M Hennessy-Milner formulas over a set of actions Act is given by the following abstract syntax:*

$$F, G ::= tt | ff | F \wedge G | F \vee G | \langle a \rangle F | [a] F$$

where $a \in Act$ and tt and ff denote true and false, respectively. If $A = \{a_1, ..., a_n\} \subseteq Act\, (n \geq 0)$, we use use the abbreviation $\langle A \rangle F$ for the formula $\langle a_1 \rangle F \vee ... \vee \langle a_n \rangle F$ and $[a] F$ for the formula $[a_1] F \wedge ... \wedge [a_n] F$. (If $A = \emptyset$ then $\langle a \rangle F = ff$ and $[a] = tt$.)

3 CCS 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.

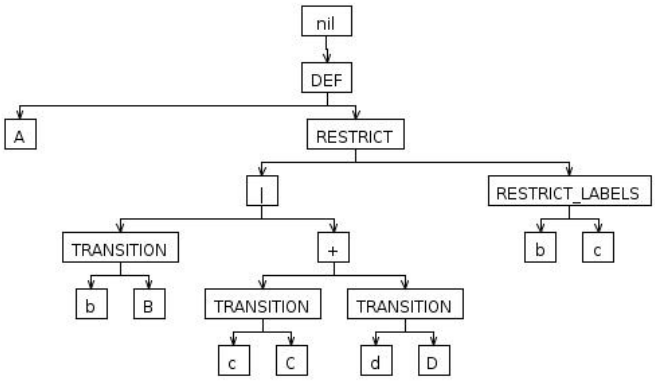

Fig. 1. AST Example

3.1 Generating the parser

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 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.

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:

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

3.2 CCS domain model and LTS graph generation

Although working directly with ASTs and performing all algorithms on them is possible, it causes a limitation in future changes, where even a small change in the grammar and/or in the structure of the generated ASTs causes a change in the implemented algorithms. Because of this, a specific domain model is built along with a domain builder algorithm, which has corresponding abstractions for all CCS operators, processes and actions. The input of the domain builder algorithm is an AST, and the output is a fully built domain model. The algorithms for constructing an LTS graph are implemented on the domain model because it is not expected for the domain model structure to change much in the future. The domain model is also a tree-like structure, so it is as easy to work with, as with the AST.

The LTS generation algorithm is a recursive algorithm which traverses the tree structure of objects in the domain model and performs SOS rule every time it reaches an operation. In this fashion all SOS transformation are performed on the domain and as result a new graph structure is created which represents the LTS which can be easily exported to a file in Aldebaran format.

3.3 Workflow of operations

In Fig. 2 the workflow of all operations that are executed for constructing an LTS graph from a CCS expression are shown. Every operation is done as a standalone algorithm independent from the other operations, that has input and output shown in the Figure. The modular design is deliberately chosen in order to help achieve better testing and maintenance of the source code.

4 LTS minimization and comparison

Bisimulation equivalence (bisimilarity) 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 with respect to behavioural relations. 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 and checking the equivalence of two labeled transition systems, using two behavioral equivalence relations: strong bisimilarity and observational equivalence (weak bisimilarity).

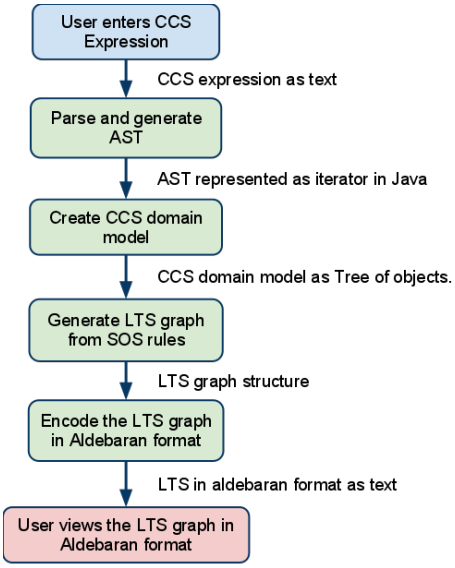

Fig. 2. Workflow of all operations for producing an LTS graph from a CCS expression

4.1 Minimization of an LTS modulo strong bisimilarity

The process of reducing the size of the state space of an LTS 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 afterwards serve as minimization procedures.

The naive algorithm for computing bisimulation equivalence stems from the theory underlying Tarski's fixed point theorem. It is based on the fact that the strong bisimulation equivalence is the largest fixed point of the monotic function F as defined in [AILS07] given by Tarsky's fixed point theorem. This algorithm has time complexity of $O(mn)$ for a labeled transition system with m transitions and n states.

Our implementation of the algorithm takes as an input an LTS in aldebaran format, and generates the corresponding labelled graph as a list of nodes representing the states which use the following data structure:

– $S_p = \{(a, q)\}$ - 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

The algorithm then computes the strong bisimulation equivalence and outputs it as pairs of bisimilar 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]. It has $O(m \log n)$ complexity for a labeled transition system with m transitions and n states.

Our implementation of Fernandez's algorithm takes an LTS in aldebaran format as an input, generates a labelled 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 states and transitions we used the following terminology represented by suitable data structures:

- $T_a[p] = \{q\}$ - 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

Having computed the strong bisimulation equivalence, the next step in the reduction of the state space of the LTS uses the bisimulation equivalence obtained 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 $k = p \cup q$;
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.

This process of reduction with respect to strong bisimilarity is illustrated below in Fig. **??**. The results obtained by applying both algorithms for computing strong bisimulation equivalence are given in Table 1. These results are then used as a basis for the reduction of the graph to its minimal form: all mutually bisimilar states are merged into a single state and their transitions are updated accordingly.

