Formal Methods Technical Report

Jane Jovanovski *†, Maja Siljanoska *‡, Vladmimir Carevski *§, Dragan Sahpaski *¶,
Petar Gorcevski *‖, Metodi Micev ***, Bojan Ilijoski *†† and Vlado Georgiev *‡‡
*Institute of Informatics
Faculty of Natural Sciences and Mathematics,
University "Ss. Cyril and Methodius",
Skopje, Macedonia
Email: †janejovanovski@gmail.com, ‡maja.siljanoska@gmail.com,
§carevski@gmail.com, ¶dragan.sahpaski@gmail.com, ‖pgorcevski@gmail.com,
**metodi.micev@gmail.com, ††b.ilijoski@gmail.com, ‡‡vlatko_gmk@hotmail.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 s certain system.**

I. Introduction

This paper is organized in four chapters. The first chapter is devoted to the problem of parsing CCS's expressions and generation of Labelled Transition System and also discusses some of the choices made during the implementaion. The second chapter discusses the problem of saturation of LTS and implementation details. The third 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 forth chapter shows typical examples of modeling, specification and verification of example systems. The first example system is Alternating bit problem 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.

II. Parsing and LTS generation

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

$$P ::= \emptyset \mid a.P_1 \mid A \mid P_1 + P_2 \mid P_1|P_2 \mid P_1[b/a] \mid P_1\backslash a \quad (1)$$

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

$$V \to w$$

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 description of CCS's process is CFG. Deterministic context-free grammars (DCFGs) are a proper subset of the context-free grammars. The DCFGs are those that a deterministic pushdown automaton can recognize, or equivalently DCFGs are those grammars that a LR parser can recognize. Although, the defined grammar is 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 some parser generator that will generate LR or LL parser which is able to recognize the define grammar.

ANTLR (ANother Tool for Language Recognition) was used to generate the parser. ANTLR is a parser generator that uses LL(*) parsing. ANTLR allows generating parsers, lexers, tree parsers and combined lexer parsers. ANTLR automatically generates abstract syntax trees which can be further processed with a tree parser. A language is specified by using context-free grammar which is expressed using Extended Backus Naur Form (EBNF). In short, 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 well defined abstract syntax trees (AST). The AST is 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

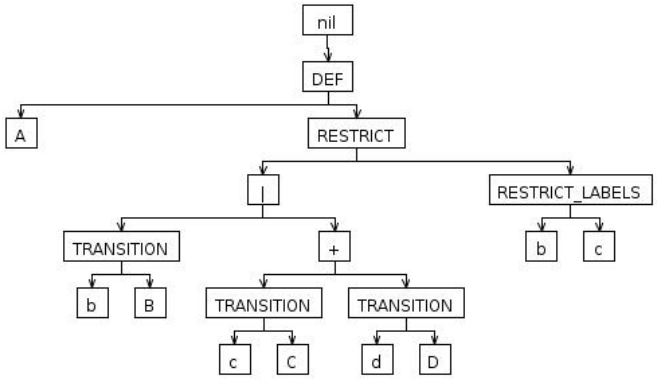


Fig. 1. AST Example

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. The tree in the figure is a 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 tree causes a change in the implemented algorithms. Beacause of this, a specific domain model is built and a domain builder (DB). The input of the domain builder is a AST and the output is a fully built domain model objects. It is not expected for the domain model to change, so all algorithms are implemented on the domain model.

III. 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 [**?**].

The bisimulation equivalence finds its extensive application in the area of formal verification of concurrent systems, for example to check the equivalence of an implementation of a certain system with respect to its specification model.

In our tool the process of determining an existance of a bisimulation equivalence between two labeled transition systems was implemented using an approach which consists of three steps:

1) Computing strong bisimulation equivalence (strong bisimilarity) for each of the two LTSs;
2) Minimizing each of the two LTSs to its canonical form using the strong bisimilarity obtained in the first step;
3) Performing a comparison between the two canonical forms obtained in the second 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 [**?**] for computing bisimulation equivalence stems from the theory underlying Tarski's fixed point theorem [**?**]. It has been proven that the strong bisimulation equivalence is the largest fixed point of the monotic function F as defined in [**?**] 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.

In our implementation 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.

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 [**?**], [**?**].

The algorithm due to Fernandez in our 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.

Both algorithms were implemented in Java. Several different data structures were used to implement the structure labelled graph. The labelled graph is represented as a list of nodes. Each node is represented by the number of the corresponding state, as a start state, and a list whose elements are couples of an outgoing action and a reachable state.

The next step uses the bisimulation equivalence computed

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

in the first step in order to minimize the graphs. 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.

Having reduced the two labeled graphs into their minimal forms, the last step in the process of checking the equivalence between the two labeled transition system consists of checking whether the two minimal labeled graphs are isomorphic.

Two graphs G and H are isomorphic if there exists a graph isomorphism between them. According to [**?**], a graph isomorphism between two graphs G and H is a bijective function $f : Nodes(G) \rightarrow Nodes(H)$ satisfying:

- $f(initialNode(G)) = g(initialNode(H))$
- (s, a, t) is an edge in $G \Leftrightarrow (f(s), a, f(t))$ is an edge in H

The graph isomorphism check is implemented recursively as follows: Starting from the initial states of the minimal graphs, the outgoing labeled transitions have to be same, and for every outgoing transition with a certain label, there has to be another one in the other graph with the same label that leads again to isomorphic states.

The two graphs are bisimilar.

The correctness of the implementation was tested with the use of ltsconvert and ltscompare tools of MCRL2, ...[ref6]

The whole process as described above is illustrated on the examples in Fig

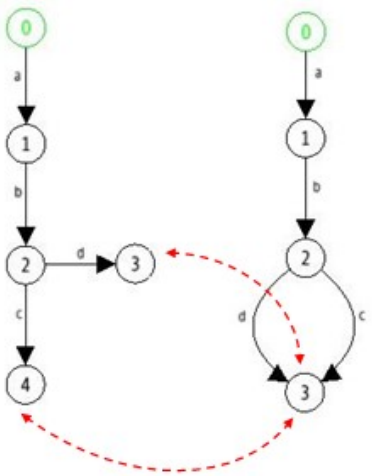
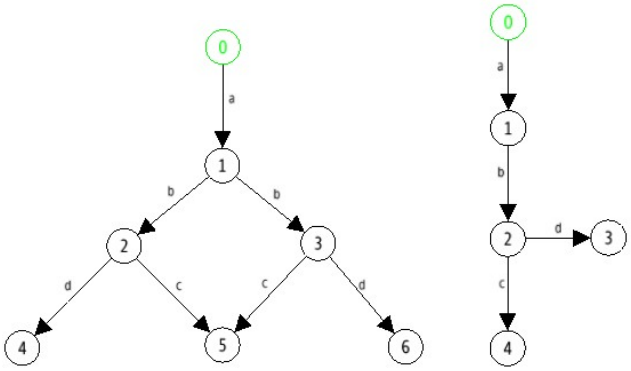


Fig. 2. Minimized Graph 2

An illustration of the output of the algorithms for the graphs shown on Fig. 2 and Fig. 3 is given in TABLE I.

The process of reduction of the Graph 1 to its minimal form is given on Fig. 4. As it can be seen from the figure, the states



(a) Graph 1 (b) Graph 2

Fig. 3. Example of two labeled graphs

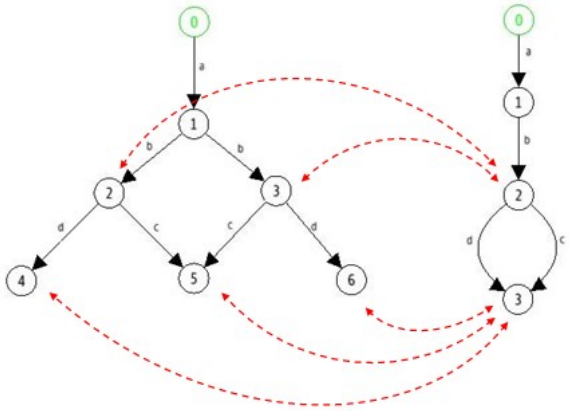


Fig. 4. Minimized Graph 1

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

TABLE I
COMPUTING STRONG BISIMULARITY FOR GRAPH 1 AND GRAPH 2

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.

The process of reduction of the Graph 2 to its minimal form is given on Fig. 5. As before, the states 3 and 4 are merged into state 3 in the minimal graph.

IV. PETERSON AND HAMILTON ALGORITHM - MODELING, SPECIFICATION AND TESTING

The algorithm

var flag: array [0..1] of boolean; turn: 0..1; flag[0] := FALSE; flag[1] := FALSE; turn := random(0..1)

repeat flag[i] := TRUE; turn := j; while (flag[j] and turn=j) do no-op; flag[i] := FALSE; until FALSE;

There are two different processes which want to enter at critical section. This means that the processes are fighting for

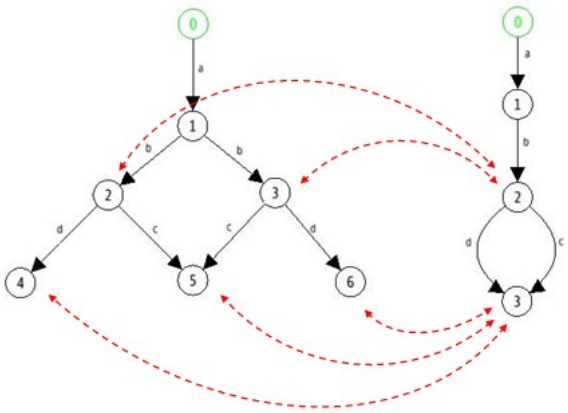


Fig. 5. Minimized Graph 1

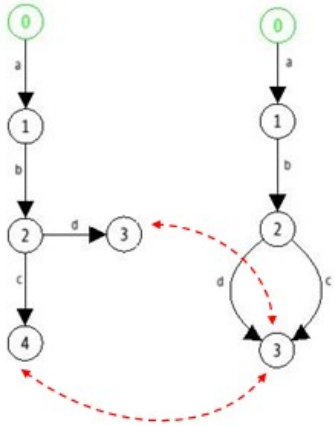


Fig. 6. Minimized Graph 2

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 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

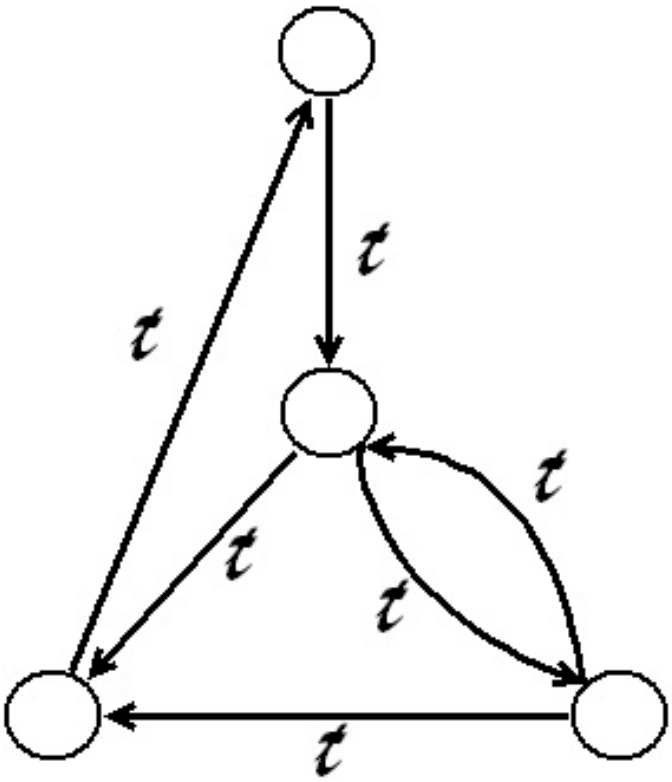


Fig. 7. AST Example

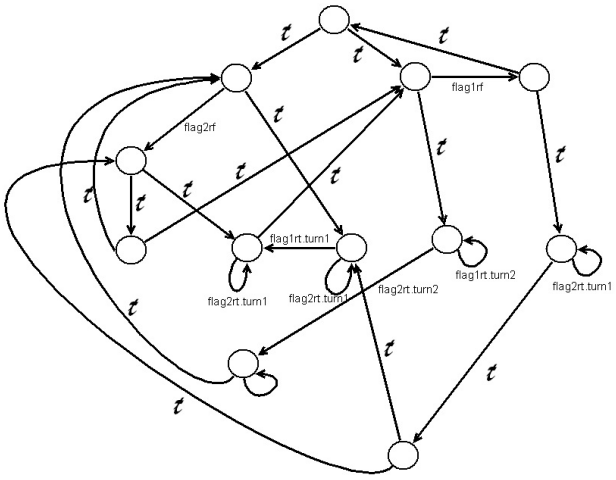


Fig. 8. AST Example

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

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 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	

V. Conclusion

The conclusion goes here.

Acknowledgment

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

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

[BPS01] J.A. Bergstra, A. Ponse and S.A. Smolka, *Handbook of process algebra,* Elsevier Science B.V., pages 9-13, 2001