Using CTL Model Checker for Verification of Domain Application Systems

LAURA FLORENTINA CACOVEAN
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

Abstract: Application driven software methodology for the execution of domain solutions is a two step process: translation and interpretation. SADL is a process-description language used in application driven software to represent domain solutions in an intermediate form. In this paper we present a method for integrating the CTL model checking technology and SADL to prove the correctness of generated computing processes that execute the algorithms expressed in SADL. This approach is applied in domain of Web applications.

Key-Words: CTL, model checker, SADL, WEB, application domain

1 Introduction
Software systems were developed using tools to solve problems that are independent of a particular problem domain. With these tools a solution of the problem is transformed into a program in a conventional programming language. These approaches are advancing to a substantial increase in software complexity. Development of methodology for structuring the application domain using ontological engineering tools allows design of domain based on Software Architecture Description Language (SADL), which take elements of problem solving process as a natural language expression of the problem domain [2]. This SADL interpretation implements and integrates the computational approaches, used in SADL expressions, and generates the computation processes that execute the algorithms in term of SADL. In application domain ontology, these approaches developed by IT experts are domain characteristic terms attached with the processes that implement them. Software developed process is based on well-define concepts that characterize application domain of the software thus developed. Language processing as domain application, illustrates this methodology of problem solving. We using as example a CTL (Computation Tree Logic) model checker and show how programming language experts can develop this model without associative programming. This model is integrated into Unified Modeling Language (UML) where we constructed the Diagram Activity. The UML Activity Diagram can help to describe the flow of control of the target system. We illustrate the computational emancipation used in an language processing with the following problem: \((M,s)\not\models f\) where \(M\) is a model of a parse system and \(s\) is a state of model and \(f\) is a model CTL formula that must be satisfied by systems. Problem solving methods provide reusable architectures and components for implementing the reasoning part of knowledge-based systems. These architectures and components manipulate computer artifacts the term problem-solving. We started from the idea presented in [2].

2 CTL model Checker
The model checkers are tools which can be used to verify a given system satisfies a given temporal logic formula. The model is a directed graph where the nodes represent the states of the system and the edges represents the state transitions. The nodes and the edges can be labeled with atomic propositions (AP) what describe the states and the transitions of the system. In order to be verified by a given model, a property is written as a temporal logic formula across the labeled propositions from the model. A model checker is an algorithm that determines the states of a model that satisfy a temporal logic formula.

We used the algebraic methodology and its utilization in developing instruments for model checker specifications as maps in form \(C:L_s\rightarrow L_t\) [4], where \(L_s\) is the source language of temporal logic, \(L_t\) is the target language representing sets of states of the model \(M\) and \(C(f\in L_s)\rightarrow s\in M[s\not\models f]\), where \(\not\models\) is satisfaction relation. This allows the possibility of automatic generation of the model checking algorithms for temporal logics into algebraic specifications sets. Extensibility and flexibility of algebraic methodology show how the model checkers for various temporal logics can be generated from algebraic specification. In paper [4] we showed how this algebraic context can be used to the specification of CTL model checker.

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. A Kripke model \(M\) over \(AP\) is a triple \(M=(S, Rel, P:AP\rightarrow 2^S)\) where \(S\) is a
finite set of states, \( Rel \subseteq S \times S \) is a transition relation, and \( P:S \to 2^{AP} \) is a function that assigns each state with a set of atomic propositions.

A. Syntax definition of a CTL model checker [5]

A CTL has the following syntax given in Backus-Naur Form:

\[
\begin{align*}
& f :: \langle X | \langle \| \rangle | \langle \| \rangle f1 | f1 \land f2 | f1 \rightarrow f2 | AX f1 | EX f1 | AG f1 | EG f1 | AF f1 | EF f1 | A[f1 U f2] | E[f1 U f2] \rangle
\end{align*}
\]

where \( \forall p \in AP \).

B. Semantics definition of a CTL model checker is provided in [5]

Let \( M=(S, Rel, P:AP \to 2^S) \) be a Kripke model for CTL. Given any \( s \) in \( S \), in [3] is defined if a CTL formula \( f \) holds in state \( s \). This is denoted with \( (M,s) \vDash f \). The satisfaction relation \( \vDash \) is defined by structural induction on fourteen CTL formulas [5].

