Fourth International Conference
Modelling and Development of Intelligent Systems
October 28 - November 1, 2015
"Lucian Blaga" University
Sibiu - Romania

Generating a CTL model checker using an attribute grammar

Laura Florentina Stoica, Florin Stoica, Florian Mircea Boian

Abstract

The attribute grammars are presented as a formal approach for model checkers development. Our aim is to design a CTL model checker from a context-free grammar which generates the language of the CTL formulas. An attribute grammar may be informally defined as a context-free grammar that has been extended with set of attributes and a collection of evaluation rules. We are using a CTL attribute grammar for specifying an operational semantics of the language of the CTL formulas by defining a translation into the language which describes the set of nodes from the CTL model where the corresponding CTL formulas are satisfied. We provide a formal definition for an attribute grammar used as input for Another Tool for Language Recognition (ANTLR) to generate an algebraic compiler. Also, is presented the technique of implementing the semantic actions in ANTLR, which is the concept of connection between attribute evaluation in the grammar that generates the language of CTL formulas and algebraic compiler implementation that represents the CTL model checker.

1 Introduction

The process of verification of a CTL model requires defining a specification which is represented by a CTL formula, and then determining whether or not that specification it is satisfied in the model.

Such a specification is performed using the CTL formulas language, which is based on well-established syntactic rules.

Verification of a CTL formula involves a translation of it, from the language in which it was defined to the language over the set of states of the model. The result of this translation will be the set of states that satisfy the given formula in the checked CTL model.

Most often the designing a translator is difficult to achieve and require significant efforts for construction and maintenance.

There are now specialized tools that generate the full code required using a grammar specification of the source language.

Traditionally, the tools used for the two phases of the translation of the text, *lexical analysis* and *syntactic analysis* were LEX *(A Lexical Analyzer Generator)* and YACC ***(Y**et **A**nother **C**ompiler **C**ompiler),* or their GNU equivalent, FLEX, BISON, BYACC/J. The disadvantage of the tools LEX and YACC respectively FLEX and BISON is that they only generate *C* code and that code is not always easily understood by the user (complexity induced by the nature of analyzers they generate).

BYACC / J is able to generate Java code, but supported semantic actions are rudimentary.

A high-performance analyzer generator is ANTLR [1] ***(An**other **T**ool for **L**anguage **R**ecognition),* capable of generating C ++, C #, Java or Python code and represents the instrument used in this article. We will use Java as the target language into which will be developed our own CTL model checker tool.

The original contribution of our approach consists in development of the CTL model checker tool by designing an ANTLR attribute grammar upon which is then generated, using ANTLR, the entire model checker tool.

The evaluation of a CTL formula is done by automatic activation of the semantic actions associated with production rules in the process of walking the Abstract Syntax Tree (AST) built into the process of syntactic analysis of the respective CTL formula, supplied as input for CTL model checker tool.

2 The CTL model

A model is defined as a Kripke structure $M=(S, Rel, P:S\rightarrow 2^{AP})$ where S is a finite sets of states also called nodes, $Rel \subseteq S\times S$ is a transition relation denoting a set of directed edges, and P is a labelling function that defines for each state $s \in S$ the set $P(s)$ of all atomic propositions from AP that are valid in s. The transition relation Rel is left-total, i.e., $\forall\, s \in S\ \exists\, s' \in S$ such that $(s,s') \in Rel$.

For each $s \in S$, the notation $succ(s)=\{s' \in S\ |(s,s') \in Rel\}$ is used to denote the set of successors of s. From definition of Rel, each state from S must have at least one successor, that is $\forall s \in S$, $succ(s)\neq\varnothing$. A path in M is an infinite sequence of states $(s_0,s_1, s_2,...)$ such that $\forall i, i\geq 0$, we have $(s_i,s_{i+1}) \in Rel$.

We use $s' \in succ(s)$ to denote that there is a relation (s, s') in Rel. The labelling function P maps for each state $s \in S$ the set $P(s)$ of all atomic propositions from AP that are valid in s [2].

We use the function $P':AP\rightarrow 2^S$, which associates each atomic proposition with the set of states labeled with that atomic proposition, such that $P'(ap)=\{s \in S \mid ap \in P(s)\},\ \forall\, ap \in AP$.

3 CTL syntax and semantics

A CTL formula has the following syntax given in Backus-Naur Form (BNF) [2]:

$\varphi :: true|false|ap|(\neg\ \varphi_1)|\ \varphi_1 \wedge \varphi_2|\ \varphi_1 \vee \varphi_2|\ \varphi_1 \Rightarrow \varphi_2|\ AX\ \varphi_1|\ EX\ \varphi_1|\ AG\ \varphi_1|\ EG\ \varphi_1|\ AF\ \varphi_1|\ EF\ \varphi_1|\varphi_1 AU \varphi_2|\ \varphi_1 EU \varphi_2,\ \forall ap \in AP$.

A CTL specification is interpreted over Kripke structures. The set of all paths through a Kripke structure is assumed to correspond to the set of all possible computations of a system. CTL logic is branching-time logic, meaning that its formulas are interpreted over all paths beginning in a given state (an initial state) of the Kripke structure.

A CTL formula encodes properties that can occur along a particular temporal path as well as to the set of all possible paths. The CTL syntax include several operators for describing temporal properties of systems: *A* (for all paths), *E* (there is a path), *X* (at the next moment), *F* (in future), *G* (always) and *U* (until)

Syntactically, *CTL* formulas are divided into three categories:

- those whose outermost operator, if any, is not a temporal operator;
- those whose outermost operator is a temporal operator (*X* (next), *U* (until), *F* (eventually) or *G* (always)) prefixed with the existential path quantifier *E*, and
- those whose outermost operator is a temporal operator prefixed with the universal path quantifier *A*.

4 Formal specification of the CTL model checker tool

A CTL model checker tool consists of an algebraic compiler $\mathcal{C}_{MC}:L_{ctl}\rightarrow L_M$ where the source language is the language of CTL formulas and the target language is the language that describes the sets of nodes (states) of CTL models (represented by Kripke structures) in that these formulas are satisfied.

The effective building of the algebraic compiler requires the implementation of a procedure for calculating the generalized homomorphism which uniquely associates to any syntactic construction of the source language a syntactic construction of the target language [3].

The algebraic compiler $\mathcal{C}_{MC}$ translates a CTL formula to the set of nodes S' of a given model M, over which the *CTL* formula is satisfied. Therefore, $\mathcal{C}_{MC}(f)=S'$ where f is the CTL formula to be verified and $S'=\{s \in S \mid (M,s) \models f\}$.

In other words, the algebraic compiler receives as input syntactic constructions of the source language $w \in L_{ctl}$ which then it maps to syntactic constructions of the target language $\mathcal{C}_{MC}(w) \in L_M$.

$\mathcal{C}_{MC}$ is generated from the specifications that define the model checker as a generalized homomorphism between the algebra of CTL formulas and the algebra of the set of states of the model [4]. When the homomorphism is evaluated using as input an object of the source algebra (a CTL formula), the *derived operations* are evaluated to generate the target image of the respective formula into destination algebra, the obtained result being the set of states in which formula is satisfied.

In general, a derived operation is a computation associated with an operation of the source language and specified using syntactic constructions of the target language. Often, the operations and the elements provided by the target language algebra are not expressive enough to specify the correct translation which should be performed by the algebraic compiler.

For each function name *op* from the operator scheme of algebras of the source language L_{ctl} is created a specification rule as a pair $(op, d_{MC}(op))$, where $d_{MC}(op)$ denote the derived operation in the syntax algebra of the target language L_M through which are constructed the target images of constructions created in the source language by the *op*.

We note with O_{ctl} the finite set of names of the operators of the language L_{ctl}, and we have $O_{ctl} = \{\top, \bot, \neg, \wedge, \vee, \rightarrow, AX, EX, AU, EU, EF, AF, EG, AG\}$.

The set of pairs $\{ op, d_{MC}(op)) \mid op \in O_{ctl} \}$ represents the *compiler specification* that can be used to generate a compiler that will associate the words from the syntax algebra of the source language L_{ctl} with words from the syntax algebra of the target language L_M.

Implementation of the algebraic compiler $\mathcal{C}_{MC}: L_{ctl} \rightarrow L_M$ which represents the CTL model checker and practical performs the checking of the CTL formulas, can be described by the following recursive function:

```
function C_MC (f ∈ L_ctl)  {
     if (A_L (f)) {
          if ( f = true)  return S;
          else if (f = false) return ∅;
          else return P'(f);
     }
     else if (A_S(f)= (op, (f_1,…, f_n)))
          return d_MC(op) ( C_MC (f_1), ... , C_MC(f_n))
     else return error;
}
```

Fig. 1: The algebraic compiler as recursive function

For formula f, the function $\mathcal{A}_{\mathcal{L}}$ determines if it belongs to the set of generators of the L_{ctl} language. If $f \in \{\top, \bot\} \cup \{ap \mid ap \in AP\}$, $\mathcal{A}_{\mathcal{L}}(f)$ returns ***true,*** else the function returns ***false.*** $\mathcal{A}_S$ is a mechanism that determines the operation and subformulas which were used to create the formula f.

The components $\mathcal{A}_{\mathcal{L}}$ şi $\mathcal{A}_S$ of the algebraic compiler $\mathcal{C}_{MC}$ can be implemented by a lexical analyzer respectively by a parser.

The lexical analyser $\mathcal{A}_{\mathcal{L}}$ should identify the lexical atoms represented by atomic sentences correctly constructed, according to a regular grammar that generates the specification language of atomic propositions.

The parser $\mathcal{A}_S$ determines whether the formula used as input for the CTL model checker is properly constructed and belongs to the language of CTL formulas whose syntax is described using the formalism of context-free grammars.

The parser $\mathcal{A}_S$ builds the derivation tree (parsing tree) of the formula into the respective grammar and thus it can determine for any sub-formula of the given formula which is the operation and sub-formulas used in its construction.

