pyModelChecking Documentation

Release 1.2.0

Alberto Casagrande

Apr 28, 2019

Contents

pyModelChecking is a simple Python model checking package. Currently, it is able to represent *Kripke structures*, *Propositional Logics*, *CTL*, *LTL*, and *CTL** formulas and it provides model checking methods for LTL, CTL, and CTL*. In future, it will hopefully support symbolic model checking.

CHAPTER 1

Basic Notions

1.1 Reactive Systems

Reactive systems are systems that interact with their environment and evolve over an infinite time horizon. This chapter presents a natural model for them: Kripke structure.

1.1.1 Directed Graphs

A **directed graph**, or **graph**, is pair (V, E) where:

- V is a finite set of *nodes*
- $E \subseteq V \times V$ is a set of *edges*

If $(s, d) \in E$, then s and d are the *source* and the *destination* of (s, d), respectively. The edge $(s, d) \in E$ is said to *go from :math:'s' to :math:'d'*. If $e \in E$ goes either from s to d or from d to s, then e is an edge **between** d and s. By extension, an edge $e \in E$ goes from $V_1 \subseteq S$ to $V_2 \subseteq S$ if there exists a pair of nodes $(v_1, v_2) \in V_1 \times V_2$ such that $(v_1, v_2) \in E$. Analogously, $e \in E$ is between V_1 and V_2 if it is either from V_1 to V_2 or from V_2 to V_1.

The **reversed graph** of a graph (V, E) is the graph (V, E') where $E' = (d, s)|(s, d) \in E$.

A **subgraph** of a graph (V, E) is a graph (V', E') such that $V' \subseteq V$ and $E' \subseteq E \cap (V' \times V')$. A subgraph (V', E') of (V, E) is a **proper subgraph** if either $V' \subsetneq V$ or $E' \subsetneq E$. A subgraph G of (V, E) **respects** a set of nodes $V' \subseteq V$ if $G = (V', E \cap (V' \times V'))$.

A sequence, either finite or infinite, $\pi = v_0 v_1 \ldots$ is a **path** for the graph (V, E) if $(v_i, v_{i+1}) \in E$ for all v_i and v_{i+1} in π. The *lenght of a path* π, denoted by $|\pi|$, is the size of the sequence.

It is easy to see that if $\pi = v_0 \ldots v_n$ and $\pi' = w_0 \ldots$ are two paths for (V, E) such that $(v_n, w_0) \in E$, then $\pi \cdot \pi' = v_0 \ldots v_n w_0 \ldots$ is path for (V, E).

Let π, π', and π'' be three paths such that $\pi = \pi' \cdot \pi''$. Then, π' is a **prefix** of π and π'' is a **suffix** of π. We write π_i to denote the suffix of π for which $\pi = \pi' \cdot \pi_i$ and $|\pi'| = i$ for some π'.

If $v_0 v_1 \ldots v_n$ is a prefix for some path π of a graph (V, E), then we say that either π *starts* from v_0 and **reaches** v_n or, equivantely, v_n is **reachable** from v_0 in (V, E).

Every *subgraph* (V', E') of G such that:

1. v is reachable from v' for all pairs $v, v' \in V'$ and
2. is not proper subgraph of any subgraph of G that satisfies 1.

is a **strongly connected component** of G. It is easy to see that the the sets of nodes of each strongly connected component of a graph (V, E) is a partition of V. A strongly connected component (V', E') is **trivial** if $|V'| = 1$ and $|E'| = 0$.

Directed Acyclic Graphs and Trees

A **directed acyclic graph** or **DAG** is a directed graph whose strongly connected components are all trivial.

A graph (V, E) is **disconnected** if there exists a $V' \subseteq V$ such that there are no edges between V' and $V \setminus V'$. If a graph is not disconnected, then is **connected**.

A **directed tree** is a connected DAG (V, E) whose subgraphs of the form (V, E'), where $E' \subsetneq E$, are disconnected.

1.1.2 Kripke Structures

A **Kripke structure** is a *directed graph*, equipped with a set of initial nodes, such that every node is source of some edge and it is labeled by a set of *atomic propositions* [CGP00]. The nodes of Kripke structure are called *states*.