3 Problem domain application

Software systems are developed using problem solving instruments that are independent of specific problem domain. With these tools, a problem solution is program conversion by a conventional programming language. The conventional consequence of following route is the absence of problem abstract domains in the process of problem solving.

Ontology engineering is the needed link because it can be used to bring the computational power of the computer to the computer end-user. Ontology provides the abstractions domain application experts need to express naturally their problems and solution algorithms while computer artifacts associated as semantics with the concepts of the ontology allow IT experts to develop software that maps domain application systems into computer processes that implement them.

The frame for description and development software based on applications is supported by: (1) methodology development for application-domain structuring using engineering tools; this will enable the design of domain-driver software architecture description language (SADL) that capture elements of the problem-solving process as expressions in the natural language of the problem domain; (2) implement SADL interpreters that integrate computing abstractions (used in SADL expressions) and generate computing processes that execute the algorithms expresses in SADL; (3) use this methodology to conduct problem solving experiments in such different problem domains (e.g. Internet Agents). This methodology will be integrated within applied computing courses through the development of “hands-on-experience course modules”.

The computational emancipation supposes two things which should be understood: (1) a particular domain application is produced by a computational structure of oriented-domain, that means the computational emancipation process of a domain application can be obtained by domain application structuring using ontology; (2) computational structure of the model is given by IT semantics, that means it is made by delivery of ontology concepts with IT semantics that has a computational mean [2].

The elements of the vocabulary of the language ontology are: alphabet, lexical entities, discourse, objects and actions, properties, syntax and semantics are related by two operators Value : Syntax \( \to \) Semantics and Discourse : Semantics \( \to \) Syntax [2]. In Fig 1 this tree given by [2] represents the vocabulary of the ontology. Nodes represent concepts constructed in terms of other concepts. The arrows and the dotted lines represent relationships between the concepts in the ontology.

![Fig 1. Language Ontology](image)

The computational meanings of the concepts used by the domain ontology are universal algorithms that solve classes of problems, are characteristic to the domain, and are mathematically proven correct. The goal of computational emancipation of a domain application is to allow domain applications experts to use computer technology as a problem solving tool dedicated to their application domain without requiring the application domain expert to develop IT representations.

We illustrate the use of computational emancipation in language processing with the following problem: \( (M,s) \vDash f \), where \( M \) is a model of analyzed system, and \( s \) is a state of model and \( f \) is a CTL model formula that must be satisfied by the system.

We use the ontology terms for show formalization of domain knowledge. In order to use the application driven software [1] for solving our problem, we are interested by the little subsequence of terms from domain which represents the computation process characterized by of the behavior of their input/output date as follows: the connecters CTL are sets \( \Gamma=\{\neg, \land, \lor, \rightarrow, EX, EF, EG, EU, AX, AF, AG, AU\} \). From connecters of above we will only refer to \( \neg, \land, EX, AF, EU \). Syntactically, CTL formulas are divided into three categories: (i) those whose outermost operator, if any, is not a temporal operator; (ii) those whose outer most operator is a temporal operator (X (next), U (until), F (eventually) or G (always) prefixed with the existential path quantifier E, and (iii) those whose outer most operator is a
temporal operator prefixed with the universal path quantifier $A$.

The idea of this algorithm is to: (a) decompose formula $f$ in sub-formulas and apply a structural induction to label the graph with sub-formulas of $f$ (the intuition is that a formula that labels a state is true in that state; (b) for each such sub-formula, parse the graph to infer the truth in a state according to the meaning of the connectives and the truth values of its sub-formulas.

The ontology development for the application domain represent the identification the terms and their organization which to permit the implication in reasoning’s process. The ontology for application domain, in our case the language programming, can be very large. However, ontology terms can be classified on a hierarchy of sub-domains and the problem solving can be centre just on the sub-domain ontology appropriate.