For the implementation of our own CTL model checker, we exploited the technological resources provided by the ANTLR system, which enables writing the derivative operations in the native language in which the whole algebraic compiler was generated (Java, C#, etc.).

To achieve the $\mathcal{C}_{MC}$ compiler, we designed the following grammar that generates the L_{ctl} language (grammar of CTL expressions) and we used ANTLR to generate automatically the components $\mathcal{A}_L$ and $\mathcal{A}_S$ on the basis of this grammar.

```
grammar CTL;

options {backtrack=true;}

@header {
   package ctl;
   import java.util.HashMap;
   import org.antlr.runtime.*;
   import java.util.HashSet;
   import java.util.Iterator;
   import org.graphstream.graph.*;
   import org.graphstream.graph.implementations.*;
   }

@lexer::header {package ctl;}

ctlFormula
  :     e1=implExpr 'au' e2=implExpr
  |     e1=implExpr 'eu' e2=implExpr
  |     'ax' e=implExpr
  |     'ex' e=implExpr
  |     'af' e=implExpr
  |     'ef' e=implExpr
  |     'ag' e=implExpr
  |     'eg' e=implExpr
  |     e=implExpr ;

implExpr
  :     e1=orExpr ( '=>' e2=orExpr  )* ;

orExpr
  :     e1=andExpr ( 'or' e2=andExpr )* ;

andExpr
  :     e1=notExpr ( 'and' e2=notExpr  )* ;

notExpr
  :     'not'  e=atomExp
  |     e=atomExp ;


atomExp
  :     '(' f=ctlFormula ')'
  |     AP
```

```
  |     'true'
  |     'false' ;

AP  :
('a'..'z'|'A'..'Z'|'0'..'9'|'!'|'~'|'_'|''|'$'|'%'|'&'|'*'|'?'|'|
         '|'/'|'{'|'}'|'['|']'|'^')+ ;

NEWLINE:'\r'? '\n' ;
WS  :   (' '|'\t')+ {skip();} ;
```

Fig. 2: The grammar of the CTL formulas language

It is noted that the precedence of CTL operators is explicitly encoded by the structure of production rules.

Grammar does not contain the code necessary to implement derivative operations associated with CTL operators.

From the grammar specification is observed that each CTL operator $op \in O_{ctl}$ has associated a production rule.

If for the production rule r we note by $op(r) \in O_{ctl}$ the CTL operator for which was defined the production r, a concise specification of the algebraic compiler $\mathcal{C}_{MC}$ is given by the set $\{\langle r, d_{MC}(op(r)) \rangle | r \in P_G\}$, where $d_{MC}(op(r))$ represents the derivative operation corresponding to the production rule r.

In the ANTLR terminology, for $d_{MC}(op(r))$ we will use the term "*semantic action attached to the production r*".

Evaluation of CTL formulas will be accomplished through implementation of the derivative operations as actions attached to the production rules. Such action can be called *semantic action*, because if by example such action is attached to the production:

$$v \to v_1 v_2 ... v_n$$

the role of respective action is to calculate the semantic value of derivation subtree having root v, ie of CTL subformula that can be built from the derivation of nonterminal v.

In order to implement the compiler $\mathcal{C}_{MC}: L_{ctl} \to L_M$ described in figure 1, we will transform the CTL grammar into an attribute grammar, by augmenting its production rules with semantic actions.

We present in the following a formal description of attribute grammars and the concrete use of such a grammar in implementation of the CTL model checker using ANTLR.

5 Verification of CTL models through attribute grammars

An attribute grammar is a context-free grammar augmented with attributes and semantic rules.

Each symbol (terminal or non-terminal) of an attribute grammar has associated a set (possibly empty) of attributes.

Each attribute has a range of possible values.

Let $G=(N,T_G,P_G,S_0)$ a context free grammar, where N is the set of non-terminal symbols, T_G – the set of terminal symbols, P_G – the set of production rules and S_0 - the start symbol of the grammar .

We denote by $G_A=(N,T_G,P_G,S_0,A,as)$ an attribute grammar built on grammar G by its augmenting with attributes *(A)* and semantic rules (actions) *(as)*. A production $p \in P_G$ is of the form: $X_0 \to X_1 X_2 ... X_{n_p}$ where $n_p \geq 1$, $X_0 \in N$ şi $X_k \in N \cup T_G$ for $1 \leq k \leq n_p$. The derivation tree of a sequence from the language generated by the grammar has the following properties:

- Each leaf node is labelled with a terminal symbol from the set T_G;

- Each inner node *t* corresponds to a production $p \in P_G$, and if the production is of the form $X_0 \to X_1 X_2 ... X_{n_p}$, with the meaning of the symbols described above, then *t* is labelled with the symbol X_0 and has n_p child nodes labelled with $X_1, X_2, ..., X_{n_p}$ from the left to the right.

