# 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 brunch 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 be 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. This tool if enough to perform modeling, specification and verification of 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 LTS and also discussed 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 [ANTLRR]. 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 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

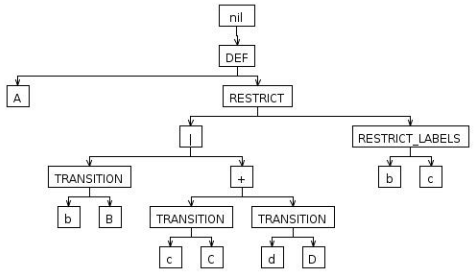


Fig. 1. AST Example

example, parentheses are implicit in the tree structure. From a development point of view it is much easier to work with trees that 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:

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

No tralalal 2..

## III. Bisimilarity

## IV. Conclusion

The conclusion goes here.


## Acknowledgment

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


## References


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

[ANTLRR] Terence Parr, *The Definitive ANTLR Reference - Building Domain-Specific Languages*, The Pragmatic Bookshelf, 2007

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