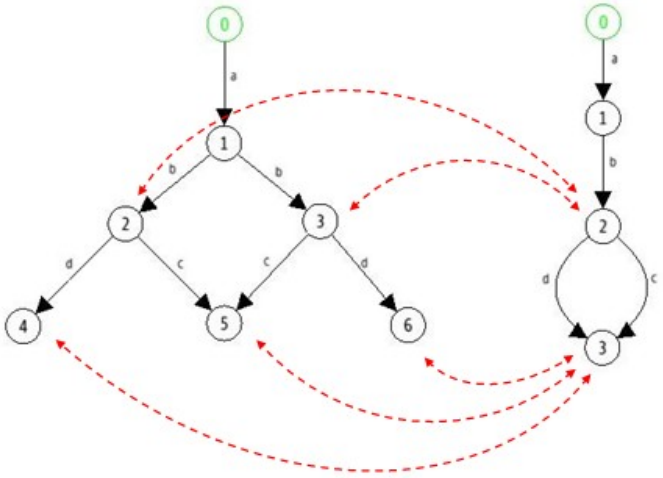

Fig. 3. Example LTS graph and its minimial form modulo strong bisimilarity. The red dashed arrows depict the mapping of the states using the computed bisimulation equivalence. States 2 and 3 are merged into state 2 in the minimal graph, and states 4,5 and 6 are merged into state 3 in the minimal graph.

Algorithm	LTS graph
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 the example LTS graph from Fig.3

4.2 Minimization of an LTS modulo weak bisimilarity

The minimization of an LTS modulo observational equivalence (weak bisimilarity) is reduced to the problem of minimization of an LTS modulo strong bisimilarity, using a technique called saturation. Intuitevely, saturation amounts to first precomputing the weak transition relation and then constructing a new pair of finite processes whose original transitions are replaced by the weak transitions [AILS07]. Once the algorithm for saturation is run and the LTS is saturated, the computation of weak bisimilarity amounts to computing strong bisimilarity of the saturated LTS.

The algorithm for saturation in our tool was implemented as follows: The set of triples R can be partitioned in $2n$ sets with:

$$T_{\tau p} = \{(p, \tau, q) \mid (p, \tau, q) \in T\}, \text{and}$$
$$T_{ap} = \{(p, \tau, q) \mid a \neq \tau \wedge (p, \tau, q) \in T\}$$

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 is constructed:

$$T^*_\tau = \bigcup_{p \in A} T^*_{\tau p} = \left\{(p, \tau, q) \,|p \left(\xrightarrow{\tau}\right)^* q\right\}$$

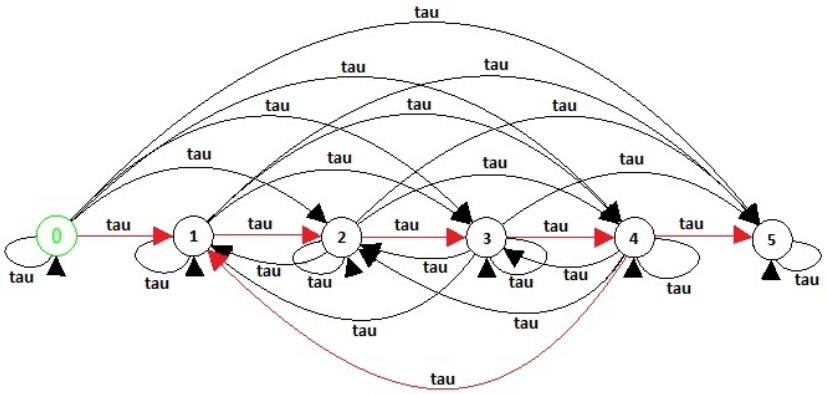

Fig. 4. Reflexive, transitive closure of τ. The original graph is depicted with red lines

The next step is to construct:

$$T'_{ap} = \bigcup_{p \in A} T'_{ap} = \left\{(p, a, q) \,|\, (\exists q' \in S)\,(p, a, q') \in T \wedge q' \left(\xrightarrow{\tau}\right)^* q\right\}: \qquad (3)$$

$T^0_{ap} = T_{ap'}$,
$T^i_{ap} = T^{i-1}_{ap} \cup \left\{(p, a, r) \,|\, (\exists q \in S)\,(p, a, q) \in T^{i-1}_{ap} \wedge (q, \tau, r) \in T^{i-1}_{\tau q}\right\}$ 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 $T'_{ap} = T^k_{ap} = T^{n|Act|}_{ap}$)

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:

$$T_{\tau q} = \{(p, \tau, q) \,|\, (p, \tau, q) \in T'\}, \text{and}$$
$$T_{bq} = \{(p, a, q) \,|\, a \neq \tau \wedge (p, a, q) \in T'\}$$

for every $p \in S$, and then construct:

$$T^0_{bq} = T_{bq},$$
$$T^i_{bq} = T^{i-1}_{bq} \cup \left\{(p', a, q) \,|\, (\exists p \in S)\, (p, a, q) \in T^{i-1}_{bq} \wedge (p', \tau, p) \in T^*_{\tau p}\right\} \text{and}$$
$$T^*_{bq} = T^{n|Act|}_{bq}$$

Finally the saturated LTS is:

$$T^* = \bigcup_{p \in S} \left(T^*_{\tau p} \cup T^*_{bp}\right) = \left(\xrightarrow{\tau}\right)^* \cup \left\{(p, a, q) \,|\, a \neq \tau \wedge (\exists p', q' \in A)\, p \left(\xrightarrow{\tau}\right)^* p' \xrightarrow{a} q' \left(\xrightarrow{\tau}\right)^* q\right\}$$

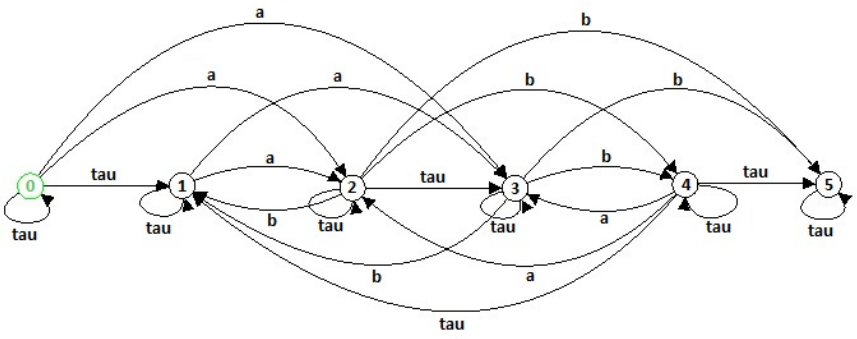

Fig. 5. Example saturated LTS

Having computed the observational equivalence (weak bisimilarity) of the LTS, the process of minimizing the original LTS is the same as the process for minimization modulo strong bisimilarity applied on the saturated LTS.

4.3 Comparison of two LTSs modulo strong 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

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.

4.4 Comparison of two LTSs modulo weak bisimilarity

The comparison of two LTSs modulo weak bisimilariy amounts to checking strong bisimilarity over the saturated LTSs [AILS07]. In another words, two LTSs are weakly bisimilar iff their saturated LTSs are strongly bisimilar. Following this fact, we implemented the comparison of two LTSs modulo weak bisimilarity by applying the saturation algorithm over the original LTSs in order to obtain their saturated LTSs, after which the process of comparison of the saturated LTSs modulo strong bisimilarity was applied as described above.

5 HML and U^w and U^s operators

Henessy-Milner logic [HM85] is a multimodal logic used to characterize the properties of a system which describe some aspects of the system's behaviour. Its syntax is defined by the following BNF grammar [**?**]:

$$\phi ::= tt \mid ff \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \langle a \rangle \phi \mid [a]\phi \tag{4}$$

A single HML formula can only describe a finite part of the overall behavious of a process. Therefore, the Henessy-Milner logic was extended to allow for recursive definitions by the introduction of two operators: the so called 'strong until' operator U^s and the so called 'weak until' operator U^w. These operators are expressed as follows [AILS07]:

$$\begin{aligned} F\ U^s\ G &\overset{min}{=} G \vee (F \wedge \langle Act \rangle tt \wedge [Act]\,(F\ U^s\ G)) \\ F\ U^w\ G &\overset{max}{=} G \vee (F \wedge [Act]\,(F\ U^w\ G)) \end{aligned} \tag{5}$$

The BNF grammar describing the set of HML formulas with recursion is as follows [AI85]:

$$\phi ::= tt \mid ff \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \langle a \rangle \phi \mid [a]\phi \mid X \mid min\,(X, \phi) \mid max\,(X, \phi) \tag{6}$$

where X is a formula variable and $min\,(X, \phi)$ (respectively $max\,(X, \phi)$) stands for the least (respectively largest) solution of the recursion equation $X = \phi$.

5.1 HML parsing

Our implementation of the Hennessy-Milner logic works as follows: For a given HML expression and a labeled transition system, we should get an output whether that expression is valid for the corresponding labeled graph. One HML expression can be made of these set of tokens {"AND", "OR", "UW", "US", "NOT", "[", "]", "⟨", "⟩", "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.

5.2 U^w and U^s implementation

The implementation of the U^w and U^s operators was done as follows. First we get the current state which is the state that contains all the nodes that satisfied the previous actions. For example if we have expression

$$\langle a\rangle\langle a\rangle TT\ U^s\ \langle b\rangle TT,$$

then 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 node. If we cannot do the action b, then we repeat the process, do action a again and check if can we do action 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 operator's type (strong or weak). If the operator is strong, then only one node in its state is enough. If it is weak, we check if there are no nodes in its new state or if we can get from the starting node to some of its neighbors through the action b.

6 Application

The tool we have developed is still in early development phase and as such it is very simple, however it still has the functionality needed to perform modeling, formal specification and verification of concurrent systems. Here we demonstrate that process of formal specification and verification for two classical problems in the concurrency theory: the Alternating Bit Protocol and the mutual exclusion algorithm of Peterson.

6.1 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 [AILS07][Kul94].

Modelling and Specification. The representation of the Alternating Bit Protocol (abbreviated as ABP in the rest of the section) consists of a *sender* S, a *receiver* R and two channels *transport* T and *acknowledge* A as shown below.

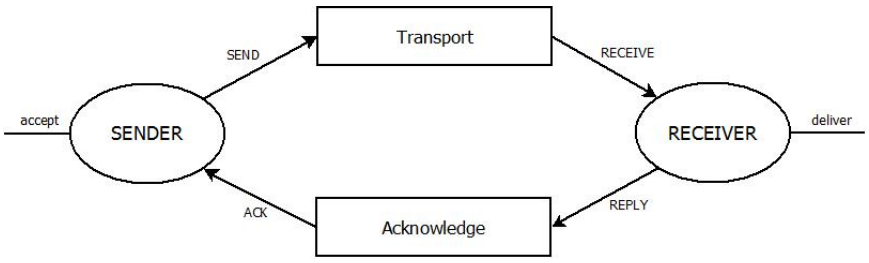

Fig. 6. Alternating Bit Protocol

The only visible transitions in the ABP are *deliver* and *accept*, which can occur only sequentially, whereas all others are internal synchronizations. Sender S sends a message to a receiver R. Each message sent by S contains the protocol bit, 0 or 1. 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 functioning of ABP can be described as follows [AILS07]:

1. When a sender S sends a message, it sends it repeatedly (with its corresponding bit) until it receives an acknowledgment ($ack0$ or $ack1$) from the receiver R that contains the same protocol bit as the message being sent. Therefore, the behaviour of the process representing the sender can be described as:

$$S = \overline{send0}.S + ack0.accept.S_1 + ack1.S$$
$$S_1 = \overline{send1}.S_1 + ack1.accept.S + ack0.S_1$$

   The transport channel transmits the message to the receiver, but it may lose the message (lossy channel) or transmit it several times (chatty channel). Therefore, the description of the behaviour of the process representing the

transport channel is given with the CCS expression:

$$\begin{aligned} T &= send0.\,(T + T_1) + send1.\,(T + T_2) \\ T_1 &= \overline{receive0}.\,(T + T_1) \\ T_2 &= \overline{receive1}.\,(T + T_2) \end{aligned}$$

2. When R receives a message, it sends a reply to S that includes the protocol bit of the message received. When then receiver will receive the message for the first time, it will deliver it for processing, while subsequent messages with the same bit will be simply acknowledged. That yields the following CCS expression for the receiver:

$$\begin{aligned} R &= receive0.\overline{deliver}.R_1 + \overline{reply1}.R + receive1.R \\ R_1 &= receive1.\overline{deliver}.R + \overline{reply0}.R_1 + receive0.R_1 \end{aligned}$$

   Again, the acknowledgement channel sends the *ack* to sender, and it can also acknowledge it several times or lose it on the way to the sender. Therefore the CCS expression describing the ackowledgement channel and its behaviour is given as follows:

$$\begin{aligned} A &= reply0.\,(A + A_1) + reply1.\,(A + A_2) \\ A_1 &= \overline{ack0}.\,(A + A_1) \\ A_2 &= \overline{ack1}.\,(A + A_2) \end{aligned}$$

3. 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][AG09]

Having described the behaviour of the ABP components, the CCS process term describing the behaviour of the protocol as a whole can be obtained as a parallel composition of the processes describing the sender, the transport channel, the receiver and the acknowledgement channel, restricted on the set of actions:

$$L = (send0, send1, receive0, receive1, reply0, reply1, ack0, ack1)$$

Therefore, the CCS implementation of the Alternating Bit Protocol is given as:

$$ABP \stackrel{def}{=} (S|T|R|A) \setminus L \tag{7}$$

On the other hand, ABP should act as a simple buffer, therefore its CCS specification is defined as follows:

$$\begin{aligned} Buf &= accept.Buf' \\ Buf' &= \overline{deliver}.Buf \end{aligned} \tag{8}$$

Verification. In order to verify the Alternating Bit Protocol, we need to prove that the implementation meets the specification with respect to some behavioural equivalence. We will show that an observational equivalence between Buf and

ABP can be found, i.e. that $ABP \approx Imp$. For that purpose, first we use our tool to obtain the labelled graphs corresponding to the CCS representations of Buf and ABP, and afterwards we perform a comparison of the LTSs modulo weak bisimilarity which yields a positive answer about the existance of weak bisimulation equivalence between Buf and ABP.

The weak bisimulation equivalence obtained by running any of the two bisimulation algorithms implemented in our tool over the saturated LTSs is given in Table 2:

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

Table 2. Verification of Alternating Bit Protocol using weak bisimilarity (observational equivalence)

6.2 Peterson's algorithm - Modeling, Specification and Testing

Peterson's algorithm [Pet81] is a simple algorithm designed to ensure mutual exclusion between two processes without any special hardware support. It represents a simple refinement of ideas from earlier mutex algorithms such as Dekker's algorithm [Dij68]. Mutual exclusion (often abbreviated as mutex) algorithms are used in concurrent programming to avoid the simultaneous use of a common resource by critical sections. A critical section is a piece of code in which a process or thread accesses a shared resource.

Modeling and Specification. In Peterson's algorithm for mutual exclusion, there are [AILS07]:

– Two processes P_1 and P_2 that want to access the same resource, i.e. eneter the critical section;
– Two shared variables b_1 and b_2 which indicate whether process P_1 and process P_2 are trying to enter the critical section;
– A shared integer variable k that can take one of the values 1 or 2 and indicates which process is next to enter the critical section;

The boolean variables b_1 and b_2 are initialized to values $'false'$ because neither of the processes is interested yet to enter the critical section, whereas the initial value of k is arbitrary.

In order to ensure mutual exclusion, each process P_i, $i \in \{1, 2\}$, executes the following algorithm presented in pseudocode, where j denotes the index of the other process.

Algorithm 1 Peterson's algorithm pseudocode

```
while true do
  '¡noncritical section¿';
  b_i ← true;
  k ← j;
  while (b_j and k = j) do
    skip;
  end while
  '¡critical section¿';
  b_i ← false;
end while
```

Modeling the algorithm of Peterson includes, among other tasks, translation of the algorithm's pseudocode description of the behaviour of the processes P_1 and P_2 into the model of CCS or LTSs.

Following the message-passing paradigm on which CCS is based, the variables manipulated by the processes P_1 and P_2 are viewed as passive agents that react to actions performed by the processes. The description of the variables used in Peterson's algorithm as processes can be done as follows [AILS07]:

1. The process representing the shared boolean variable b_1 has two states and its behaviour can be represented by the following CCS expressions:

$$B1f = \overline{b1rf}.B1f + b1wf.B1f + b1wt.B1t$$
$$B1t = \overline{b1rt}.B1t + b1wf.B1f + b1wt.B1t$$

Similarly for the process describing the behaviour of the variable b_2:

$$B2f = \overline{b2rf}.B2f + b2wf.B2f + b2wt.B2t$$
$$B2t = \overline{b2rt}.B2t + b2wf.B2f + b2wt.B2t,$$

where the pattern for the channel name is $b < i >< x >< y >$, with
– $i \in \{1, 2\}$ for the process ID

- $x \in \{r, w\}$ for the type of operation (read or write)
- $y \in \{f, t\}$ for the variable value to be written or read (false or true)

2. The process representing the variable k has two states, denoted by the constants K_1 and K_2, because the variable k can only take one of the two values 1 and 2, and its CCS representation is as follows

$$K1 = \overline{kr1}.K1 + kw1.K1 + kw2.K2$$
$$K2 = \overline{kr2}.K2 + kw2.K2 + kw2.K2,$$

where the pattern for the channel name is $k < x >< n >$, with
- $x \in \{r, w\}$ for the type of operation (read or write)
- $n \in \{1, 2\}$ for the value to be written or read

The final step is the CCS formalisation of the behaviour of the processes P_1 and P_2. The process behaviour outside of the critical region can be ignored and the focus can be put on the process entering and exiting the critical section. For simplicity, an assumption is made that the processes cannot fail or terminate within the critical section. Under these assumptions, the initial behaviour of the process P_1 can be described by the following CCS expression:

$$P1 = \overline{b1wt}.\overline{kw2}.P11,$$

where $P11$ models the while loop (with short-circuit evaluation):

$$P11 = b2rf.P12 + b2rt.\,(kr2.P11 + kr1.P12)$$

and $P12$ models the critical section:

$$P12 = enter1.exit1.\overline{b1wf}.P1$$

The behaviour of the process P_2 can be described with CCS expressions in the similar manner:

$$P2 = \overline{b2wt}.\overline{kw1}.P21$$
$$P21 = b1rf.P22 + b1rt.\,(kr1.P21 + kr2.P22)$$
$$P22 = enter2.exit2.\overline{b2wf}.P2$$

Finally, the CCS process term representing the whole Peterson's algorithm consists of the parallel composition of the terms describing the two processing running the algorithm and of those describing the variables. Since we are only interested in the behaviour of the algorithm pertaining to the access to, and exit from, their critical sections, we shall restrict all the communication channels that are used to read from, and write to, the variables. That set of restricted channel names is

$$L = \{b1rf, b1rt, b1wf, b1wt, b2rf, b2rt, b2wf, b2wt, kr1, kw1, kr2, kw2\}$$

Assuming that the initial value of the variable k is 1, our CCS description of Petersons algorithm is therefore given by the term:

$$PETERSON \stackrel{def}{=} (B1f|B2f|K1|P1|P2) \setminus L \tag{9}$$

On the other hand, the desired behaviour of the Peterson's algorithm is as the one of any simple mutex algorithm. Initially, both processes enter their critical sections, however once one of the processes has entered its critical section, the other process cannot enter its own critical section until the first process has exited its critical section. Therefore, a suitable CCS specification of the behaviour of a mutual exclusion algorithm like Peterson's is given as follows:

$$MutexSpec = enter1.exit1.MutexSpec + enter2.exit2.MutexSpec \tag{10}$$

Testing. Testing the preservation of the mutual exclusion property is one of the approaches that can be used to establish the correctness of the Peterson's algorithm. A test is a finite rooted LTS over the set of actions $Act \cup \{\overline{bad}\}$, where *bad* is a distinguished channel name not occurring in *Act*. The idea is that the test would act as a monitor process that 'observes' the behaviour of a process and reports an error in case of an occurrence of an undesirable situation by performing a *bad*-labelled transition. Assuming that the monitor process outputs 'bad' when it discovers that two enter actions have occurred without intervening exit, a CCS process describing this behaviour is [AILS07]:

$$\begin{aligned} MutexTest &= \overline{enter1}.MutexTest1 + \overline{enter2}.MutexTest2 \\ MutexTest1 &= \overline{exit1}.MutexTest + \overline{enter2}.\overline{bad}.0 \\ MutexTest2 &= \overline{exit2}.MutexTest + \overline{exit1}.\overline{bad}.0 \end{aligned}$$

Now, in order to check whether process $PETERSON$ ensures mutual exclusion, it is now sufficient to let it interact with $MutexTest$ and see if the resulting system

$$(PETERSON | MutexTest) \setminus M, \tag{11}$$

where

$$\{enter1, enter2, exit1, exit2\},$$

can initially perform the action $\overline{bad}$.

Indeed, the LTS of the above CCS expression generated by our tool does not have states which afford *bad* transitions. This proves that Peterson's algorithm ensures mutual exclusion.

7 Conclusions and Future Work

In this paper we presented a tool for modeling, manipulation and analysis of concurrent systems. It includes recognizing CCS expressions, building LTS graphs and checking equivalence between two LTS graphs with respect to strong and weak bisimilarity. The tool is very simple, but yet functional enough to perform specification and verification of concurrent systems described as CCS expressions and/or LTS graphs. We have successfully tested the tool with large number of examples comparing the results obtained with mCRL2 and CWB, and we also presented few examples showing the tool usage.

Future work can include optimization of the tool response times for very large LTS graphs, as well as implementation of prunung strategy for infinite LTS graphs. It would be also usefull if the LTS graph is better visualized where every SOS rule is shown, with its source and sink nodes showing the CCS expression for each of the nodes. Such a visualization can be very helpfull for understanding the semantics of the CCS language for students that are starting to study it. Furthermore, the LTS minimization and LTS comparison functionality which at the moment includes only strong and weak bisimilarity can be extended to include other behavioural equivalences as well. Another direction for future development can be to extend the tool to support not only CCS expressions, but variety of other process semantics as well.

Acknowledgment

We would like to thank d-r Jasen Markovski for introducing us to the theory of formal methods and its application, being our mentor and guiding us all throughout the project with his suggestions and constructive feedback.

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

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

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

CWB. Perdita Stevens et al. *The Edinburgh Concurrency Workbench*, `http://homepages.inf.ed.ac.uk/perdita/cwb/summary.html`

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

GR09a. J.F. Groote and M. Reniers, *Modelling and Analysis of Communicating Systems,* Technical University of Eindhoven, rev. 1478, pages 67-69, 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

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

AG09. M.Alexander and W.Gardner, *Process Algebra for Parallel and Distributed Processing*, Chapman&Hall/CRC Press, Computational Science Series, pages 114-118, 2009

HM85. M. Hennessy and R. Milner, *Algebraic laws for nondeterminism and concurrency*, J. Assoc. Comput. Mach., pages 137-161, 1985
AI85. Luca Aceto and Anna Ingolfsdottir, *Testing Henessy-Milner logic with recursion*, BRICS Report Series, pages 3-5, 1998
Pet81. Gary L. Peterson, *Myths about the mutual exclusion problem*, Information Processing Letters, 12(3), pages 115-116, 1981
Dij68. E. W. Dijkstra, *Co-operating sequential processes*, In F. Genuys, editor, Programming Languages, pages 43112. Academic Press, New York, 1968 Reprinted from: Technical Report EWD-123, Technological University, Eindhoven, the Netherlands, 1965
mCRL2. Technical University of Eindhoven, LaQuSo, CWI and University of Twente, *micro Common Representation Language 2 (mCRL2)*, `http://www.mcrl2.org`

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 labeled transition systems suitable for proper testing of bisimulation equivalence. Therefore, we used the academical examples from mCRL2 as experimental test models.

The experiments were conducted on a portable computer with the following specifications:

1. CPU: Intel Pentium P6100 2.0 GHz
2. Memory: 3 GB
3. OS: Windows 2007 Premium

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.

The results from the experiments carried out are presented graphically in Fig. 7 and Fig. 8 for the naive algorithm and the algorithm of Fernandez, respectively.

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

			Naive		Fernandez				Ratio
aut	states	transitions	$t(ms)$	$\Delta t(ms)$	$t(ms)$	$\Delta t(ms)$	bisimilar pairs	bisimilar classes	t_n/t_f
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 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. 9. 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. 10 shows the ratios of the running times of the two algorithms for each of the examples.

Appendix B: CCS grammar

One of the few standard ways of showing a grammar is a syntax diagram. On Fig. 11 we show the syntax diagram for the grammar that recognizes CCS expressions and is used in our tool.

Appendix C: HML grammar

The grammar that recognizes HML expressions and is used in our tool is given below.

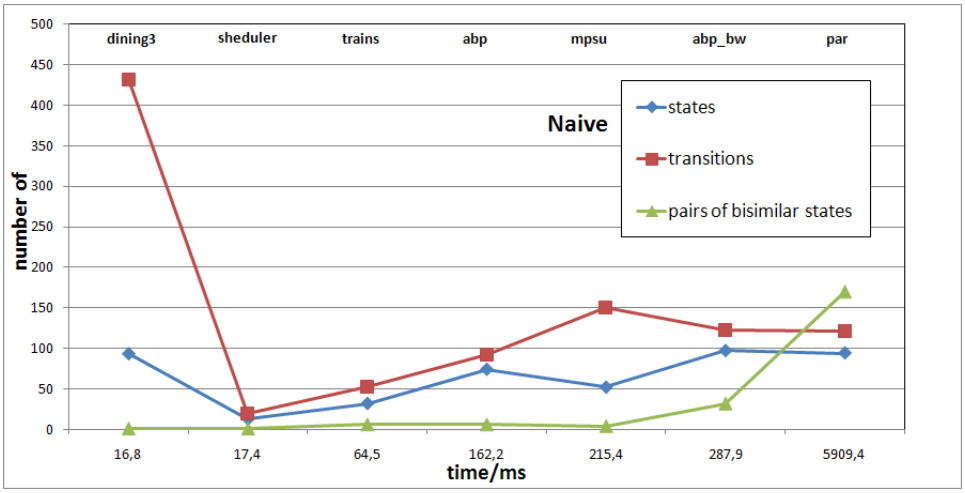

Fig. 7. Analysis of the running time of the naive bisimulation algorithm

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

Appendix D: Tool Usage

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

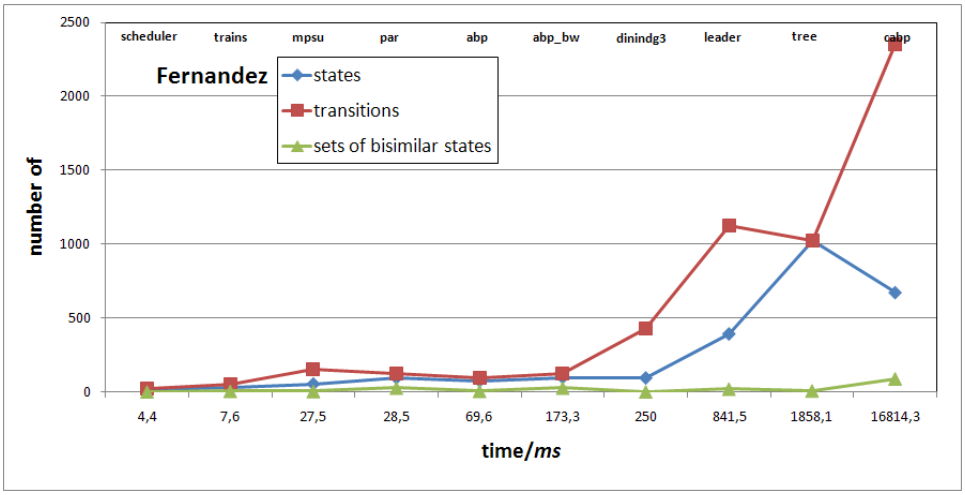

Fig. 8. Analysis of the running time of the bisimulation algorithm due to Fernandez

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 Fig. 12 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 Fig. 13 and Fig. 14 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 Fig. 15 and Fig. 16 two LTSs have to be loaded from file. In order to check for LTS bisimilarity the user has to chose a method for calculation and click on the button "Calculate". The results will be displayed on the right side of the button and they give information whether the LTSs are bisimilar and how long the calculation lasted.

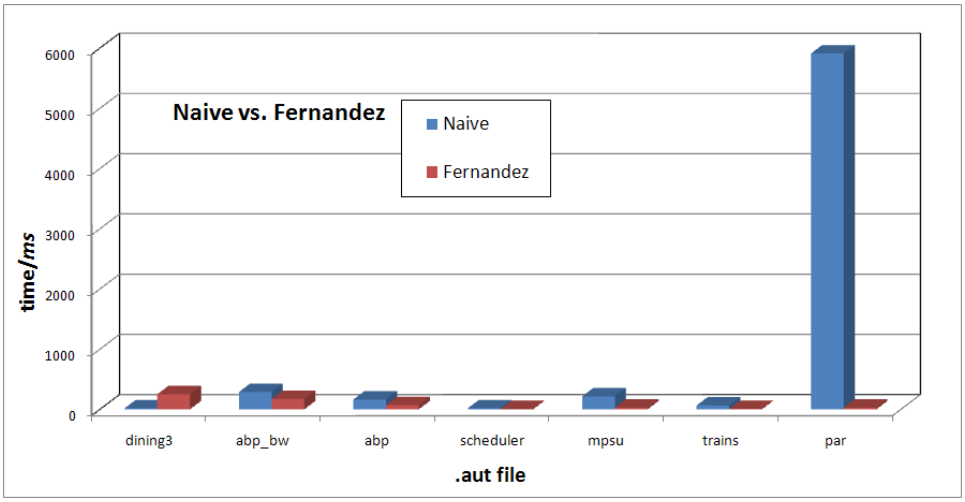

Fig. 9. Comparison of the running time of the two bisimulation algorithms

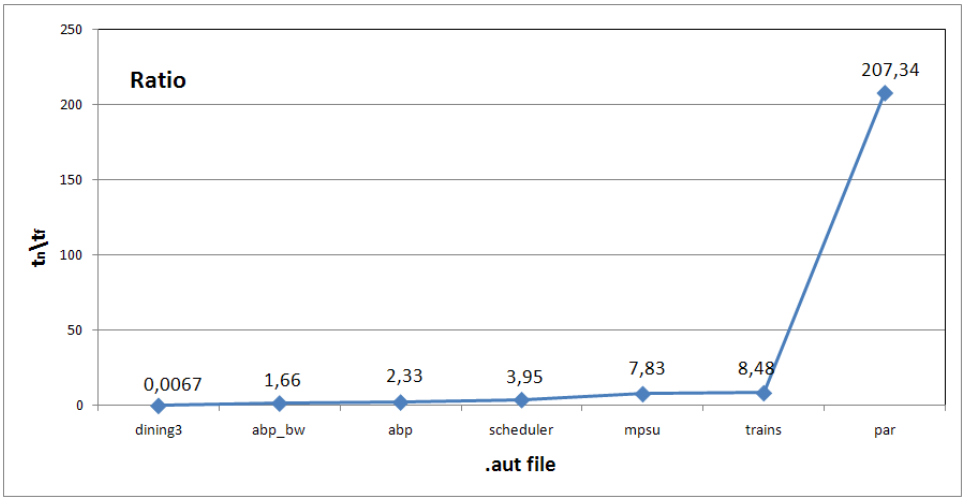

Fig. 10. Ratio of the running times of the two bisimulation algorithm

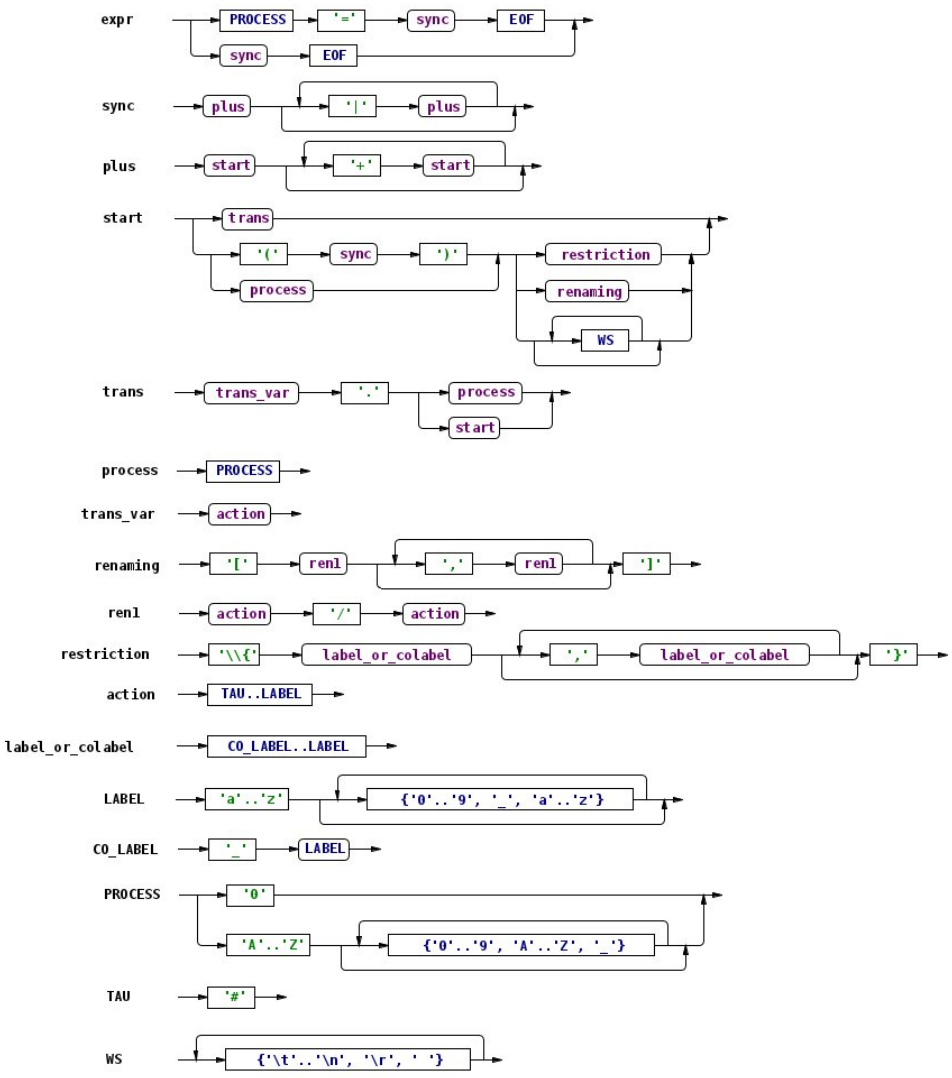

Fig. 11. Syntax diagram for the CCS grammar

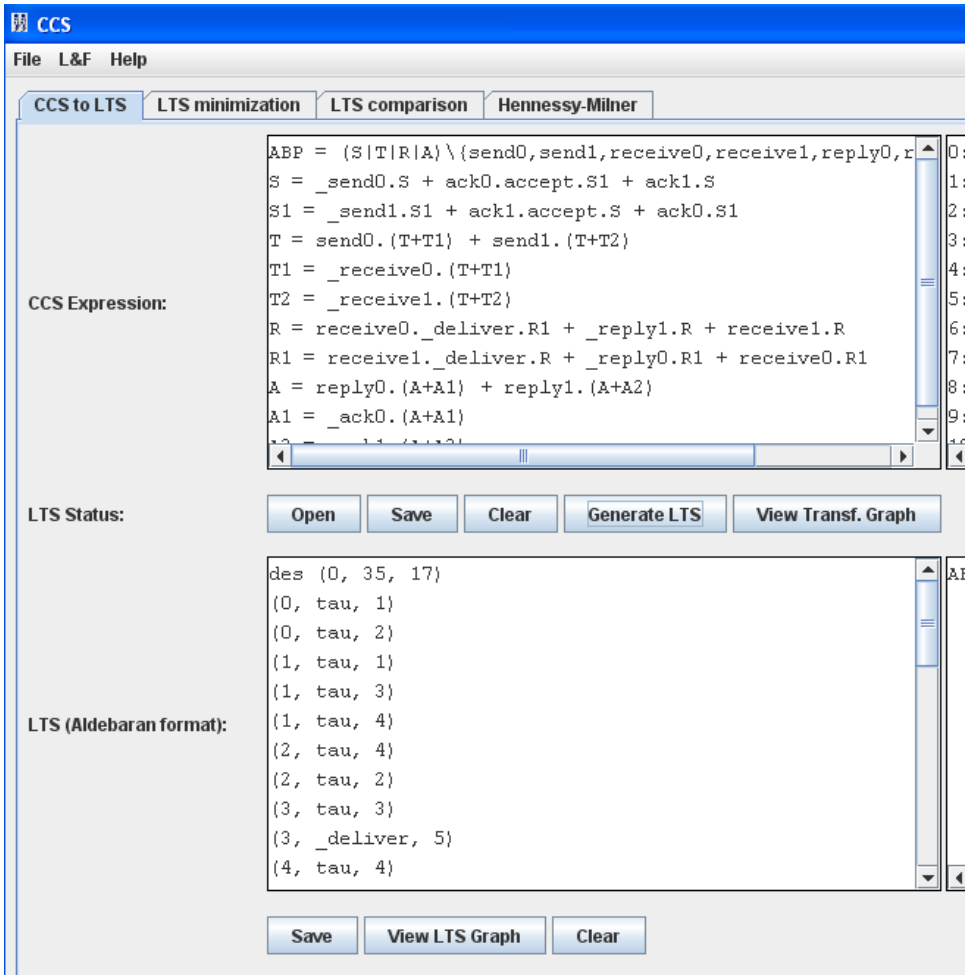

Fig. 12. Tool Usage: ABP Modeling and Generating LTS

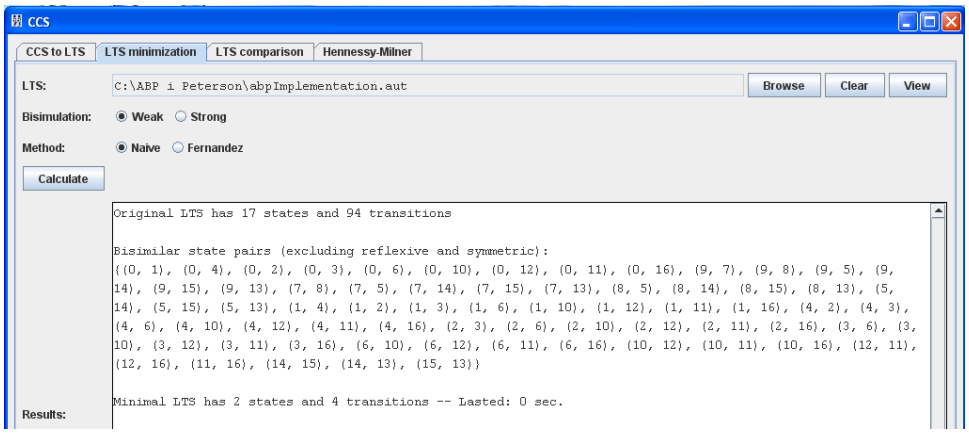

Fig. 13. Tool Usage: ABP Minimization, Weak, Naive

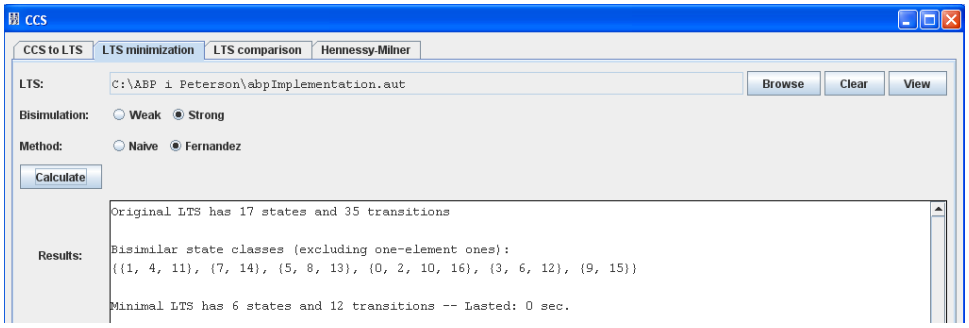

Fig. 14. Tool Usage: ABP Minimization, Strong, Fernandez

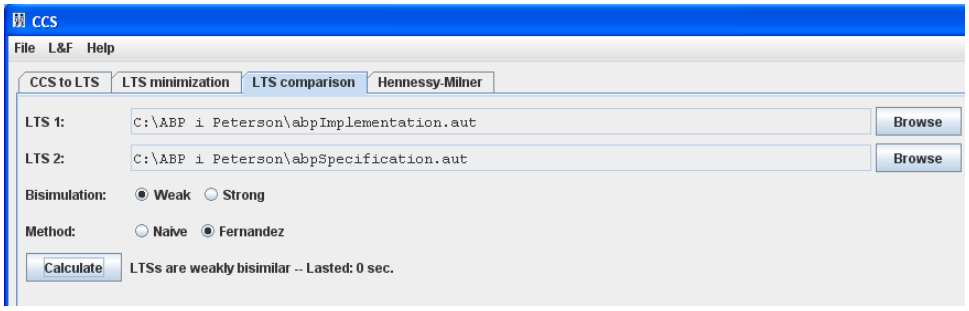

Fig. 15. Tool Usage: ABP Bisimilarity, Weak, Fernandez

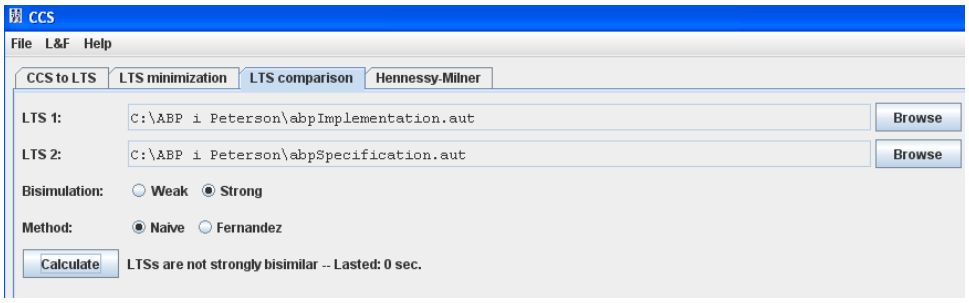

Fig. 16. Tool Usage: ABP Bisimilarity, Strong, Naive