For any non-terminal symbol $X \in N$ attributes can be divided into:
- *Synthesized attributes* if their values are computed using attributes of child nodes. We denote by *S(X)* the set of synthesized attributes of non-terminal *X*.
- *Inherited attributes* if their values are computed using the values of attributes attached to the parent or siblings nodes. The set of inherited attributes of non-terminal *X* is denoted by *M(X).*

The set of all attributes of non-terminal *X* is denoted by *A(X)* and is equal to:

$$A(X) = S(X) \cup M(X)$$

With these notations, the set of attributes of the grammar G_A is defined as:

$$A = \bigcup_{X \in N} A(X)$$

Considering the set of production rules of the form $P_G = \{p_1, \ldots, p_n\}$, with $n \geq 1$, we denote by $as(j) = \{action_j^{i_1}(), ..., action_j^{i_j}()\}$ the set of semantic actions attached to the production p_j, for each $j \in \{1, \ldots, n\}$. With these notations, the set of semantic actions of the grammar G_A is defined as:

$$as = \bigcup_{1 \leq j \leq n} as(j)$$

The values of all attributes of grammar symbols that appear in the derivation tree of a sequence from the language generated by the grammar are determined by the semantic actions associated with productions of the grammar that are involved in the process of derivation of the respective sequence.

These values are effectively calculated in the process of analysis, through invoking the semantic actions by the parser, generally at the moment of recognition of the next production rule used in the derivation process.

The attribute grammar used in the implementation of the CTL model checker will be denoted by G_A^{CTL} and has the following features:

- $A(X) = S(X), \forall X \in N$ – all attributes are *synthesized attributes*;
- $|A(X)| = 1, \forall X \in N$ – any non-terminal symbol *X* has a single attribute, denoted by $a(X)$.
- $|as(j)| = 1$, for each $j \in \{1, \ldots, n\}$ – each production rule has attached a single semantic action, $as(j) = \{action_j()\}$
- We consider that CTL model is given in the form of a Kripke structure: $M=(S, Rel, P:S \to 2^{AP})$, and the CTL attribute grammar corresponding to the model *M* is: $G_{A,M}^{CTL} = (N, T_G, P_G, S_0, A, as)$. Then $AP \subseteq T_G$ (the atomic propositions are terminal symbols of the attributive grammar). Each atomic proposition $ap \in AP$ has associated a single attribute denoted by $a(ap)$, such as $|A(ap)| = 1 \; \forall ap \in AP \subseteq T_G$. The set of attributes of the CTL grammar is extended to:

$$A = (\bigcup_{X \in N} A(X)) \cup (\bigcup_{ap \in T_G} A(ap)) = (\bigcup_{x \in N} \{a(X)\}) \cup (\bigcup_{ap \in T_G} \{a(ap)\})$$

Also, each atomic proposition $ap \in AP \subseteq T_G$ has associated the semantic action $action_{ap}()$ with the purpose of calculating the value of attribute $a(ap)$. The set of semantic actions of the CTL attribute grammar becomes:

$$as = (\bigcup_{1 \le j \le n} \{action_j()\}) \cup (\bigcup_{ap \in T_G} \{action_{ap}()\})$$

- For any symbol $x \in N \cup AP \cup \{true, false\}$, we denote by $v(x)$ the value of attribute $a(x)$ and we have $v(x) \subseteq S$ (the value of the respective attribute is a subset of the set of states of the model *M*).
- The evaluation of the attributes of the non-terminal symbols is context-dependent: if for rewriting of symbol $X \in N$ was used the production i_X, then the value of attribute of non-terminal X in the respective context is:

$$v(X) = action_{i_X}()$$

- The evaluation of terminal symbols is context-free:
  $v(ap) = action_{ap}() = \{s \in S \mid ap \in P(s)\} \; \forall \; ap \in AP \subseteq T_G$, $v(true) = S$ and $v(false) = \varnothing$.

Because often the model *M* is implicit, we write G_A^{CTL} instead of $G_{A,M}^{CTL}$.

A CTL formula *f* is syntactically correct if and only if there is derivation:

$$ctlFormula \overset{*}{\Rightarrow} f$$

where we suppose that *ctlFormula* = S_0 (the start symbol of the G_A^{CTL} grammar). The derivation tree of the formula *f* has its border composed by terminal symbols that appear in the formula *f*.

We consider that derivation $ctlFormula \overset{*}{\Rightarrow} f$ has a length *k*. Then there is the succession of direct derivations $f_0 \overset{i_1}{\Rightarrow} f_1 \overset{i_2}{\Rightarrow} f_2 \ldots \overset{i_k}{\Rightarrow} f_k$, where f_0 = *ctlFormula*, f_k = *f* and i_l represents the number of production rule involved in the direct derivation *l*, where $l \in \{1, ..., k\}$.