In the following, we are concerned with a small subset of terms in the domain whose meanings are computational processes characterized by their input/output behavior as follows:

The function $\text{CTL}_f(f)$ takes a CTL formula as input and returns the set of states satisfying the formula. It calls the function $\text{CTL}_\neg(f)$, $\text{CTL}_\text{EX}(f)$, $\text{CTL}_\text{AF}(f)$, $\text{CTL}_\text{EU}(f_1,f_2)$, if EX or EU is the root of the input’s parse tree.

function $\text{CTL}_f(f)$ /*determines the set of states satisfying $f$*/
Input: a CTL model $M = (S,\rightarrow,F)$ and a CTL formula $f$ (in $\Gamma$-format)
Output: the set of states of $M$ which satisfy $f$,
\( (M,s_0)\models f \)

In real applications, the emancipation of calculating a scope is a transaction on knowledge. This is represented using description logic where each term will be replaced with a uniform resource identifier (URI), which indicates that computer artifact implements the concept represented by this term.

The function presented hereinbefore can be writing like follows:

\begin{verbatim}
CTL_f;
Input: \{Model M, Formula F\}
Output: Sets S
S:= SetOfStateSatisfyFormula(M,F);
\end{verbatim}

The function presented hereinbefore can be writing in SADL expression as:

\begin{verbatim}
<system name="URI(CTL_f)"
  input="URI(M) URI(F)"
  output="URI(S)"/>
</system>
</sadl>

For the other functions will show only the functions definition with input and output data.

function $\text{CTL}_f(p)$ /* determines the set of states satisfying $p$*/
Input: $p$
Output: the set of states of $M$ which satisfy $p$, $(M,s_0)\models p$

function $\text{CTL}_\neg(f)$ /* determines the set of states satisfying $\neg f$*/
Input: $f$
Output: the set of states of $M$ which satisfy $\neg f$, $(M,s_0)\models \neg f$

Input: formula $f$
Output: the set of states of $M$ which satisfy $Af$

The function $\text{CTL}_\text{EX}$ computes the states satisfying $f$ by calling $\text{CTL}_f$. Then, it looks backwards along $\rightarrow$ to find the states satisfying $EXf$.

function $\text{CTL}_\text{EX}(f)$ /*determines the set of states satisfying $EXf$*/
Input: formula $f$
Output: the set of states of $M$ which satisfy $EXf$

The algorithm and its functions use program variable $Y$ is a sets of states [5]. The SAT algorithm uses direct cases more easily and passes to complicated cases using special procedures, which, in turn, could call recursively the sub-expression. The special procedures rely on functions.

\begin{verbatim}
\text{pre_exist}(Y)=\{ s \in S \mid exists s', (s\rightarrow s' and s' \in Y) \}
\text{pre_all}(Y)=\{ s \in S \mid for all s', (s\rightarrow s' implies s' \in Y) \}
\end{verbatim}

Both functions compute a pre-image of set of states. $\text{pre_exist}(Y)$, instrumental in $\text{CTL}_\text{EX}$ and $\text{CTL}_\text{EU}$, takes of subset $Y$ of states and returns the set of states which can make a transition into $Y$. The function $\text{pre_all}(Y)$ is used in $\text{CTL}_\text{AF}$, takes a set $Y$ and returns a set of states which make transitions only into $Y$.

\begin{verbatim}
\text{pre_all}(Y) = S - \text{pre_exist}(S-Y)
\end{verbatim}

We start with the equivalence $E[f_1\cup f_2] \equiv f_1 \cup (f_1 \land EX E[f_2])$ and we write it as $E\{f_1\cup f_2\} = (E\{f_1\}) \cup (E\{f_2\}) \land $ $\text{pre_exist}[E\{f_1\cup f_2\}]$. The values $Y_0 = \text{CTL}_f(f)$, $Y_i=Y_{i-1}\cap\text{pre_exist}(Y_{i-1})$, $Y_f = Y_0\cap \text{pre_exist}(Y_{\infty})$. In [5] is show the proof of equivalence $Y_f = Y_0\cap \text{pre_exist}(Y_{\infty})$.

function $\text{CTL}_\text{EU}(f_1,f_2)$ /*determines the set of states satisfying $E[f_1\cup f_2]$*/
Input: formula $f_1,f_2$
Output: the set of states of $M$ which satisfy $E[f_1\cup f_2]$

The SADL expression of $\text{CTL}_\text{EU}$ function can be writing like:
The SADL expression thus obtained is still mapped to the interpretation of SADL in the scope that the algorithm runs through interpretation.

