May 27, 2021

1 The Syntax of Computation Tree Logic

Computation tree logic (CTL) was introduced by Turing award winners Clarke and Emerson [3]. The formulas of this logic consist of the constants `true` and `false` and so-called atomic propositions which are combined by means of several operators that we will discuss below. The *atomic propositions* are used to express basic facts about the states of the system. That is, these atomic propositions are state predicates. In the next section, we provide some concrete examples of atomic propositions in the context of Java code.

CTL contains the operators

- negation, denoted $\neg$,
- conjunction, denoted by $\wedge$,
- disjunction, denoted $\vee$,
- implication, denoted $\rightarrow$, and
- equivalence, denoted $\leftrightarrow$.

Furthermore, it contains

- universal quantification, denoted $\forall$, and
- existential quantification, denoted $\exists$.

Finally, it contains the so-called temporal operators

- next, denoted $\bigcirc$,
- until, denoted U,
- always, denoted $\Box$, and
- eventually, denoted $\Diamond$.

Let us formally define the syntax of CTL. Let AP be the set of atomic propositions. The set of CTL formulas is defined by the following grammar.

$$\begin{aligned}\varphi ::= &(\varphi) \mid a \\ &\mid \texttt{true} \mid \texttt{false} \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \rightarrow \varphi \mid \varphi \leftrightarrow \varphi \\ &\mid \forall \bigcirc \varphi \mid \exists \bigcirc \varphi \mid \forall \varphi \,\mathsf{U}\, \varphi \mid \exists \varphi \,\mathsf{U}\, \varphi \mid \forall \Box \varphi \mid \exists \Box \varphi \mid \forall \Diamond \varphi \mid \exists \Diamond \varphi\end{aligned}$$

where $a \in AP$.

In order to make sense of a CTL formula such as

$$\forall \bigcirc a \rightarrow b \rightarrow c$$

we need to define the precedence of the operators. Furthermore, we need to specify whether the binary operators are left or right associative. For the order of precedence, we use the commonly accepted order (from highest to lowest): $\neg$, $\wedge$, $\vee$, $\rightarrow$, and $\leftrightarrow$. According to Baier and Katoen [1], U takes precedence over $\wedge$, $\vee$, and $\rightarrow$ (they do not consider $\leftrightarrow$). Usually, unary operators have higher precedence than binary ones. Hence, the operators, listed from highest to lowest precedence, are

$$\begin{array}{l}\neg \\ \forall\bigcirc, \exists\bigcirc, \forall\Box, \exists\Box, \forall\Diamond, \exists\Diamond \\ \forall\mathsf{U}, \exists\mathsf{U} \\ \wedge \\ \vee \\ \rightarrow \\ \leftrightarrow\end{array}$$

The binary operators $\wedge$, $\vee$ and $\leftrightarrow$ are (left) associative. Usually, $\rightarrow$ is considered right associative. According to Baier and Katoen [1], U is also right associative.

Using the above specified precedence and associativity rules, the above CTL formula is interpreted as

$$(\forall \bigcirc a) \rightarrow (b \rightarrow c)$$

To express the CTL formulas in ASCII, we use the following grammar.

$$\begin{aligned}\varphi ::= &(\varphi) \mid a \\ &\mid \texttt{true} \mid \texttt{false} \mid \texttt{!}\varphi \mid \varphi \,\texttt{\&\&}\, \varphi \mid \varphi \,\texttt{||}\, \varphi \mid \varphi \,\texttt{->}\, \varphi \mid \varphi \,\texttt{<->}\, \varphi \\ &\mid \texttt{AX}\,\varphi \mid \texttt{EX}\,\varphi \mid \varphi \,\texttt{AU}\, \varphi \mid \varphi \,\texttt{EU}\, \varphi \mid \texttt{AG}\,\varphi \mid \texttt{EG}\,\varphi \mid \texttt{AF}\,\varphi \mid \texttt{EF}\,\varphi\end{aligned}$$

The ASCII representation of $\neg$, $\wedge$, and $\vee$ is taken from Java. It is common practice to use `A` and `E` for universal (for *a*ll) and existential (*e*xists) quantification. In the seminal paper by Turing award winner Pnueli [5], the temporal operators $\bigcirc$, U, $\Box$, and $\Diamond$ are represented as `X` (ne*x*t), `U` (*u*ntil), `G` (*g*lobally), and `F` (*f*uture). The representation of $\forall\varphi \,\mathsf{U}\, \varphi$ as φ `AU` φ is new, as far as we know. The above CTL formula is represented in ASCII as follows.

```
AX a -> b -> c
```

2 The Syntax of Computation Tree Logic for Java

The next operator ◯ expresses that something holds in the next state. For Java code, if one were to define the notion of next state, it would probably be the state after the next bytecode instruction has been executed. However, expressing properties of Java code in terms to steps taken at the bytecode level seems of limited, if any, use. Therefore, we do not consider the next operator ◯.