A Kripke structure is a tuple (S, S_0, R, L) such that:

- S is a finite set of states
- $S_0 \subseteq S$ is a set of *initial states*
- $R \subseteq S \times S$ is a set of *transitions* such that for all $s \in S$ there exists a $(s, s') \in R$ for some $s' \in S$
- $L : S \rightarrow AP$ maps each state into a set of atomic propositions

Sometime, the set of initial states is omitted. In such cases, S and S_0 coincide.

A **computation** of a Kripke structure (S, S_0, R, L) is an infinite path of (S, R) that starts from some $s \in S_0$.

1.2 Logics

1.2.1 Propositional Logics

Propositional Logics or **PL** is an extension of Boolean logics that handles propositional symbols. Beside the standard logical operators logical operators $\neg$, $\wedge$, $\vee$, and $\rightarrow$, it is also equipped with propositional variables whose Boolean values can be declared at evalutation time. An **atomic proposition** or **AP** is either a Boolean value –i.e., $\top$ (true) or $\bot$ (false)– or a prositional variable.

Syntax

A PL formula is either:

- $\top$ or $\bot$
- a propositional variable
- $\neg\varphi_1$, $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$, or $\varphi_1 \rightarrow \varphi_2$ where both φ_1 and φ_2 are PL formulas

Semantics

Since *pyModelChecking* is primary meant to support model checking, we provide the semantics of propositional formulas with respect to a *Kripke structure*. If K is a Kripke structure, s one of its states, and φ a propositional formula, we write $K, s \models \varphi$ (to be read "K and s satisfy φ") meaning that φ holds at state s in K.

Let K be the Kripke structure (S, S_0, R, L); the relation $\models$ is defined recursively as follows:

- $K, s \models \top$ and $K, s \not\models \bot$ for any state $s \in S$
- if $p \in AP$, then $K, s \models p \iff p \in L(s)$
- $K, s \models \neg\varphi \iff K, s \not\models \varphi$
- $K, s \models \varphi_1 \wedge \varphi_2 \iff K, s \models \varphi_1$ and $K, s \models \varphi_2$
- $K, s \models \varphi_1 \vee \varphi_2 \iff K, s \models \varphi_1$ or $K, s \models \varphi_2$
- $K, s \models \varphi_1 \rightarrow \varphi_2 \iff K, s \not\models \varphi_1$ or $K, s \models \varphi_2$

1.2.2 Computational Tree Logic*

The **Computational Tree Language*** or **CTL*** is a the temporal logic that describes the properties of computation trees over Kripke structures ([CE81], [CES86]). It is a proper extension of propositional logics and, beside a set of atomic propostions and the standard logical operators $\neg$, $\wedge$, $\vee$, and $\rightarrow$, the alphabet of CTL* contains the two path quantifiers $\mathbf{A}$ ("for all paths") and $\mathbf{E}$ ("for some path") and the five temporal operators $\mathbf{X}$ ("at the next step"), $\mathbf{G}$ ("globally"), F ("in the future"), $\mathbf{U}$ ("until"), and $\mathbf{R}$ ("release").

Syntax

Any CTL* formula is either a *state formula* (i.e., a formula that are evaluated in a single state) or a *path formula* (i.e., a formula whose truth value depend on an infinite path).

A CTL* state formula is either:

- $\top$ or $\bot$
- a propositional variable
- $\neg\varphi_1$, $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$, or $\varphi_1 \rightarrow \varphi_2$ where both φ_1 and φ_2 are CTL* state formulas
- $\mathbf{A}\psi$ or $\mathbf{E}\psi$ where ψ is a CTL* path formula

A CTL* path formula is either:

- a state formula
- $\neg\psi_1$, $\psi_1 \wedge \psi_2$, $\psi_1 \vee \psi_2$, or $\psi_1 \rightarrow \psi_2$ where both ψ_1 and ψ_2 are CTL* path formulas
- $\mathbf{X}\psi_1$, $\mathbf{F}\psi_1$, $\mathbf{G}\psi_1$, $\psi_1\mathbf{U}\psi_2$, or $\psi_1\mathbf{R}\psi_2$ where both ψ_1 and ψ_2 are CTL* path formulas

Semantics

The semantics of CTL* formulas is given with respect to a *Kripke structure* and it is a proper extension of the sematics of propositional logics. If K is a Kripke structure, s one of its states, and φ a state formula, we write $K, s \models \varphi$ meaning that φ holds at state s in K. Analogously, If K is a Kripke structure, π one of its computations, and ψ a path formula, we write $K, \pi \models \psi$ meaning that ψ holds along π in K.