4 Computational Emancipation of Application Domain

Applying this approach presented in third section to the language sub-domain, the computational emancipation of application domain is done by associating each term in the language ontology with a uniform resource identifier (URI) pointing to the IT component implementing it. We organize these terms in a tree giving us the ontology process of computational emancipation language shown in Fig 2.

We get away from our example we shall build follow-up the ontology of processing ontology of emancipation language.

For any problem, the solution is formulated in the domain of discourse from which the problem belongs. As language processing experts we provide the following formal solution for our problem, formulated in terms of the computationally emancipated language processing domain.

Input: formula $f$
Output: $M \not\models f$

switch ($f$)

case cond1 : RESULT := CTL_EX;
case cond2 : RESULT := CTL_AF;
case cond3 : RESULT := CTL_EU;
default: return Error;
end switch

where cond1 means formula is $p$; cond2 means formula is $\neg f$; cond3 means formula is $f_1 \land f_2$; cond4 means formula is $\text{EX} f_1$; cond5 means formula is $\text{AF} f_1$; cond6 means formula is $E[f_1 \cup f_2]$. CTL$_f$, CTL$_\neg$, ..., CTL$_E$ are function which have a result a value of true if the formula satisfy the model $M$.

Follow up we shall execute the SADL solution. Application driven software methodology for the execution of domain solutions is a two step process: translation and interpretation. Before a solution can be executed it must be first translated into a process interpretable language called the Software Architecture Description Language [1]. This translation replaces each domain concept with its associated IT artifact and each operational instruction with an appropriate SADL operator. After translation to SADL, the solution process is carried out by the SADL interpreter.

5 Interpretation of SADL Language

SADL is a process-description language used in application driven software to represent domain solutions in an intermediate form. The structure of the SADL language is built on the following principles. The lexical elements of the language are application domain terms and SADL operators. The semantics of application domain terms are the software artifacts associated with them in the application domain ontology. The semantics of SADL operators are specified by the SADL interpreter. The syntax of SADL is built on the extensible mark-up language (XML) syntax. The two types of SADL processes are represented by the two types of XML elements: (a) SADL primitive processes are represented by empty XML elements of the form $<\text{op} \ atr_1 = \text{val}_1 \ldots \ atr_n = \text{val}_n />$ where op is a SADL operator which performs a process and $\ atr_1, \ldots, \ atr_n$ defined the properties of that process such as the URI of the code that implements input, output, etc.; (b) SADL composed processes are represented by content XML elements of the form $<\text{op} \ atr_1 = \text{val}_1 \ldots \ atr_n = \text{val}_n >$ $p_1 \ldots p_n</<\text{op} />$ where op is a SADL operator that composes the processes $p_1 \ldots p_n$ using $\ atr_1, \ldots, \ atr_n$ determine the behavior of the composition.

The process performed by the SADL for each element is determined by the SADL interpreter. For our solution the interpreter recognizes the following set of SADL process operators.

Interpretation of the SADL language is handled by the SADL interpreter acting as a virtual processor. The processes performed by the SADL interpreter are based
on SADL semantics defined above and each process defined by the software artifact used in the ontology is performed on the computer platform in the network where it exists. The challenge for the interpreter is in performing process composition. In sequential composition this means executing software artifacts, waiting for execution to terminate and performing the next composition. This challenge is compounded by the fact that solutions contain sub-compositions as is the case of the switch with case process. The assumption here is that software artifacts associated with ontology nodes are correct and terminating.

The first problem is to represent the input and output nodes associated with computer processes the XML file and associated ontology of ontology nodes with an extensible stylesheet language, XSLT transformer that performs mapping of inputs expected results [1,2]. We can use an XSLT designed for use as part of XSL (extensible stylesheet language) which is a stylesheets language for XML, to transform a UML model encoded in an XMI document into XML Schema.

The second problem is to find a universal way to address specific processes associated with computing ontology nodes.