Recall that atomic propositions are used to express basic facts about the states. For now, we restrict our attention to static Boolean fields. Such an atomic proposition holds in those states in which the field has the value true. In Java, static Boolean fields are of the form

- ⟨package name⟩ . ⟨class name⟩ . ⟨field name⟩ or
- ⟨class name⟩ . ⟨field name⟩.

For example, the package `java.awt` contains the classes `AWTEvent` and `InvocationEvent`. The former contains the static field `consumed` and the latter contains `catchExceptions`. Hence, the static Boolean field `java.awt.AWTEvent.consumed` is an atomic proposition, as is `java.awt.InvocationEvent.catchExceptions`. Those fields are used as atomic propositions in the following CTL formula.

```
AG (java.awt.AWTEvent.consumed
    || EF !java.awt.event.InvocationEvent.catchExceptions)
```

3 A Lexer and Parser for CTL Formulas

A lexer and parser for CTL formulas has been developed using ANTLR [4]. The above described grammar can be specified in ANTLR format as folows.

```
formula
  : 'true'                                  #True
  | 'false'                                 #False
  | ATOMIC_PROPOSITION                      #AtomicProposition
  | '(' formula ')'                         #Bracket
  | '!' formula                             #Not
  | 'AG' formula                            #ForAllAlways
  | 'AF' formula                            #ForAllEventually
  | 'EG' formula                            #ExistsAlways
  | 'EF' formula                            #ExistsEventually
  | formula 'AU'<assoc=right> formula       #ForAllUntil
  | formula 'EU'<assoc=right> formula       #ExistsUntil
  | formula '&&' formula                    #And
  | formula '| |' formula                   #Or
  | formula '->'<assoc=right> formula       #Implies
  | formula '<->' formula                   #Iff
```

The order of the alternatives is consistent with the precedence of the operators (if an operator has higher precedence, then its alternative occurs earlier). By default, binary operators are left associative. The operators `AU`, `EU`, and `->` are specified as right associative. The second column of the above rule contains the labels of the alternatives (see [4, Section 8.2]). We will discuss their role below.

Recall that the atomic propositions are static attributes. To specify these, we used relevant snippets of the ANTLR grammar for Java[1] Whitespace, that is, spaces, tabs, and returns are skipped.

Later, we add here a discussion of error handling.

4 From Parse Tree to Abstract Syntax Tree

Next, we translate a parse tree, generated by the lexer and parser, to an abstract syntax tree. An abstract syntax tree for CTL is represented by an object of type `Formula`, which is part of the package `ctl`. A UML diagram with the classes of the `ctl` package can be found in Figure 1. The CTL formula

```
AG (java.awt.AWTEvent.consumed
    || EF !java.awt.event.InvocationEvent.catchExceptions)
```

is represented by the following `Formula` object.

```
Formula formula =
  new ForAllAlways(
    new Or(
      new AtomicProposition("java.awt.AWTEvent.consumed"),
      new ExistsEventually(
        new Not(
          new AtomicProposition("java.awt.event.InvocationEvent.catchExcepti
        )
      )
    )
  );
```

To implement this translation, we use the visitor design pattern. ANTLR supports this design pattern (see [4, Section 7.3]). From the CTL grammar, ANTLR generates a `CTLVisitor` interface. This interface contains a visit method for each alternative. For example, for the alternative labelled `ExistsAlways`, the interface contains the method `visitExistsAlways`.

ANTLR also generates the `CTLBaseVisitor` class. This adapter class provides a default implementation for all the methods of the `CTLVisitor` interface. We implement our translation by extending this class and overriding methods. For example, when we visit a node of the parse tree corresponding to the alternative labelled `Implies`, we first visit the left child and obtain the `Formula` object corresponding to the translation of the parse tree rooted at that left child. Next,

[1] See github.com/antlr/grammars-v4/tree/master/java/java8.

we visit the right child and obtain the `Formula` object for the parse tree rooted at that right child. Finally, we create an `Implies` object from those two `Formula` objects.

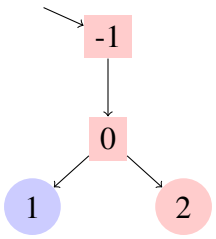
```
@Override
public Formula visitImplies(ImpliesContext context) {
  Formula left = (Formula) visit(context.formula(0));
  Formula right = (Formula) visit(context.formula(1));
  return new Implies(left, right);
}
```

5 Testing the Lexer, the Parser, and the Translation

6 A New Semantics for CTL

The normal semantics of CTL is described in [1, Section 6.2.2]. This normal semantics is defined for a transition system $(S, Act, \rightarrow, I, AP, L)$. Such a transition system is defined in [1, Definition 2.1] The new semantics considers a partial transition system. A partial transition system is a tuple $(S, F, Act, \rightarrow, I, AP, L)$, where all components are defined as before and $F \subseteq S$ is a set of fully explored states. A transition system is called partial because the states $S \setminus F$ are not fully explored yet, that is, these states have transitions that are not part of the transition relation $\rightarrow$.

Consider the following partial transition system.



State -1 is the initial state. The states -1 and 0 are fully explored, and states 1 and 2 are not fully explored. Consider, for example, the CTL formula $\exists\Diamond$ blue. This formula holds in the above partial transition system, since state 1 is blue and can be reached from the initial state. The CTL formula $\forall\Box$ red does not hold for the same reason. Now consider the CTL formula $\forall\Box$ (red $\vee$ blue). The above partial transition system does not provide a counterexample to this formula as all states that can be reached from the initial state are either red or blue. However, since states 1 and 2 are not fully explored, either state may have a successor that is neither red nor blue. So, the best we can say is "don't know." Hence, whether a partial transition system satisfies a CTL formula can be answered as either yes ($\top$), no ($\bot$), or don't know (?).

Recall that the satisfaction relation $\models$, defined in [1, Definition 6.4], can be viewed as mapping a state s of a transition system and a CTL formula φ to a Boolean, that is, (s, φ) is mapped to true if $s \models \varphi$ and mapped to false otherwise. The satisfaction relation $\models$ for CTL formulas on partial transition systems can be viewed as a mapping from states and formulas to $\top$, $\bot$, and ?.

We modify the definition of a transition system, as given in [1, Definition 2.1], as follows.

Definition 1. A *partial transition system* is a tuple $(S, F, Act, \to, I, AP, L)$ consisting of

- a set S of *states*,
- a set $F \subseteq S$ of *fully explored states*,
- a set Act of *actions*,
- a transition relation $\to \subseteq S \times Act \times S$,
- a set $I \subseteq S$ of *initial states*,
- a set AP of *atomic propositions*, and
- a *labelling function* $L : S \to 2^{AP}$.

The difference between a partial transition system and an ordinary transition system is the set F of fully explored states. Since the set Act of actions does not play in the remainder, we will drop it from the definition and simplify the transition relation to $\to \subseteq S \times S$. The partial transition system depicted in the introduction can be formally defined as $(S, F, \to, I, AP, L)$ where

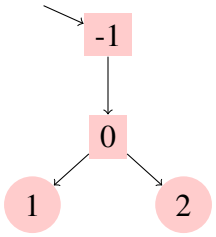
- $S = \{-1, 0, 1, 2\}$,
- $F = \{-1, 0\}$,
- $\to = \{(-1, 0), (0, 1), (0, 2)\}$,
- $I = \{-1\}$,
- $AP = \{\text{blue}, \text{red}\}$, and
- and the function $L : S \to 2^{AP}$ is defined by

$$\begin{aligned} L(-1) &= \{\text{red}\} \\ L(0) &= \{\text{red}\} \\ L(1) &= \{\text{blue}\} \\ L(2) &= \{\text{red}\} \end{aligned}$$

7 JPF Listener that Writes a Partial Transition System to File

The first objective of this project is to develop a listener for Java PathFinder (JPF) that writes its partial transition system to file. In [2, Section 7.3], a listener that writes a transition system to file has been developed. This listener is extended to the setting of partial transition systems.

Consider the following partial transition system.



State -1 is the initial state. The states -1 and 0 are fully explored, and states 1 and 2 are not fully explored. The listener produces a file, the name of which is the name of the system under test with ".tra" as suffix (see [2, Section 7.4]), with the following content.

```
-1 -> 0
0 -> 1
0 -> 2
1 2
```

The first three lines describe the transitions. The last line contains the states that are not fully explored.

References

[1] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, Cambridge, MA, USA, 2008.

[2] Franck van Breugel. Java PathFinder: a tool to detect bugs in Java code. 2020.

[3] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In Dexter Kozen, editor, *Proceedings of the 3rd Workshop on Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71, Yorktown Heights, NY, USA, May 1981. Springer-Verlag.

[4] Terence Parr. *The Definitive ANTLR 4 Reference*. The Pragmatic Bookshelf, Dallas, TX, USA, 2013.

[5] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57, Providence, RI, USA, October/November 1977. IEEE.

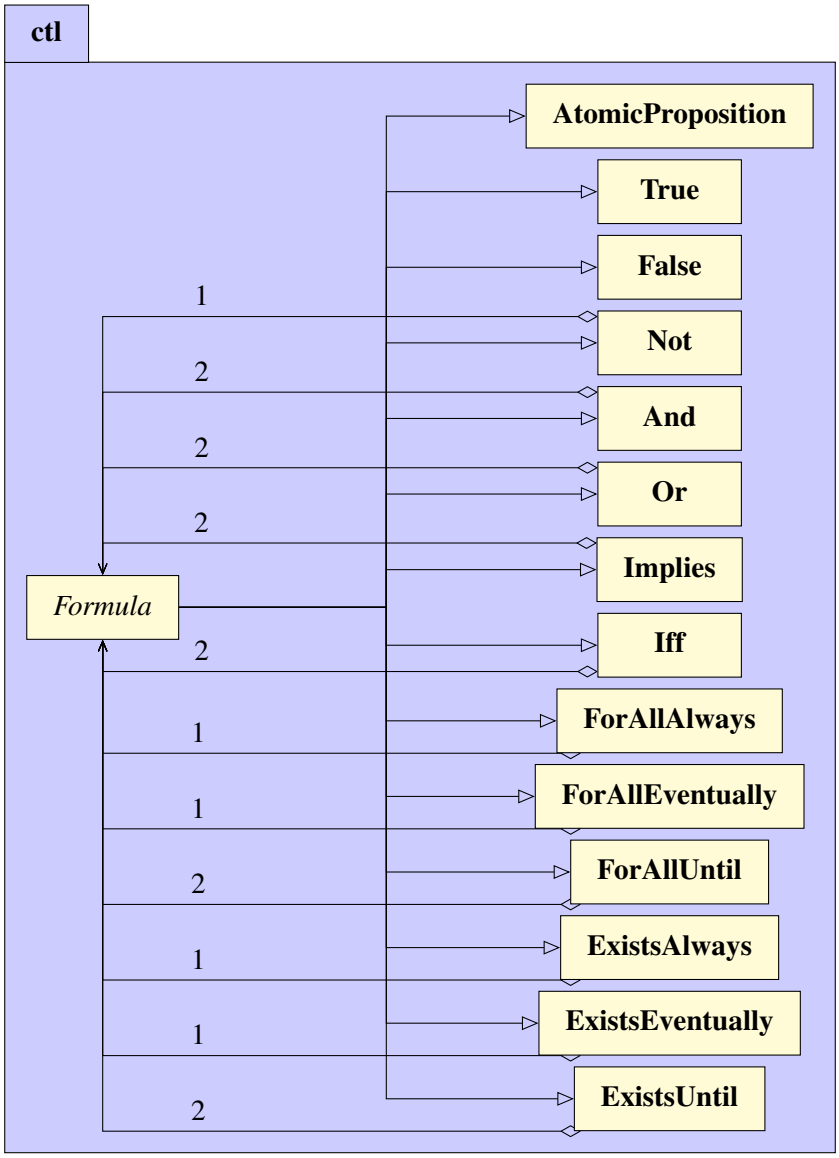


Figure 1: UML class diagram of the abstract syntax classes.