If we note by $action_j()$ the name of the semantic action attached to the production $j \in \{1, ..., n\}$, the parser will call, at the same time with building the tree analysis of the formula *f*, the semantic actions attached to production rules from the derivation:

$$ctlFormula \overset{*}{\Rightarrow} f$$

in the following order:

$$action_{i_1}(),\ action_{i_2}(),\ \ldots,\ action_{i_k}().$$

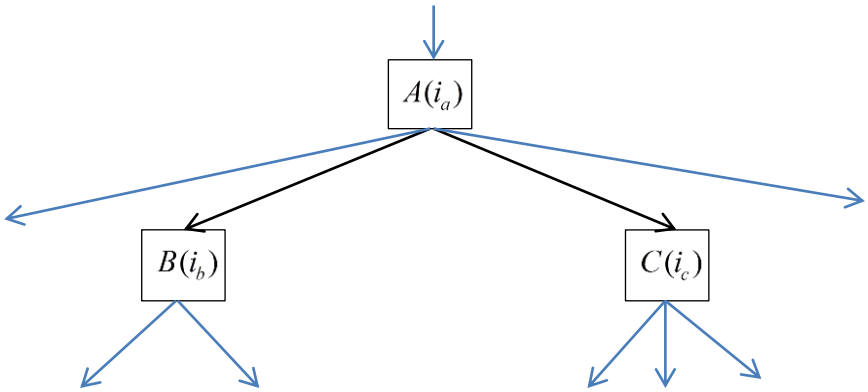

Fig. 3: A syntax subtree for the formula *f*

Assuming that in figure 3 is represented a subtree of the derivation tree built for the formula *f*, where *A*, *B*, *C* are non-terminals of the grammar and i_a, i_b, i_c are the numbers of the production rules used in rewriting of the respective non-terminals, the function $action_{i_a}()$ will contain calls of actions $action_{i_b}()$ and respectively $action_{i_c}()$. An action will return without a call to another action only when it is attached to a terminal symbol. In this case, the respective action carries out the evaluation of that symbol.

In the case of the CTL model $M=(S,Rel,P:S\rightarrow 2^{AP})$, for the symbol *true* will be returned the whole set of states, *S*. For *false* will be returned the empty set $\varnothing$, and for some symbol *ap* from the set *AP* will be returned the set of states $\{s \in S \mid ap \in P(s)\}$.

For a given formula $f \in L_{ctl}$, in the function $\mathcal{C}_{MC}(f)$ the parser $\mathcal{A}_S$ identifies the first production of grammar used in derivation:

$$X_0 \overset{*}{\Rightarrow} f$$

Assuming that production is *r:* $X_0 = t_0 X_1 t_1 X_2 ... X_n t_n$, $\mathcal{C}_{MC}(f)$ will call the derived operation associated with production *r*, $d_{MC}(op(r))$, and will store the result in the meta-variable $\$set$, as follows:

$$\$set = d_{MC}(op(r))\ (\$a_1.set\ ,\ldots\ \$a_n.set)$$

where $\$a_i.set$ are semantic evaluation of non-terminals X_i, $1 \le i \le n$, by which rewriting are obtained the subformulas $f_1,\ldots, f_n$ of *f*. These evaluations are performed recursively, and we have:

$$\$a_i.set = \mathcal{C}_{MC}(f_i),\ 1 \le i \le n$$

If we denote by $G=(N,T_G,P_G,S_0)$ the context-free grammar that generates the CTL formulas language, with production set in the form of $P_G = \{p_1, \ldots, p_n\}$, $n \ge 1$, a concise specification of the compiler $\mathcal{C}_{MC}$ is given by the set of pairs $\{\langle p_i,\ d_{MC}(op(p_i))\rangle \mid 1 \le i \le n\}$, where $d_{MC}(op(p_i))$ is the *derived operation* corresponding to the production p_i and $op(p_i)$ is the CTL operator for which was defined the production p_i, $1 \le i \le n$.

Automatic generation of the CTL model checker from the above specification is accomplished in ANTLR by building an attribute grammar $G_{A,M}^{CTL} = (N,T_G,P_G,S_0,A,as)$ in the meta-description language of ANTLR grammars, with the following properties:

- The grammar productions are those specified in the Section 4.
- Attributes associated to the generators of language L_{ctl} have the following values:

$$v(ap) = action_{ap}() = P'(ap), \forall\, ap \in AP \subseteq T_G$$
$$v(true) = action_{true}() = S$$
$$v(false) = action_{false}() = \varnothing$$

- For production p_i: $X_0 = t_0 X_1 t_1 ... X_n t_n$, the attribute value of non-terminal X_0 is calculated as:

  $$v(X_0) = action_i() = d_{MC}(op(p_i))\ (v(X_1),...,v(X_n)).$$

  Using the notational descriptions of the meta-description language used in specification of the ANTLR attribute gramars, equality becomes:

  $\$set = d_{MC}(op(p_i))\ (\$a_1.set\ ,\ldots\ \$a_n.set)$.

The components $\mathcal{A}_{\mathcal{L}}$, $\mathcal{A}_S$ of the algebraic compiler are automatically generated by ANTLR, using as input the unique file containing the definition of the grammar $G^{CTL}_{A,M}$.

In the ANTLR grammar, the meta-variables of form \$*set* are used to store the attribute values of grammars symbols ($N \cup T_G$). At the time of construction of the derivation tree, for meta-variables that appear in the definition of a semantic action attached to a production used in the derivation process, ANTLR generates code to invoke the semantic actions of productions used in rewriting of non-terminals appearing in the right member of the corresponding production.

The role of semantic actions associated with the production rules of the ANTLR grammar is to calculate and return the attribute values of non-terminal symbols from the left member of respective productions (nonterminals rewrited by these productions).

For example, for the rule *atomExp*:

```
atomExp returns [HashSet set]
:
…
;
```

the generated code has the following form (simplified):

```
public HashSet atomExp () throws RecognitionException
{
      HashSet set = null;    // Return value, referenced in
      ...                    // the definition file of grammar
      return set;            // by $set
}
```

Verification of the given formula *f* involves the building of the derivation $ctlFormula \overset{*}{\Rightarrow} f$ (and hence the corresponding derivation tree) and ends when is returned the attribute value of the start symbol of the grammar $v(ctlFormula) \subseteq S$, which represents the set of all states from *S* which satisfy the formula *f* in the given model M.

Implementing a model checker based on an attributive grammar requires a detailed description of its semantic actions.

In figure 4 is presented the formal definition of the derivative operation $d_{MC}(AG)$.

In our approach, the implementation of the CTL temporal operators is based on two functions $pre_\forall$, $pre_\exists : 2^S \rightarrow 2^S$, defined as:

$$pre_\forall(Z) = \{s \in S \mid succ(s) \subseteq Z\}, \quad \text{respectively}$$
$$pre_\exists(Z) = \{s \in S \mid succ(s) \cap Z \neq \phi\}, \forall Z \subseteq S,$$

where *succ(s)={s'∈S|(s,s')∈Rel}* represents the set of successor states of the state *s* in M.

The functions $pre_\forall, pre_\exists$ are implemented in Java code, specifications being included in the single definition file of the CTL grammar.

The advantage of this solution is that entire algebraic compiler code is generated in a single step without the need for previous pre-processing.

The semantic action of production corresponding to the *AG* operator implements the derivative operation $d_{MC}(AG)$. The function $pre_\forall()$ is dependent on the verified CTL model, so that the model must be accessible to the algebraic compiler when verifying a CTL formula, in the form of internal data structures required by the call $pre_\forall(Z')$.

The argument of the derived operation, $*set1*, represents the calculated image (satisfaction set) of CTL sub-formula to which is applied the AG operator.

The value returned by the semantic action is stored in the variable $*set* to be propagated in the analysis / evaluation process of the CTL formula for which the process of verification was launched:

$$\$set = d_{MC}(AG)\ (\$set_1).$$

```
ctlFormula: $set → 'ag' implExpr: $set1
{  Set Z, Z';
         Z:=∅; Z':= $set1;
         while (¬(Z=Z')) {
            Z:=Z'; Z':=Z'∩ pre∀(Z');
         }
   $set :=Z';
}
```

Fig. 4: The formal definition of the derivative operation $d_{MC}(AG)$

For the CTL operator *AG*, the corresponding *action* included in our ANTLR grammar of CTL language is detailed in figure 5.

```
private HashSet PreAll(HashSet Z) {
  HashSet rez = new HashSet();
  for (Node n1 : model) {
    Iterator<Edge> it =
    n1.getLeavingEdgeIterator();
    HashSet succ = new HashSet();
    while (it.hasNext()) {
       Edge e = it.next();
       Node n2 = e.getTargetNode();
       succ.add(n2.getIndex());
    }
    if (Z.containsAll(succ)) {
       rez.add(n1.getIndex());
    }
  }
  return rez;
}

ctlFormula returns [HashSet set]
@init { }
: 'ag' e=implExpr {
    HashSet rez = new HashSet();
    HashSet rez1 = new HashSet($e.set);
    while (!rez.equals(rez1)) {
       rez.clear();
```

```
        rez.addAll(rez1);
         HashSet tmp = PreAll(rez1);
        rez1.retainAll(tmp);
    }
    $set = rez1;
}
```

Fig. 5: Implementation of the *AG* operator in ANTLR

6 Example

It is noted that for some CTL formula *f*, the syntactic analysis is top-down but evaluation of the formula is at the end of successive function calls using a stack of execution, so it is made in a bottom-up fashion.

The evaluation of the formula *f* is practically accomplished by walking the derivation tree in a bottom-up manner, starting from the leaves and eventually going to the root of the tree.

In each intermediate node, the semantic evaluation of the child nodes are used to compute the semantic value of CTL subformula associated with that intermediate node. The calculation algorithm is determined by the semantic of the CTL operator that appears in the rule of rewriting the non-terminal corresponding to the respective intermediate node.

For the model described in the figure 6:

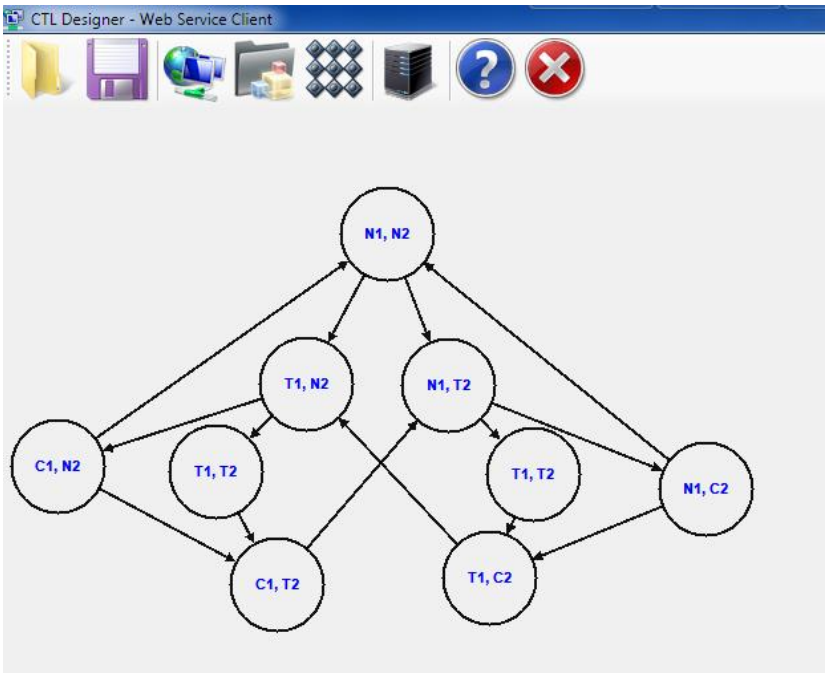

Fig. 6: The CTL model of mutual exclusion of two processes, build with CTL Designer

the parse tree constructed when analyzing the formula *ag (not (C1 and C2))* is presented in figure 7, generated using ANTLRWorks [5]:

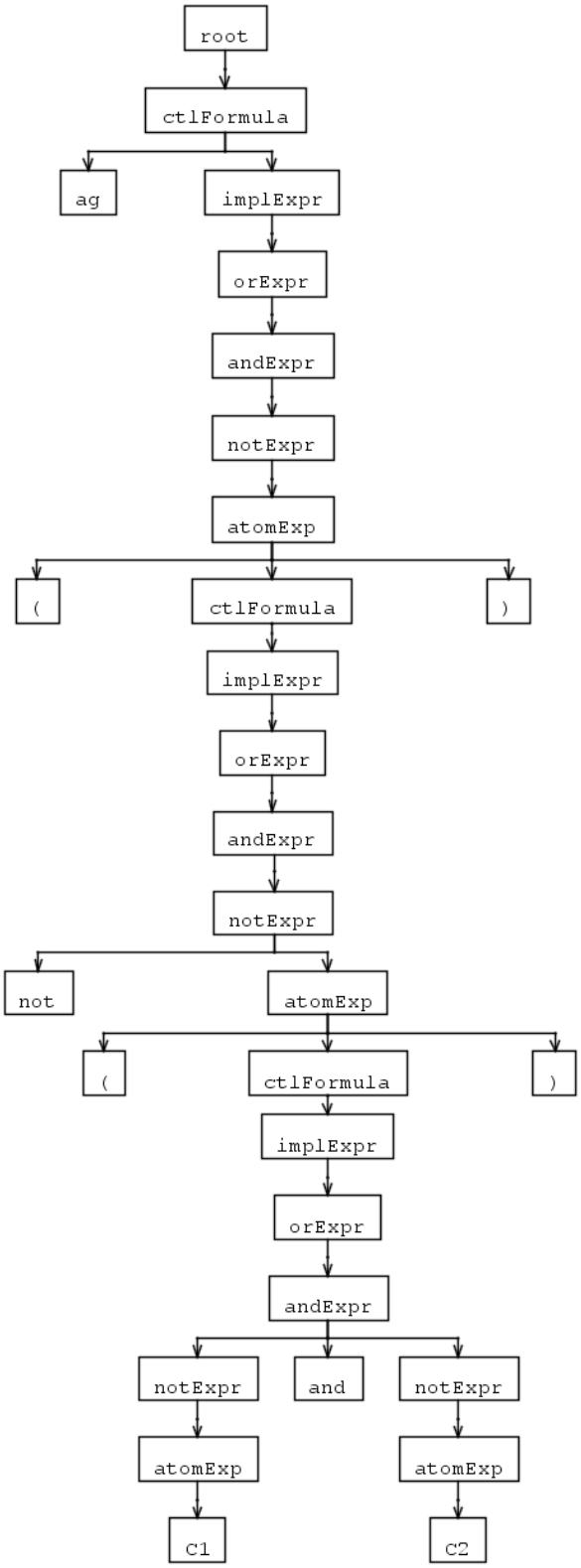

Fig. 7: The parse tree for formula *ag (not (C1 and C2))*

The evaluation of the verified formula is made bottom-up, as we can see from the output of our CTL model checker:

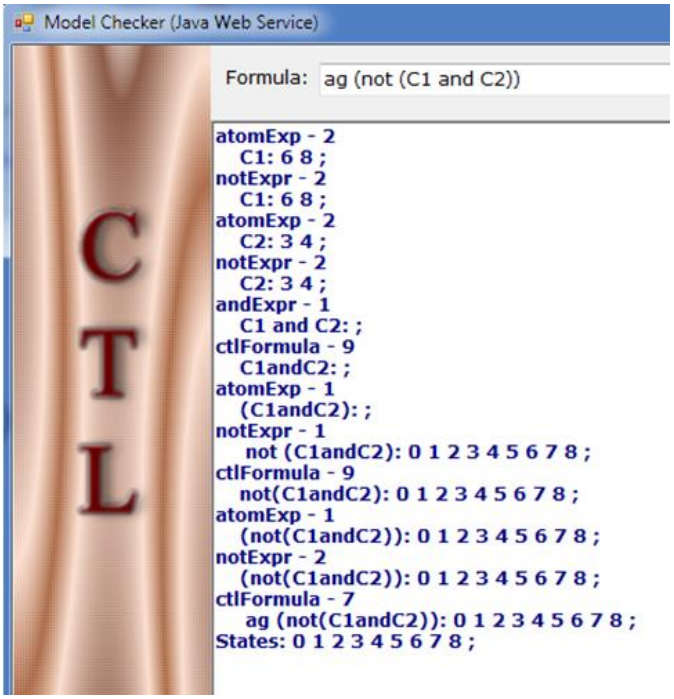

Fig. 8: Verification of formula *ag (not (C1 and C2))* in CTL Designer [6]

7 Conclusions

We enumerate some of the arguments that recommend the utilization of the ANTLR attribute grammars in implementing model checker tools:

- The verified model can be encoded and accessed by classes of objects in the chosen target language (C++, Java, C#, Objective C, Python) directly in the attribute grammar specification file.
- For the implementation of the semantic actions in ANTLR, we can exploit the full power of an advanced programming language (Java, C #, etc.).
- We can specify multiple target languages to generate the model checker tool, and the semantic actions can be implemented by efficient code, taking into account the features of the chosen language.
- The proposed methodology has a generic character since it can be applied to generate model checkers for different temporal logics (CTL, ATL, LTL, etc.).

As future work, we will investigate an alternative approach to generate a CTL model checker using the concept of labelled stratified graph (LSG) [7].

We intend to use the concept of accepted structured path [8] over a stratified graph in order to build a parser for the language of CTL formulas. Also, the inference process [9] developed in a stratified graph can be used to implement a model checker: a CTL formula is transposed in a labelled graph in order to construct inferences based on the given representations, providing as result the set of states where the given CTL formula is satisfied.

References

[1] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Version: 2007-3-20.
[2] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*, Cambridge University Press, pages 1–405, 2000.
[3] E.V. Wyk. *Specificaion Languages in Algebraic Compiler*. CitiSeerX, pages 1–38, 2000.

[4] T. Rus, E. Van Wyk and T. Halverson. *Generating model checkers from algebraic specifications*. Springer, Formal Methods în System Design. Vol. 20, Issue 3, pages 249–284, 2002.
[5] Jean Bovet. *ANTLRWorks: The ANTLR GUI Development Environment*, http://www.antlr.org/works/index.html
[6] L. F. Cacovean, F. Stoica, *WebCheck – ATL/CTL model checker tool*, http://use-it.ro
[7] Daniela Dănciulescu, *Formal Languages Generation in Systems of Knowledge Representation based on Stratified Graphs*, INFORMATICA 2015, vol. 26, no. 3, pp. 407-417, ISSN 0868-4952 (2015)
[8] Daniela Dănciulescu, Mihaela Colhon, *Splitting the structured paths in stratified graphs. Application in Natural Language Generation*, Analele Ştiinţifice ale Universităţii Ovidius Constanţa, Seria Matematică, vol. 22, no. 2, pp.59-69, ISSN: 1224-1784 (2014)
[9] Daniela Dănciulescu, *Systems Of Knowledge Representation Based On Stratified Graphs And Their Inference Process*, 9th International Conference of Applied Mathematics, Abstracts and Pre-Proceedings, Baia Mare 25-28 September (2013)

Laura Florentina STOICA
Faculty of Science
"Lucian Blaga" University
Department of Mathematics and Informatics
5-7 Dr. Ratiu Street, Sibiu
ROMANIA
E-mail: laura.cacovean@ulbsibiu.ro

Florin STOICA
Faculty of Science
"Lucian Blaga" University
Department of Mathematics and Informatics
5-7 Dr. Ratiu Street, Sibiu
ROMANIA
E-mail: florin.stoica@ulbsibiu.ro

Mircea Florian BOIAN
Faculty of Mathematics and Computer Science
"Babes Bolyai" University
1 M. Kogalniceanu Street, Cluj-Napoca
ROMANIA
E-mail: florin@cs.ubbcluj.ro