For the solution given in Fig 3 the SADL form of our domain solution is given in Section 3. The process performing this solution is the composition of processes between the <system> and </system> tags.

```
<system name="verification">
  <execute component="&it:semipc_formula" input="input" output="formula"/>
  <switch atr="formula">
    <case test="cond1">
      <node name="RESULT" term="CTL_f"/>
      <execute component="&it:semipcCTL_f" input="input" output="CTL_f"/>
    </case>
    ...
    <case test="cond4">
      <node name="RESULT" term="CTL_EX"/>
      <execute component="&it:semipcCTL_EX" input="input" output="CTL_EX"/>
    </case>
  </switch>
</system>
```

Fig 3. The SADL solution

6 Verification a Web Application using CTL Model Checker

Driving model architecture [6] is currently used in software engineering as a generic term to describe using models in software engineering process. The models are computer representations of the components correspond to the elements or concepts of the domain problem. We use UML for construct the activity diagram to express the model that we want to be check.

We propose in this section to present a sample web application where we use the CTL model checking for verification. The proposed methods are based on a model checking technique.

Web application refers to a login page with user name and password. The model will check if logging is allowed or not depending on input data. In Fig 4 is presented the place of model checking into verification process. Model checking is also helpful to allocate errors because, when the system does not satisfy the property, counterexample is output with the result.

```
Property of CTL model checker
Specification Rule
CTL model checker
false
true

Fig 4. Web application verified with CTL
```

The UML Activity Diagram can help to describe the flow of control of the target system. The activity diagram for the web application is presented in Fig 5.
The Kripke model has five states and the propositional variables are from the set \{UserName, Password, Button, Error\}. UserName represented the insert the user name, Password means inserting the password, and Button is push the button for Sing-in or Sign-out and Error means occur some error. The formal definition of the Kripke structure of the elevator control is given by: \( M = (S, \text{Rel}, P) \), where \( S = \{s_0, s_1, \ldots, s_4\} \), \( \text{Rel} = \{(s_0, s_1), (s_1, s_0), (s_1, s_2), (s_2, s_1), (s_2, s_3), (s_2, s_4), (s_3, s_0), \} \), \( \text{AP} = \{\text{UserName, Password, Button, Error}\} \), \( P \) assigns state \( s_0 \) in \( M \) with \{not UserName, not Password, not Button pushed and not Error\}, that is set \{\neg \text{UserName}, \neg \text{Password}, \neg \text{Button}, \neg \text{Error}\}. \( P \) assigns state \( s_1 \) in \( M \) with \{UserName, not Password, not Button, not Error\}, the state \( s_2 \) in \( M \) with \{UserName, Password, not Button, not Error\}, the state \( s_3 \) in \( M \) with \{UserName, Password, Button, not Error\}, the state \( s_4 \) in \( M \) with \{UserName, Password, not Button, Error\}.

The Kripke model is shown in Fig 6:

![Fig 6. The CTL structure of login Page Controller](image)

For the model presented in Fig 6 an important task is to verify formula \( \text{ag} (\neg (\text{Button and Error})) \) who means to see if there exist a node which not have in the same time the push sign-in button and have the error. The interpretation and verification of this formula presuppose these: We have the atomic proposition \( \text{Button} \) in state \( \{s_3\} \) and the atomic proposition \( \text{Error} \) in state \( \{s_4\} \). The \( (\text{Button and Error}) \) is empty sets because any state from model does not contain both atomic propositions. Hence \( \neg (\text{Button and Error}) \) formula is true in all states \( \{s_0, \ldots, s_4\} \) and the same \( \text{ag} (\neg (\text{Button and Error})) \) satisfy formula for the proposed model.

7 Conclusion
The behavior of the model checker interpretation performed by the SADL presented in the third section consists in identifying the set of states of a given model \( M \) which satisfy each sub formula of a given CTL formula \( f \) and constructing the set of states, from these sets, that satisfy the formula \( f \). In this paper we show an alternative methodology whose goal is to move the process of problem solving from the problem domain where it belongs. We use the computational domain concepts and abstractions to develop algorithms and express solutions in a natural language related problem. These expressions are then interpreted by personalized software which executes calculations involved in computer network under the appropriate operating systems. These presented approaches are showing how the CTL model checker can be easily integrated into web applications for verification of transitions from one state to another.

References: