A New Model Checking Tool

LAURA F. CACOVEAN, FLORIN STOICA, DANA SIMIAN

Department of Computer Science
“Lucian Blaga” University of Sibiu, Faculty of Sciences
Str. Dr. Ion Ratiu 5-7, 550012, Sibiu
ROMANIA

laura.cacovean@ulbsibiu.ro, florin.stoica@ulbsibiu.ro, dana.simian@ulbsibiu.ro

Abstract: - In this paper we present a new CTL model checking tool used to prove whether a CTL model represented as a directed graph satisfies a set of specifications given in form of one or more temporal logic formulas. Our tool is implemented in client-server paradigm: CTL Designer, the client tool, allows an interactive construction of the Kripke model as a directed graph and the CTL Checker, the core of our tool, represents the server part and is published as Web service. The CTL Checker includes an algebraic compiler implemented with ANTLR (Another Tool for Language Recognition) support. The main function of the Web service is to parse a given formula, find the set of nodes in which the formula is satisfied and return result to the user. As test case for our tool, we choose a CTL Model for Login Page Controller. The model will check if logging is allowed or not depending on input data.

Key-Words: - CTL model checking, Web services, ANTLR, algebraic compiler

1 Introduction

Model checking is a technology often used for the automated system verification. The model checking algorithms are currently used as verification techniques implemented in varied programming environments. The verified system can be a physical system or a real-time concurrent program. The behavior of checked system is described by the Kripke model. The Kripke models are based on the states and use the SMV (Symbolic Model Verifier) technique. The SMV model checking takes as input the model and formula then check whether or not the formula is satisfied or not by the model. Until now were developed tools for model checking which are using temporal logics like ALLOY, BANDERA, CADENCE SMV, GEAR, MCMAS, NuSMV. These instruments have support for the CTL (Computation Tree Logic) specification properties, expressed as formulas of propositional temporal logic.

Our CTL model checking tool has a great facility because being a Web service can be accessed by any Internet user. We also provide a .NET GUI client which has capability of interactive graphical specification of the CTL model.

Because the algebraic operators of a process are easily expressed in scenarios of a system, and the state variables are very suitable for the specification of properties, a CTL model appear to be sufficient to handle most common properties of analyzed system.

The remainder of this paper is organized as follows. In section 2 we present a short definition of a CTL model checker. In section 3 is presented the implementation of an algebraic compiler used by our tool to verify CTL formulas for given models. Invocation of the compiler will be accomplished through a Web service described in section 4. CTL Designer, the client component of our tool is presented in section 5. Section 6 deals with a test case for the new model checking tool. Conclusions are presented in section 7.

2 CTL model checker

A model checking tool can be used to verify if a given system satisfies a temporal logic formula.

A CTL model is defined as a directed graph. A Kripke model M over AP is a triple $M=(S, Rel, P:S→2^{AP})$ where $S$ is a finite set of states, $Rel⊂S×S$ is a transition relation, and $P:S→2^{AP}$ is a function that assigns each state with a set of atomic propositions, denoted by AP.

CTL model checker is branching-time logic, meaning that its formulas are interpreted over all paths beginning in a given state of the Kripke structure.

For each state from graph $M$ there is a successor and a path composed by a sequence of some states.
Details about formal definitions for syntax and semantics of a CTL model checker can be found in paper [4].

A syntax definition of a CTL model checker [4]

A CTL has the following syntax given in BNF (Backus-Naur Form): $f ::= \top | \bot | [\neg f] | f_1 \wedge f_2 | f_1 \vee f_2 | f_1 \rightarrow f_2 | AX f_1 | EX f_1 | AG f_1 | EG f_1 | AF f_1 | EF f_1 | A[f_1],U,f_2] | E[f_1],U,f_2]$ where $\forall p \in \mathcal{AP}$.

B. Semantics definition of a CTL model checker is detailed in [4]

Let $M=(S, R, P: S \rightarrow 2^{\mathcal{AP}})$ be a Kripke model for CTL. Given a state $s$ in $S$, in [2] is defined if a CTL formula $f$ holds in state $s$. This is denoted by $(M, s) \models f$. The satisfaction relation $\models$ is defined by structural induction on fourteen CTL formulas [4].

The model checker takes as input the model and the formula and then verifies if the formula is satisfied by the model. The query can be expressed as: $(M, s) \models f^*$ where $M$ will be the model analyzed, $s$ will be a state from the model, and $f$ will be the verified formula, which is satisfied or not by the model $M$.

The base idea of the verification algorithm is to split the CTL formula $f$ into valid sub-formulas, and using a structural induction will be created a parser tree labeled with these sub-formulas. Depending on the meaning of connectors and the truth value of each sub-formula, will be deducted the satisfaction of formula from each state using the parser tree created.

3 Implementation of a model checker in ANTLR

The flexibility of algebraic methodology shows us how the CTL model checkers for various temporal logics can be generated from the algebraic specification. In paper [2] we showed in detail how this algebraic approach can be used to the specification of a CTL model checker.

The CTL language is defined as a $\Sigma$ - language [2]. The operator scheme $\Sigma_{ctl}$ is defined as a triple $(\mathcal{O}_{ctl}, \mathcal{S}_{ctl}, \sigma_{ctl})$ where sets $\mathcal{S}_{ctl}=\{F\}$ are representations of the formula $f$, $\mathcal{O}_{ctl} = \{\top, \bot, \neg, \land, \lor, \rightarrow, AX, EX, AU, EU, EF, AF, EG, AG\}$ is the set of operators, and the $\sigma_{ctl}: \mathcal{O}_{ctl} \rightarrow \mathcal{S}_{ctl} \times \mathcal{S}_{ctl}$ is a function which defines the signature of the operators [2]. The CTL model checker can be defined as the $\Sigma_{ctl}$ - language given in the form $L_{ctl} = \langle \mathcal{Sem}_{ctl}, \mathcal{Syn}_{ctl}, L_{ctl}, \mathcal{Sem}_{ctl} \rightarrow \mathcal{Syn}_{ctl} \rangle$ where $\mathcal{Syn}_{ctl}$ is the word algebra of the operator scheme $\Sigma_{ctl}$ generated by the operations from $\mathcal{O}_{ctl}$ and a finite set of variables, representing atomic propositions, denoted by $\mathcal{AP}$. $\mathcal{Sem}_{ctl}$ represents CTL semantic algebra defined over the set of CTL formulas which are satisfied by the CTL model $M$. $L_{ctl}$ is a mapping which associates the set of satisfied formulas from $\mathcal{Sem}_{ctl}$ to CTL expressions from $\mathcal{Sin}_{ctl}$ which satisfy these formulas.

From a formal point of view, implementation of a CTL model checker will be equated with implementation of an algebraic compiler.

The definition of sigma algebras, sigma languages and also the definition of sigma heterogenic homomorphism can be found in detail in papers [3], [7].

In paper [2] it is mentioned that an algebraic compiler can be defined using $\Sigma$ - algebras and $\Sigma$ - heterogenic homomorphism as $C:L_{ctl} \rightarrow L_{s}$, where $L_{s}$ is the source language and $L_{ctl}$ is the target language. The source language $L_{s}$ is CTL, and the target language $L_{s}$ is a language which describes the set of nodes from the model $M$ where the formula $f$ is satisfied.

The algebraic compiler $C$ translates formula $f$ of the CTL model to set of nodes $S'$ over which formula $f$ is satisfied. That is, $C(f)=S'$ where $S' = \{s \in S \mid (M, s) \models f\}$.

The implementation of the algebraic compiler $C$ is made in two steps. First, we need a syntactic parser to verify the syntactic correctness of a formula $f$. Then, we should deal with the semantics of the CTL language ($\mathcal{Sem}_{ctl}$), respectively with the implementation of the operators from set $\mathcal{O}_{ctl}$.

Writing a translator for certain language is difficult to be achieved requiring time and a considerable effort. Currently there are specialized tools which generate most of necessary code beginning from a specification grammar of the source language.

For implementation of the algebraic compiler we choose the ANTLR (Another Tool for Language Recognition). ANTLR [5] is a compiler generator which takes as input a grammar - an exact description of the source language, and generates a recognizer for the language defined by the grammar. ANTLR support the EBNF (Extended BNF) notation, useful for specification of operations that requires the use of recursion.

In order to translate a formula $f$ of a CTL model to the set of nodes $S'$ over which formula $f$ is satisfied, is necessary the attachment of actions to grammatical constructions within specification grammar of CTL.

The actions are written in target language of the generated parser (in our case, Java). These actions are incorporated in source code of the parser and are
activated whenever the parser recognizes a valid syntactic construction in the translated CTL formula. In case of the algebraic compiler $C$, the actions define the semantics of the CTL model checker, i.e., the implementation of the $O_{ct}$ operators.

The model checker generated by ANTLR from our specification grammar of CTL, takes as input the model $M$ (where are defined the sets $S$, $Rel$, $P$) and formula $f$, and provides as output the set $S'\{s\in S \mid (M,s) \not\models f\}$ – the set of states where the formula $f$ is satisfied.

Given a CTL formula $t$, $d_{ct}(t)$ denotes the set of states in which the formula $t$ holds. Each operation $o_{ct}$ from algebra $Sem_{ct}$ is associated with the expression $d(o_{ct}(t))$ representing the set of states from the algebra $Syn_{ct}$ in which the formula $o_{ct}(t)$ is satisfied [2]. In case of the AG operator we are using the following notation: $d(AG(t)) = d(All_{global}(t))$.

For the formal specification of the AG operator given in figure 1, the corresponding action included in the ANTLR grammar of CTL language is detailed in figure 2.

```java
Z := ∅; Z' := d_{ct}(t);
while (Z ≠ Z') do
    Z := Z';
    Z' := Z' \cap \{s \in S \mid succ(s) \subseteq Z'\};
endwhile
d_{ct}(All_{global}(t)) := Z';
```

Fig 1. Formal definition of the set expression $d_{ct}(AG(t))$

//CtlFormula—to"ag" formula
ctlFormula returns [HashSet set] :
    'ag' f=formula
{
    HashSet rez = new HashSet(); // Z := ∅;
    HashSet rez1 = new HashSet(S.set);
    // Z' := d_{ct}(O);
    while (!rez.equals(rez1)) // while Z ≠ Z' do
    {
        rez.clear(); //rez.addAll(rez1); // Z := Z';
        HashSet tmp = new HashSet();
        boolean include;
        for (int i=0; i<MAXARY; i++)
        {
            include = true;
            for (int j=0; j<MAXARY; j++)
            if (rel[i][j] == 1) // ∃ succ(s), s ∈ S
                if (!rez1.contains(new Integer(j)))
                    include = false;
                if (include)
                    tmp.add(new Integer(i)); //succ(s) ⊆ Z'
        } // end for
        rez1.retainAll(tmp);
    }
}
```

Fig 2. Implementation of the AG formula in ANTLR

For verification of formula $f=ag(not((Btn and Err)))$ we can use the ANTLR debugging facility [5] to visualize the abstract syntactic tree AST, presented in figure 4.

In figure 3 is represented the algebraic compiler implementation process, based on our specification grammar of CTL language.

```
/*BNZ grammar together with Java source code */
```

Fig 3. Algebraic compiler implementation

```
/*BNZ grammar together with Java source code */
```

Fig 4. AST tree for ag(not((Btn and Err))) formula in ANTLRWorks
4 Publishing the CTL model checker as a Web service

Web Services, as a distributed application technology, simplifies interoperability between heterogeneous distributed systems. Clients can access Web services regardless of the platform or operating system upon which the service or the client is implemented. In addition to interoperability, Web service clients can use standardized approaches to access services through firewalls. Such access extends the capabilities of clients. The transport protocol used by Web Services enables clients to operate with systems through firewalls [6].

In order to make available our implementation of algebraic compiler as a reusable component of a CTL model checking tool, we published it as a Web service.

The architecture of the Web service implementation is represented in figure 5.

The Web service will receive from a client the XML representation of a CTL model M and a CTL formula f.

The original form of the CTL model M is passed then to the algebraic compiler C generated by ANTLR using our CTL extended grammar. The compiler will return as result $C(f) = \{s \in S | (M,s) \models f\}$, the set of nodes in which the formula is satisfied.

Obviously, the formula f may contain syntactical errors.

In order to notify the client about these possible errors, we must override the default behavior of the ANTLR error-handling.

We install our error-handling in lexer:

```java
@lexer::members {
    @Override
    public void reportError(RecognitionException re) {
        throw new RuntimeException("Lexical error!
           Position: " + re.line + ":" + re.charPositionInLine + " bad character: "+(char)re.c + "");
    }
}
```

and also in parser:

```java
@members {
    @Override
    public void reportError(RecognitionException re) {
        throw new RuntimeException("Syntactical error!");
    }
}
```

and finally instruct ANTLR to throw the error, allowing the Web service to send it to the client:

```java
@rulecatch {
    catch (RecognitionException err) {
        throw err;
    }
}
```

5 The C# client

The model checking tool is based on a C# GUI client who allows interactive graphical development of the CTL models.

All facilities are accessible through a right-click contextual menu: adding nodes, labeling nodes, deleting nodes, adding arcs, display nodes numbers, etc.

The model is sent as a XML document to the Web service, together with the formula to be...
verified. The response from server is displayed in a separate window, as we will see in the following section.

6 CTL Model for Login Page Controller

In this section we consider as testing example a CTL model presented in [1], and we use our tool to verify a CTL formula. Our model refers to a login page with user name and password. The model will check if logging is allowed or not depending on input data.

The Kripke model has five states and the propositional variables are from the set \( AP = \{ \text{Usr}, \text{Psw}, \text{Btn}, \text{Err}, \text{not}\_\text{Usr}, \text{not}\_\text{Psw}, \text{not}\_\text{Btn}, \text{not}\_\text{Err} \} \). \text{Usr} represents the fill in the user name, \text{Psw} corresponds to fill in the password, \text{Btn} is the push button event for Sing-in or Sign-out page and \text{Err} point out an error to login.

The Kripke model is shown in figure 7:

**Fig 7. The CTL structure of “Login Page Controller”**

The formal definition of the Kripke structure for the Login Page Controller is given by: \( M = (S, \text{Rel}, P) \), where \( S = \{ s_0, s_1, s_2, s_3, s_4 \} \), \( \text{Rel} = \{ (s_0, s_1), (s_1, s_0), (s_1, s_2), (s_2, s_1), (s_2, s_3), (s_3, s_2), (s_3, s_4), (s_4, s_3), (s_4, s_0) \} \), \( AP = \{ \text{Usr}, \text{Psw}, \text{Btn}, \text{Err}, \text{not}\_\text{Usr}, \text{not}\_\text{Psw}, \text{not}\_\text{Btn}, \text{not}\_\text{Err} \} \), \( P \) assigns state \( s_0 \) in \( M \) with \{ \text{not}\_\text{Usr}, \text{not}\_\text{Psw}, \text{not}\_\text{Btn}, \text{not}\_\text{Err} \}, the state \( s_1 \) in \( M \) with \{ \text{Usr}, \text{Psw}, \text{not}\_\text{Btn}, \text{not}\_\text{Err} \}, the state \( s_3 \) in \( M \) with \{ \text{Usr}, \text{Psw}, \text{Btn}, \text{not}\_\text{Err} \}, the state \( s_4 \) in \( M \) with \{ \text{not}\_\text{Usr}, \text{not}\_\text{Psw}, \text{not}\_\text{Btn}, \text{Err} \}.

For the model presented in figure 7 an important task is to verify formula \( ag \ (\text{not} \ (\text{Btn} \text{ and } \text{Err})) \) that means to see if there exists a node which did not have in the same time the push sign-in button and have the error. The interpretation and verification of this formula presupposes these: We have the atomic proposition \( \text{Btn} \) in state \( \{ s_3 \} \) and the atomic proposition \( \text{Err} \) in state \( \{ s_4 \} \). The \( (\text{Btn} \text{ and } \text{Err}) \) is empty sets because any state from model does not contain both atomic propositions. Hence \( \text{not} \ (\text{Btn} \text{ and } \text{Err}) \) formula is true in all states \( \{ s_0, \ldots, s_4 \} \) and also the \( ag \ (\text{not} \ (\text{Btn} \text{ and } \text{Err})) \) is a formula satisfied by the proposed model.

Verification of the formula with our CTL model checking tool allows us to conclude that formula is satisfied in all states of the Kripke model presented in figure 7. The states of model are numbered from 0 to 4. This information is available selecting Details from the contextual menu of the CTL Designer.

**Fig 8. Invoking the algebraic compiler (Web service) from the CTL Designer (client)**

7 Conclusions

In this article we built a CTL model checking tool, based on robust technologies (Java, .NET) and well-known standards (XML, SOAP, HTTP). As a great facility we mention the capability of interactive graphical specification of the CTL model, using the client tool (CTL Designer).

The CTL algebraic compiler, implemented as a Web service and based on Java code generated by
ANTLR using an original CTL grammar, provides error-handling for eventual lexical/syntax errors in formula to be translated.

References:


