Building a new CTL model checker using Web Services

Fiorin Stoica
Department of Computer Science
Faculty of Sciences, “Lucian Blaga” University of Sibiu
Sibiu, Romania
florin.stoica@ulbsibiu.ro

Laura Florentina Stoica
Department of Computer Science
Faculty of Sciences, “Lucian Blaga” University of Sibiu
Sibiu, Romania
laura.cacovean@ulbsibiu.ro

Abstract—This Computation Tree Logic (CTL) is widely used to capture compositions of reactive systems. Model checking is particularly well-suited for the automated verification of finite-state systems, both for software and for hardware. A CTL model checker tool allows designers to automatically verify that systems satisfy specifications expressed in the language of CTL logic. In this paper we present a new CTL model checker implemented in client-server paradigm. CTL Designer, the client tool, allows an interactive construction of the CTL models as state-transition graphs. Java and C# APIs are provided for programmatic construction of large models. The server part of our tool embeds the core of the CTL model checker and is published as a Web service. The performance evaluation in terms as speed and scalability was accomplished implementing an algorithm to find a winning strategy in the Tic-Tac-Toe game.

Keywords—Model Checking, CTL, Web services

I. INTRODUCTION

Testing and simulation can give us only confidence in the implementation of a software system, but cannot prove that all bugs have been found. However testing is neither exhaustive nor very effective for software, especially concurrent software, which is much more complex than sequential software. Thus, there has been a tremendous push for efficient algorithms and techniques that allow one to prove that a program satisfies certain properties. The process of stating and proving properties about programs is known as program verification. Verification of a software system involves checking whether the system in question behaves as it was designed to behave. Design validation involves checking whether a system design satisfies the system requirements. Both of these tasks, system verification and design validation can be accomplished thoroughly and reliably using model-based formal methods, such as model checking [17]. Model checking is the formal process through which a given specification representing a desired behavioral property is verified to hold for a given system (the model). Main concern of formal methods in general, and model checking in particular, is helping to design correct systems [1].

In the following we consider that model is a directed graph where the nodes represent the states of the system and the edges represents the state transitions, more specifically a Kripke structure.

The state explosion problem and reducing the time required to verify models are widely agreed to be the most formidable challenges facing the application of model checking to large and complex real-world systems. This naturally raises interest in using parallelism to improve the performance of many formal model checkers.

A. Related work

Two most common methods of performing model checking are explicit enumeration of states of the model and respectively the use of symbolic methods. Some examples of well-known explicit-state parallel model checkers are DiViNe and PSPIN. DiVinE [1], [2] is a distributed model checker for explicit state LTL (Linear Temporal Logic) model checking and is able to handle large systems consisting of as many as 419 million states, as stated in [3]. PSPIN has also been used for performing distributed model checking with the capability of handling up to around 2.8 million states [13]. The basic idea behind symbolic model checking is to use a more efficient “symbolic” representation for the Kripke structure being checked and for sets of states of the Kripke structure. Since the sizes of these representations is typically the limiting factor in applying model checking, an efficient representation can potentially allow much larger structures to be checked. Symbolic model checkers, such as CadenceSMV [15] and NuSMV [5] analyze the state space symbolically using binary decision diagrams (BDDs), which were introduced in [4]. The binary decision diagram is a data structure for representing Boolean functions. In contrast with explicit-state model checking, states in symbolic model checking are represented implicitly, as a solution to a logical equation. This approach saves space in memory since syntactically small equations can represent comparatively large sets of states [17]. A symbolic model checker represents the Kripke structure itself symbolically using BDDs to represent transition relations by Boolean expressions and perform all calculations directly using these Boolean expressions, rather than using the Kripke structure explicitly.

B. Comparing Symbolic and Explicit Model Checking

In symbolic model checking, the space used to represent a state is limited by the internal data structures. In explicit model checking, the state size is not strictly limited, but is related with state space size in the total memory consumption. In [8] is
presented a comparison between RULEBASE, a symbolic model checker developed at IBM Haifa Research Laboratory and the explicit LTL model checker SPIN [10]. The software verified was a distributed storage subsystem software application. The state space size handled by SPIN was $10^8$ in a 3-process model. Using symbolic model checking, RULEBASE keeps a compressed representation of the state space and thus was able to manage $10^{10}$ states. On the other hand, because of the limit on state size, RULEBASE could not represent a state large enough to include the information needed for more than 2-process configuration [8]. Most hardware designs are based on a clocked-approach and thus are synchronous. For these systems, the symbolic model checking approach is more appropriate [14]. On the other hand, for nondeterministic, high-level models of hardware protocols, it has previously been argued that explicit model checking is better than symbolic model checking [11]; this is because the communication mechanisms inherent in protocols tend to cause the BDDs in symbolic model checking to blow up [3]. In their basic form, symbolic approaches tend to perform poorly on asynchronous models where concurrent interleaving are the main source of explosion, and explicit-state model-checkers have been the preferred approach for such models [3]. A detailed experimental comparison between performance of explicit-state model checkers and symbolic model checkers can be found in [19]. The study follows an automata-theoretic approach in program verification, originally proposed by Vardi and Wolper [20]. Given a program $P$ and a property $\phi$, the task is to check whether the program satisfies the property $\phi$. If the program $P$ is viewed as a finite-state generator of words, and the specification $\phi$ as a finite-state acceptor, the model-checking problem is reduced to an automata-theoretic question: whether the automaton $A_\phi \cap A_\psi$ is empty. A non-deterministic automaton $A$ defined over a nonempty finite alphabet $\Sigma$ is said to be universal if it accepts $\Sigma^*$. The universality problem is to check if $A$ is universal. If $A_\psi$ is a universal automaton, the model checking problem is reduced to check whether $A_\phi$ is also universal. The study presented in [19] uses two approaches, explicit and symbolic, for solving the universality problem. In order to present a full comparison between the symbolic and the explicit algorithms, was performed a scaling comparison of Cadence SMV, NuSMV as symbolic BDD-based model checkers and respectively SPIN. The direct comparison of the three model checkers shows that the explicit one (SPIN) is much faster than the symbolic ones. In conclusion, experimental results show that the explicit approach scales better than the symbolic one, which was rather surprising but confirms similar statements from [3], [11].

C. Motivation and objective

The broad goal of our research was to develop a reliable, easy to maintain, scalable model checker tool to improve applicability of CTL (Computation Tree Logic) model checking in design of general-purpose computer software.

Concurrent software is asynchronous as the different components might be running on different processors or be interleaved by the scheduler of the operating system. Taking into account the above considerations, in our tool we are using an explicit-state model technique. In order to scaling its verification ability to handle real-world applications and to address the state explosion problem, our tool is based on an efficient data structure for internal representation of the model to be verified [9]. An orthogonal approach to increase the capacity of an explicit-state model checker tool is to exploit the memory and computational resources of multiple computers in a distributed computing environment [3]. Following this idea, our tool is based on Web Services technology to address the time constraints in verification of large models.

The paper is organized as follows. In section 2 are introduced concepts related to the CTL model checking problem: first is defined the CTL model then are presented the CTL syntax and semantics. In section 3 is described the implementation of the CTL model checker in ANTLR. The architecture of the Web service which embeds the server part of our tool is presented in section 4. The performance evaluation of our tool is presented in section 5. Conclusions are presented in section 6.

II. CTL Model Checking

A temporal logic is essentially an ordinary predicate or propositional logic with the addition of modal operators for describing how the interpretation of symbols changes over time [11]. The CTL logic is interpreted over Kripke structures. 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. The problem of model checking is to verify if a finite-state abstraction of a reactive system satisfies a temporal-logic specification expressed in the language of CTL logic. A CTL model checker is a tool which verifies if a given system satisfies a given CTL logic formula, using an algorithm that determines the states of a model that satisfy the formula.

A. The CTL model

A model is defined as a Kripke structure $M=(S, R, P: S \rightarrow 2^P)$ 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 labeling 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 $succ(s) = \{s' \in S \mid (s, s') \in Rel\}$ is used to denote the set of successors of $s$. From definition of $Rel$, each state from $S$ must have at least one successor, i.e., $\forall s \in S, succ(s) \neq \emptyset$. 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 labeling function $P$ maps for each state $s \in S$ the set $P(s)$ of all atomic propositions from $AP$ that are valid in $s$ [12]. A CTL formula has the following syntax:

$\phi :: true | false | ap[(\neg \phi)] | \phi \lor \psi | \phi \land \psi | \phi \rightarrow \psi | AX \phi | EX \phi | AG \phi | EG \phi | AF \phi | EF \phi | AFU \phi | EU \phi | Vap \in AP$.

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 state), $F$ (in future), $G$ (always) and $U$ (until). A CTL formula encodes properties that can occur
along a particular temporal path as well as to the set of all possible paths. The set of all paths through a Kripke structure is assumed to correspond to the set of all possible computations of a system.

B. Fixed-point characterization of CTL

Let \( M=(S, Rel, P:S \rightarrow 2^S) \) be an arbitrary finite Kripke structure. Given a state \( s \in S \), is defined a satisfaction relation \( (M, s) \models \phi \) to specify that formula \( \phi \) holds in \( s \). We denote by \( [\phi]_M = \{ s \in S \mid (M, s) \models \phi \} \) the set of all states from \( S \) which satisfy the formula \( \phi \). \([\phi]_M \) is called the denotation of \( \phi \) in model \( M \). Because often \( M \) is implicit, we write \( [\phi] \) rather than \( [\phi]_M \). Thus \( (M, s) \models \phi \Leftrightarrow s \in [\phi] \). In the following \( \mathcal{Z} \) is used to denote the power set of the set \( S \). Let \( f : 2^S \rightarrow 2^S \) be a set valued function and \( Z \subseteq S \) a subset of \( S \). \( Z \) is called a fixed point of \( f \) if \( f(Z) = Z \). \( Z \) is called the least fixed point (LFP) of \( f \) if \( f(Z) = Z \) and \( \forall U \subseteq S \), \( f(U) = U \Rightarrow Z \subseteq U \). \( Z \) is called the greatest fixed point (GFP) of \( f \) if \( f(Z) = Z \) and \( \forall U \subseteq S \), \( f(U) = U \Rightarrow U \subseteq Z \).

The universal and existential pre-image functions \( \text{pre}_\exists, \text{pre}_\forall : 2^S \rightarrow 2^S \), are defined by:

\[
\text{pre}_\exists(X) = \{ s \in S \mid \exists \phi \text{ such that } s \models (\phi \land X) \} \\
\text{pre}_\forall(X) = \{ s \in S \mid \forall \phi \text{ such that } s \models (\phi \land X) \}
\]

For a CTL formula \( \phi \), the model checker will compute \([\phi]\) recursively, using the rules described in the following table, where LFP and GFP represent the least fixed point and respectively the greatest fixed point of the specified functions.

The algorithm for effective computation of \([\phi]\) is presented in the following section.

### Table I. Recursively Computation of Denotation \([\phi]\) of CTL Formula \( \phi \).  

<table>
<thead>
<tr>
<th>Formula ( \phi )</th>
<th>Computation of ([\phi])</th>
</tr>
</thead>
<tbody>
<tr>
<td>( ap )</td>
<td>( { s \in S \mid ap \in P(s) } )</td>
</tr>
<tr>
<td>( true (false) )</td>
<td>( S (\emptyset) )</td>
</tr>
<tr>
<td>( \neg \psi )</td>
<td>( S \setminus [\psi] )</td>
</tr>
<tr>
<td>( \psi \land \phi )</td>
<td>( [\phi] \cap [\psi] )</td>
</tr>
<tr>
<td>( \psi \lor \phi )</td>
<td>( [\phi] \cup [\psi] )</td>
</tr>
<tr>
<td>( AX \psi ), ( EX \psi )</td>
<td>( \text{pre}<em>{\forall}([\psi]), \text{resp. } \text{pre}</em>{\exists}([\psi]) )</td>
</tr>
<tr>
<td>( \psi \land AX \phi ), ( \psi \land EX \phi )</td>
<td>( [\phi] \cap \text{pre}_{\forall}(X) )</td>
</tr>
<tr>
<td>( \psi \lor AX \phi ), ( \psi \lor EX \phi )</td>
<td>( [\phi] \cup \text{pre}_{\exists}(X) )</td>
</tr>
<tr>
<td>( AG \phi ), ( EF \phi )</td>
<td>( \text{GFP of } [\phi] = [\phi] \cap \text{pre}_{\forall}(\phi) )</td>
</tr>
<tr>
<td>( AF \phi )</td>
<td>( \text{LFP of } [\phi] = [\phi] \cup \text{pre}_{\exists}(\phi) )</td>
</tr>
<tr>
<td>( EF \phi )</td>
<td>( \text{LFP of } [\phi] = [\phi] \cup \text{pre}_{\exists}(\phi) )</td>
</tr>
</tbody>
</table>

III. IMPLEMENTATION OF A MODEL CHECKER IN ANTLR

The CTL model checker is provided as a compiler \( \mathcal{E} : L_{s} \rightarrow L_{t} \), where \( L_{s} \) is the source language and \( L_{t} \) is the target language. The source language \( L_{s} \) is the language describing the CTL formulas and the target language \( L_{t} \) is a language which describes the set of nodes from the model \( M \) where the corresponding CTL formulas are satisfied. The compiler \( \mathcal{E} \) translates a formula \( \phi \) of the CTL model to the set of nodes \([\phi]\) over which formula \( \phi \) is satisfied. That is, \( \mathcal{E} (\phi) = [\phi] \) where \([\phi] = \{ s \in S \mid (M, s) \models \phi \} \). The implementation of the compiler \( \mathcal{E} \) is made in two steps. First, we need a syntactic parser to verify the syntactic correctness of a given formula \( \phi \). Then, we should deal with the semantics of the CTL language, respectively with the implementation of the CTL operators presented in Table I.

Writing a translator for certain language is difficult to be achieved, requiring time and a considerable effort [18]. For implementation of the algebraic compiler we choose the ANTLR (Another Tool for Language Recognition). ANTLR [16] 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. In order to translate a formula \( \phi \) of a CTL model to the set of nodes \([\phi]\) over which formula \( \phi \) is satisfied, is necessary to attach 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 our compiler \( \mathcal{E} \), the actions define the semantics of the CTL model checker, i.e., the implementation of the CTL operators. The model checker generated by ANTLR from our specification grammar of CTL takes as input the model \( M \) (where are defined the sets \( S, Rel \), and \( P \)) and a formula \( \phi \), and provides as output the denotation of \( \phi \) – the set of states where the formula \( \phi \) is satisfied, using an algorithm which handles temporal operators \( AX, EX \) by computing pre-images using expressions given in (1) and implements the temporal operators \( AG, AF, EF, AU, EU \) by applying rules described in Table I, until a fixpoint is reached. The algorithm for computing \([EF \phi]\) is presented in Fig. 1 [6]. For the formal specification of the EF operator given in Fig. 1, the corresponding action included in our ANTLR grammar of CTL language is detailed in Fig. 2, where \( PreExist() \) represents the implementation of function \( \text{pre}_{\exists}(X) \) described in equation (1).

\[
Z := S; \\
Z' := [\phi]; \\
\text{while } (2Z \neq Z') \text{ do} \\
Z := Z'; \\
Z' := Z' \cup \text{pre}(Z'); \\
\text{endwhile} \\
[EF \phi] := Z';
\]

Fig. 1. Formal definition of the set expression \([EF \phi]\).

ctlFormula returns [HashSet set] 

```java
@init { 
  'ef' = implExp {
    HashSet rez = new HashSet(all_setS);
    HashSet rez1 = new HashSet($e.set);
    while ($e.equals(rez1)) {
      rez.clear();
      rez.addAll(rez1);
      rez1.addAll(tmp);
    }
  }
  $set = rez1;
}
```

Fig. 2. Implementation of the EF operator in ANTLR

Analogue were implemented all CTL temporal operators. For efficient representation of CTL models, our tool is based on SingleGraph class from GraphStream package [9].
IV. ARCHITECTURE OF THE CTL MODEL CHECKER TOOL

The system architecture of the CTL model checker tool is depicted in the following UML package diagram:

![UML diagram](image)

The CTL model checker tool contains the following packages:

- The algebraic compiler (CTL Compiler) embedded into the Web Service (CTL Checker).
- The GUI client application written in C# and used for interactive construction of the CTL models as directed graphs (CTL Designer).
- The CTL non-GUI model package contains classes used for internal representation of a CTL model as a directed graph. There are available two libraries, for C# and respectively for Java. For internal representation of a CTL model in C#, our implementation is based on data structures provided by [7], more precisely symmetrically stored forward and backward adjacency lists. The Java implementation is based on GraphStream [9].
- The XML API for CTL models package contains classes needed to encode the CTL model into XML.
- The CTL GUI Model package is responsible with graphical representation of the Kripke structures as directed graphs.

We choose to publish our implementation of CTL model checker as a Web service in order to utilize the combined resources of distributed computers and to bring advantages of distributed verification to various clients over the Web. For a better understanding of the CTL model checking process, in Fig. 4 is represented the Use Case Diagram of our model checker. The Web service will receive from a client the XML representation of a CTL model \( M \) and a CTL formula \( \phi \) to be verified. The original form of the CTL model \( M \) is then reconstructed and passed to the algebraic compiler \( C \) generated by ANTLR using our CTL extended grammar. For a syntactically correct formula \( \phi \), the compiler will return as result \( C(\phi) = \{ s \in S \mid s \vDash \phi \} \), the set of states in which the formula is satisfied. If as input is an erroneous formula \( \phi \), the model checker will return to client an error message describing the error. In our tool is enabled a compression facility for large CTL models, to reduce the network traffic between client and server. Our Web service is using GlassFish or Tomcat as a Web container. For testing purposes, the CTL model checker described in this paper is available online via two Web services hosted by use-it.ro and respectively by mcheck-useit.rhcloud.com.

V. PERFORMANCE EVALUATION OF OUR TOOL

In this section we describe the usage of our model-checker to design a game strategy when playing Tic-Tac-Toe (called TTT for short in the rest of this paper). First, let us describe the (classical) game of Tic-Tac-Toe. The game is played by two opponents, \( X \) and \( O \), with a turn-based modality on a 3×3 board. The two players take turns to put pieces on the board. A single piece is put for each turn and a piece once put does not move. A player wins the game by first lining three of his or her pieces in a straight line, no matter horizontal, vertical or diagonal. If the entire board becomes full but no player has formed a line, the result is a draw. In our example, player \( X \) is played by the application and player \( O \) should be played by a human. CTL model checking algorithm is used to return a strategy to achieve a winning strategy for the computer. The TTT is a turn-based synchronous game. In such a system, at every transition there is just one agent that is permitted to make a choice (and hence determine the future). In the following we will show how to use the CTL formalizations in finding winning strategies in case of TTT game.

A. Modelling the Game

We suppose that positions of the board are numbered as in Fig. 5:

![Labeling the grids on the board](image)

Values of the board locations are denoted by \( x_i \in \{0,1,2\} \), where \( i \in \{0,1,...,8\} \). The value 0 means an empty position, the value 1 denotes a previous move of the player \( X \) and the value 2 represents a move of the player \( O \). For the sequence of values \( x_0 x_1 x_2 \), we define:

\[
\sum x_i x_j x_k = \min(x_i, 1) + \min(x_j, 1) + \min(x_k, 1)
\]

where \( l,m,n \in \{0,1,...,8\} \). Formally, the Kripke model of TTT is defined as \( M = (S, Rel, P:S \rightarrow 2^P) \) with its structure explained in the following. The set of atomic propositions \( AP \) is denoted by:
The state we denote by \( s \) is a winning state for player \( X \) if the number of empty positions of the board.

Another situation where the game is not continuing is when a player won. The state \( s \) is a winning state for player \( X \) if \( \bar{T} = 1 \) signify that is turn of the player \( X \) for making the move and if \( \bar{T} = 2 \) then the player \( O \) will make the next move. The game stops (so no moves are possible) if the board moves locations are full, i.e.:

\[
\sum_{i=0,3,6} x_i x_{i+1} y_{i+2} = 9
\]

A state labeled with value \( \bar{T} = 1 \) signify that is turn of the player \( X \) for making the move and if \( \bar{T} = 2 \) then the player \( O \) will make the next move. The game stops (so no moves are possible) if the board moves locations are full, i.e.:

\[
\sum_{i=0,3,6} x_i x_{i+1} y_{i+2} = 9
\]

Another situation where the game is not continuing is when a player won. The state \( s \) is a winning state for player \( X \) if \( \bar{T} = 1 \) signify that is turn of the player \( X \) for making the move and if \( \bar{T} = 2 \) then the player \( O \) will make the next move. The game stops (so no moves are possible) if the board moves locations are full, i.e.:

\[
\sum_{i=0,3,6} x_i x_{i+1} y_{i+2} = 9
\]

A state labeled with value \( \bar{T} = 1 \) signify that is turn of the player \( X \) for making the move and if \( \bar{T} = 2 \) then the player \( O \) will make the next move. The game stops (so no moves are possible) if the board moves locations are full, i.e.:

\[
\sum_{i=0,3,6} x_i x_{i+1} y_{i+2} = 9
\]

B. Experimental results

Although the game implemented is relatively simple, due to the large size of the structure representing the CTL model at the first moves, it represents a good opportunity to study the effectiveness of our approach in designing and implementing a CTL model checker. In the Table II are presented the results showing the performance of the CTL model checker when running on Intel Core i5, 2.5 GHz, 4Gb RAM to find a winning strategy for Tic-Tac-Toe game.

| TABLE II. EVALUATING THE PERFORMANCE OF THE CTL MODEL CHECKER |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
|                  | States          | Time (sec)      |                  |                  |                  |
|                  | 2236            | 0.13            | 2307            | 0.14            | 2425            | 0.16            | 3423            | 0.32            | 3683            | 0.35            | 3732            | 0.43            | 4255            | 0.6             | 4791            |

VI. Conclusions

In this paper was presented a new CTL model checking tool, based on client/server architecture. In contrast to previous approaches, our tool permits an interactive design of the CTL models as state-transition graphs, and is based on client/server architecture. The implementation of the CTL model checking algorithm is based on Java code generated by ANTLR using an original grammar for the language of CTL formulas, having semantic actions attached to production rules. Programmatic construction of large models is supported through two libraries (C# and Java). The server part of our tool was published as a Web service and is available online at two Internet locations, use-it.ro and mcheck-useit.rhcloud.com. All components of our CTL model checker tool can be downloaded from http://use-it.ro.

REFERENCES