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 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:

$$V \to w$$

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 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 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

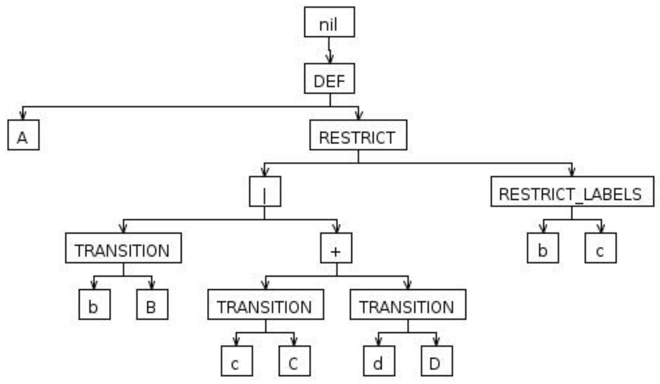


Fig. 1. AST Example

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 expected 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.

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 [BK08].

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 [AILS07a] for computing bisimulation equivalence stems from the theory underlying Tarski's fixed point theorem [AILS07b]. It has been 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 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 implementation in Java 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:

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

- $T_a[p] = \{q\}$ - an a-transition from state p to state q
- ${T_a}^{-1}[q] = \{p\}$ - an inverse a-transition from state q to state p
- ${T_a}^{-1}[B] = \cup \{{T_a}^{-1}[q], q \in B\}$ - 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 uses the bisimulation equivalence computed 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 $r \overset{a}{\to} p$ and $s \overset{a}{\to} q$ are replaced by transitions $r \overset{a}{\to} k$ and $s \overset{a}{\to} k$;
3) All outgoing transitions $p \overset{a}{\to} r$ and $q \overset{a}{\to} s$ are replaced by transitions $k \overset{a}{\to} r$ and $k \overset{a}{\to} s$;
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 systems consists of checking whether the two minimal labeled graphs are isomorphic. Two isomorphic graphs are always bisimilar [BPS01b].

Two graphs G and H are isomorphic if there exists a graph isomorphism between them. According to [BPS01a], a graph isomorphism between two graphs G and H is a bijective function $f : Nodes(G) \to Nodes(H)$ 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

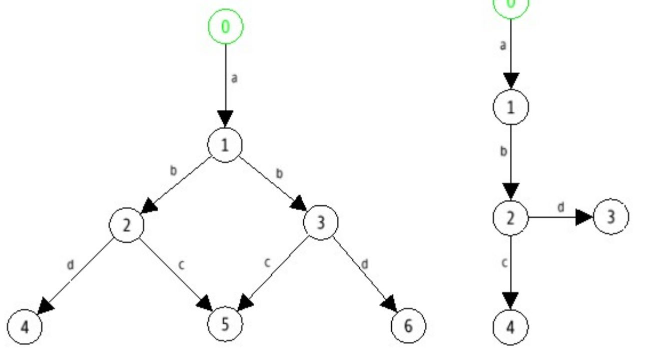
The graph isomorphism check was implemented recursively as follows: Starting from the initial states of the minimal graphs, the outgoing labeled transitions have to be same, and for every outgoing transition with a given label in the first graph, there has to be another one in the other graph with the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

The process of determining equivalence of two labeled transition systems with respect to strong bisimularity is illustrated on the graphs Graph 1 and Graph 2 as input labeled graphs.

The result from the first step, computing the strong bisimulation equivalence, is given in TABLE I for both the naive algorithm and the advanced algorithm due to Fernandez:

The second step involves minimization of the input graphs using the bisimulation equivalence obtained in the first step.

The process of reduction of Graph 1 to its minimal form is given on Fig. 3. As it can be seen from the figure, all mutually bisimilar states are merged into a single state: 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.



(a) Graph 1 (b) Graph 2

Fig. 2. Example of two labeled graphs

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

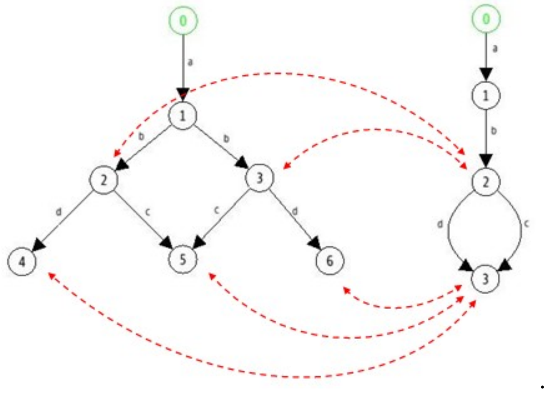


Fig. 3. Minimized Graph 1

Similarly, Fig. 4 illustrates the process of reduction of Graph 2 to its minimal form. Here, the states 3 and 4 are merged into state 3 in the minimal graph.

In the last step, the two minimal graphs from 3 and 4 are compared for graph isomorphism. From the figures, it can be seen that the minimal graphs are isomorhic which implies that the two input graphs Graph 1 and Graph 2 are bisimular and they exhibit the same behaviour.

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].

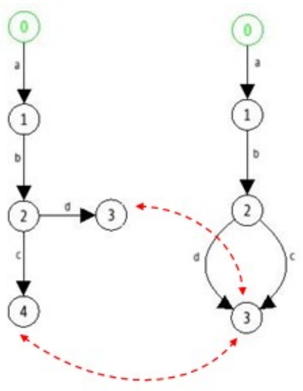


Fig. 4. Minimized Graph 2

IV. Peterson and Hamilton algorithm - Modeling, Specification and Testing

The algorithm:

```
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;
```

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 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

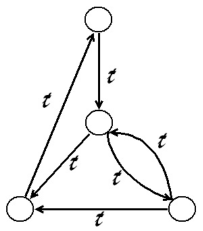


Fig. 5. Live cycle of one process

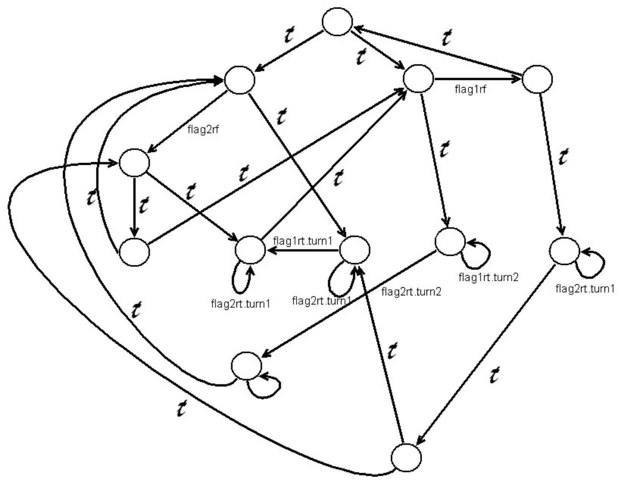


Fig. 6. Model of Peterson-Hamilton algorithm

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 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. U^w AND U^s IMPLEMENTATION

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", "[", "]", "⟨", "⟩", "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

VI. 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

[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

[mCRL2Ref] mCRL2
http://www.mcrl2.org