Let K be the Kripke structure (S, S_0, R, L); the relation $\models$ is defined recursively as follows:

- $K, s \models \top$ and $K, s \not\models \bot$ for any state $s \in S$

- if $p \in AP$, then $K, s \models p \iff p \in L(s)$
- $K, s \models \neg\varphi \iff K, s \not\models \varphi$
- $K, s \models \varphi_1 \wedge \varphi_2 \iff K, s \models \varphi_1$ and $K, s \models \varphi_2$
- $K, s \models \varphi_1 \vee \varphi_2 \iff K, s \models \varphi_1$ or $K, s \models \varphi_2$
- $K, s \models \varphi_1 \rightarrow \varphi_2 \iff K, s \not\models \varphi_1$ or $K, s \models \varphi_2$
- $K, s \models \mathbf{A}\varphi \iff K, \pi \models \varphi$ for any computation π of K that starts from s
- $K, s \models \mathbf{E}\varphi \iff K, \pi \models \varphi$ for some computation π of K that starts from s
- $K, \pi \models \psi \iff K, s \models \psi$, where π is a computation of K that starts from s
- $K, \pi \models \neg\psi \iff K, \pi \not\models \psi$
- $K, \pi \models \psi_1 \wedge \psi_2 \iff K, \pi \models \psi_1$ and $K, \pi \models \psi_2$
- $K, \pi \models \psi_1 \vee \psi_2 \iff K, \pi \models \psi_1$ or $K, \pi \models \psi_2$
- $K, \pi \models \psi_1 \rightarrow \psi_2 \iff K, \pi \not\models \psi_1$ or $K, \pi \models \psi_2$
- $K, \pi \models \mathbf{X}\psi \iff K, \pi_1 \models \psi$
- $K, \pi \models \mathbf{F}\psi \iff K, \pi_i \models \psi$ for some $i \in \mathbb{N}$
- $K, \pi \models \mathbf{G}\psi \iff K, \pi_i \models \psi$ for all $i \in \mathbb{N}$
- $K, \pi \models \psi_1 \mathbf{U} \psi_2 \iff$ there exists an $i \in \mathbb{N}$ such that $K, \pi_i \models \psi_2$ and $K, \pi_j \models \psi_1$ for all $j \in [0, i-1]$
- $K, \pi \models \psi_1 \mathbf{R} \psi_2 \iff$ for all $i \in \mathbb{N}$, if $K, \pi_j \not\models \psi_1$ for all $j \in [0, i-1]$, then $K, \pi_i \models \psi_2$

Whenever $K, \sigma \models \psi \iff K, \sigma \models \varphi$ for any σ and and any K, we say that ψ and φ are **equivalent** and we write $\varphi \equiv \psi$.

Two set of formulas $\mathcal{F}$ and and any $\mathbf{G}$ are **equivalent** if any formula $\mathbf{G}$ has an equivalent formula in $\mathcal{F}$ and vice versa.

Restricted Syntax

It is easy to prove that $\bot$, $\mathbf{F}\psi$, $\mathbf{G}\psi$, $\varphi\mathbf{R}\psi$, $\mathbf{A}\varphi$, $\varphi \wedge \psi$, and $\varphi \rightarrow \psi$ are equivalent to $\neg\top$, $\top\mathbf{U}\psi$, $\neg(\top\mathbf{U}\neg\psi)$, $\neg(\neg\varphi\mathbf{U}\neg\psi)$, $\neg\mathbf{E}\neg\varphi$, $\neg(\varphi \vee \psi)$, and $\neg\varphi \vee \psi$, respectively. Thus, the CTL* language whose alphabet is restricted to $\neg$, $\vee$, $\mathbf{X}$, $\mathbf{U}$, $\mathbf{A}$, $\top$, and atomic propositions is equivalent to the full CTL* language (e.g., see [CGP00]).

1.2.3 Computational Tree Logic

The **Computational Tree Language** or **CTL** is a subset of *CTL** ([BMP83], [CE81], [CE80]). In CTL, each occurence of the two path quantifiers $\mathbf{A}$ and $\mathbf{E}$ should be coupled to one of the temporal operators $\mathbf{X}$, $\mathbf{G}$, $\mathbf{F}$, $\mathbf{U}$, or $\mathbf{U}$.

Syntax

More formally, a CTL state formula is either:

- $\top$ or $\bot$
- propositional variable
- $\neg\varphi_1$, $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$, or $\varphi_1 \rightarrow \varphi_2$, where both φ_1 and φ_2 are CTL state formulas
- $\mathbf{A}\psi$ or $\mathbf{E}\psi$ where φ is a CTL path formula

A CTL path formula is either $\mathbf{X}\varphi_1$, $\mathbf{F}\varphi_1$, $\mathbf{G}\varphi_1$, $\varphi_1\mathbf{U}\varphi_2$, or $\varphi_1\mathbf{R}\varphi_2$ where both φ_1 and φ_2 are CTL state formulas.

Semantics

CTL has the same *semantics of CTL**.

Restricted Syntax

Despite the apperent syntatic complexity of CTL, any possible property definable in it can be expressed by a CTL formula whose syntax is restricted to the use of $\top$, $\neg$, $\vee$, and $\mathbf{E}$ coupled to either $\mathbf{X}$, $\mathbf{U}$, or $\mathbf{G}$ (e.g., see [CGP00]). As a matter of the facts, it is easy to prove that:

- $\bot \equiv \neg\top$
- $\varphi_1 \wedge \varphi_2 \equiv \neg(\neg\varphi_1 \vee \neg\varphi_2)$
- $\varphi_1 \rightarrow \varphi_2 \equiv \neg\varphi_1 \vee \varphi_2$
- $\mathbf{AX}\varphi \equiv \neg\mathbf{EX}(\neg\varphi)$
- $\mathbf{E}F\varphi \equiv \mathbf{E}(\top\mathbf{U}\varphi)$
- $\mathbf{AG}\varphi \equiv \neg\mathbf{E}(\top\mathbf{U}\neg\varphi)$
- $\mathbf{A}F\varphi \equiv \neg\mathbf{EG}(\neg\varphi)$
- $\mathbf{A}(\varphi_1\mathbf{U}\varphi_2) \equiv \neg(\mathbf{E}((\neg\varphi_2)\mathbf{U}\neg(\varphi_1 \vee \varphi_2)) \vee \mathbf{EG}(\neg\varphi_2))$
- $\mathbf{A}(\varphi_1\mathbf{R}\varphi_2) \equiv \neg\mathbf{E}((\neg\varphi_1)\mathbf{U}(\neg\varphi_2))$
- $\mathbf{E}(\varphi_1\mathbf{R}\varphi_2) \equiv (\mathbf{E}(\varphi_2\mathbf{U}(\neg\varphi_1 \vee \neg\varphi_2)) \vee \mathbf{EG}(\varphi_2))$

1.2.4 Linear Time Logic

The **Linear Time Logic** or **LTL** is a subset of of *CTL** ([P77]).

Syntax

LTL formulas have the form $A\rho$ where ρ is a LTL path formula and a LTL path formula is either:

- $\top$ or $\bot$
- propositional variable
- $\neg\varphi_1$, $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$, or $\varphi_1 \rightarrow \varphi_2$, where both φ_1 and φ_2 are LTL path formulas
- $\mathbf{X}\varphi_1$, $\mathbf{F}\varphi_1$, $\mathbf{G}\varphi_1$, $\varphi_1\mathbf{U}\varphi_2$, or $\varphi_1\mathbf{R}\varphi_2$ where both φ_1 and φ_2 are LTL path formulas.

Semantics

LTL has the same *semantics of CTL**.

Restricted Syntax

It is easy to prove that:

- $\psi_1 \wedge \psi_2 \equiv \neg(\neg\psi_1 \vee \neg\psi_2)$
- $\psi_1 \rightarrow \psi_2 \equiv \neg\psi_1 \vee \psi_2$
- $\mathbf{F}\psi \equiv \top\mathbf{U}\psi$
- $\mathbf{G}\psi \equiv \neg(\top\mathbf{U}\neg\psi)$
- $\psi_1\mathbf{R}\psi_2 \equiv \neg((\neg\psi_1)\mathbf{U}(\neg\psi_2))$

Hence, the LTL restricted language that allows exclusively the path formulas whose operators are $\neg$, $\vee$, $\mathbf{X}$, or $\mathbf{U}$ is equivalent to the full LTL language (e.g., see [CGP00]).

1.3 Model Checking

Model checking is a technique to establish the set of states in Kripke structure that satisfy a given temporal formula. More formally, provided a Kripke structure $K = (S, S_0, R, L)$ and a temporal formula φ, model checking aims to identify $S' \subseteq S$ such that

$$K, s_i \models \varphi$$

for all $s_i \in S'$.

Model checking problem for *CTL**, *CTL* and *LTL* is decidable even though the time complexity of the algorithm is logics dependent: the complexities of the *CTL*, *LTL* and *CTL** decision procedures are $O(|\varphi| * (|S| + |R|))$, $O(2^{O(|\varphi|)} * (|S| + |R|))$ and $O(2^{O(|\varphi|)} * (|S| + |R|))$, respectively.

1.3.1 Fair Model Checking

A *fair Kripke structure* is a Kripke structure (S, S_0, R, L) added with a set of *fair states* $F \subseteq S$. A *fair path* for it is an infinite path that passes through all the fair states infinitely often.

Fair model checking only considers fair paths. A *fair state* is a path from which at least one fair path originates.

1.4 Symbolic Representation

Binary Decision Diagrams (BDDs) and *Ordered Binary Decision Diagrams* (OBDDs) are data structures to represent binary functions [Bryant86].

1.4.1 Binary Decision Diagrams

BDDs are *directed graphs* whose nodes can be either **terminal** or **non-terminal**. Terminal nodes are labelled by a *binary value* and they are not source of any edge. If t is a terminal node, we write $t.value$ to denote the value of t. Non-terminal nodes are labelled by a *variable* name and they are source of two edges called *low* and *high*. If n is a non-terminal node, we write $n.var$, $n.low$, and $n.high$ to denote the variable name, the edge low, and the edge high of the node n.

Any terminal node t represents the binary function $t.value$, while any non-terminal node n encodes the binary function $(\tilde{n}.var\&f_l)|(n.var\&f_h)$ where f_l and f_h are the binary functions associated to $n.low$ and $n.high$, respectively.

A BDD **respects a variable ordering** $<$ whenever $n.var < n.low.var$ for all non-terminal nodes n and $n.low$ and $n.var < n.high.var$ for all non-terminal nodes n and $n.high$.

1.4.2 Ordered Binary Decision Diagrams

The logical equivalence of two binary functions can be reduced to the existence of an isomorphism between the BDD encoding them under three conditions:

1. the two BDDs respect the same variable ordering;
2. $n.low$ and $n.high$ are different nodes for any non-terminal node n in both the BDDs;
3. for each of the BDDs and for all pairs of nodes in it, there is no isomophism between them.

OBDDs are BDDs equipped of a variable ordering and satisfying condition 2. and 3.

Whenever two binary functions f_1 and f_2 are stored as OBDD and they share the same variable ordering, it is possible to:

- test logical equivalence between f_1 and f_2 in time $O(1)$;
- compute the OBDD that represents:
  - the bitwise negation of the formula f_1 in time $O(|f_1|)$;
  - the bitwise binary combinations of the functions f_1 and f_2 in time $O(|f_1| + |f_2|)$.

CHAPTER 2

Using *pyModelChecking*

2.1 Modelling Reactive Systems

2.1.1 Directed Graphs

Directed graphs can be represented in *pyModelChecking* by using the class `DiGraph` (see *Graph API*).

```
>>> from pyModelChecking import *
>>> G = DiGraph(V=['a',3],
...             E=[('a','a'), ('a','b')])
>>> print(G)

(V=['a', 3, 'b'], E=[('a', 'a'), ('a', 'b')])

>>> G.nodes()

['a', 3, 'b']

>>> G.edges()

[('a','a'), ('a','b')]

>>> G.add_edge('c','b')
>>> G.nodes()

['a', 'c', 3, 'b']

>>> G.edges()

[('a', 'a'), ('a', 'b'), ('c', 'b')]
```

The same class provides methods to compute **reachable sets**, **reversed graphs** and **subgraphs** of a given directed graph.

```
>>> print(G.get_reversed_graph())

(V=['a', 'c', 3, 'b'], E=[('a', 'a'), ('b', 'a'), ('b', 'c')])

>>> print(G.get_subgraph(['a','b',3]))

(V=['a', 3, 'b'], E=[('a', 'a'), ('a', 'b')])

>>> print(G.get_reachable_set_from(['a', 3]))

set(['a', 3, 'b'])

>>> print(G.get_reachable_set_from(['d']))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "pyModelChecking/graph.py", line 203, in get_reachable_set_from
    for d in self.next(s):
  File "pyModelChecking/graph.py", line 120, in next
    'of this DiGraph')
RuntimeError: src = 'd' is not a node of this DiGraph
```

pyModelChecking can also compute the strongly connected components of a directed graph.

```
>>> G.add_edge('b','a')
>>> print(list(compute_strongly_connected_components(G)))
[['a', 'b'], ['c'], [3]]
```

Refer to *Graph API* for more details.

2.1.2 Kripke Structures

Kripke structures are representable by using the class `Kripke` (see *Kripke API*).

```
>>> from pyModelChecking import *
>>> K = Kripke(S=[0, 1, 3],
...            R=[(0, 2), (2, 2), (0, 1), (1, 0), (3, 2)],
...            L={1: ['p', 'q'], 2: ['p', 'q'], 3: ['q']})
>>> print(K)

(S=[0, 1, 2, 3],S0=set([]),R=[(0, 1), (0, 2), (1, 0), (2, 2), (3, 2)],L={0: set([]),
↪1: set(['q', 'p']), 2: set(['q', 'p']), 3: set(['q'])})
```

The sets of Kripke's states and transitions can be obtained by using the following syntax:

```
>>> K.states()

[0, 1, 2, 3]

>>> K.transitions()

[(0, 1), (0, 2), (1, 0), (2, 2), (3, 2)]
```

It is possible to get the successors of a given state with respect to the Kripke's transitions:

```
>>> K.next(0)

set([1, 2])
```

Finally, the API provides a method for getting the labels of Kripke's states.

```
>>> K.labels()

set(['q', 'p'])

>>> K.labels(3)

set(['q'])
```

2.2 Encoding Formulas and Model Checking

pyModelChecking provides a user friendly support for building *CTL**, *CTL* and *LTL* formulas. Each of these languages corresponds to a *pyModelChecking*'s sub-module which implements all the classes required to encode the corresponding formulas.

Propositional logic is also supported by *pyModelChecking* as a shared basis for all the possible temporal logics.

2.2.1 Propositional Logics

Propositional logics support is provided by including the *pyModelChecking.language* sub-module. This sub-module allows to represents atomic propositions and Boolean values through the `pyModelChecking.formula.AtomicProposition` and `pyModelChecking.formula.Bool` classes, respectively.

```
>>> from pyModelChecking.PL import *
>>> AtomicProposition('p')

p

>>> Bool(True)

true
```

Moreover, the `pyModelChecking.PL` sub-module implements the logic operators $\land$, $\lor$, $\rightarrow$ and $\neg$ by mean of the classes `pyModelChecking.PL.And`, `pyModelChecking.PL.Or`, `pyModelChecking.PL.Imply` and `pyModelChecking.PL.Not`, respectively. These classes automatically wrap strings and Boolean values as objects of the classes `pyModelChecking.PL.AtomicProposition` and `pyModelChecking.PL.Bool`, respectively. All cited classes are subclasses of the class `pyModelChecking.PL.Formula`.

```
>>> And('p', true)

(p and true)

>>> And('p', true, 'p')

(p and true and p)

>>> f = Imply('q','p')
>>> And('p', f, Imply(Not(f), Or('q','s', f)))
```

(continues on next page)

(continued from previous page)

```
(p and (q --> p) and (not (q --> p) --> (q or s or (q --> p))))

>>> Imply('p', 'q', 'p')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __init__() takes exactly 3 arguments (4 given)
```

In order to simplify formula encoding, the operators ~, &, and | –i.e., `pyModelChecking.PL.__not__()`, `pyModelChecking.PL.__and__()`, and `pyModelChecking.PL.__or__()`– were overwritten to be used as shortcuts to `pyModelChecking.PL.Not`, `pyModelChecking.PL.And`, and `pyModelChecking.PL.Or` constructors, respectively. At least one of the operator parameters should be an object of the class `pyModelChecking.PL.Formula`.

```
>>> AtomicProposition('p') & True

(p and true)

>>> True & AtomicProposition('p')

(true and p)

>>> f = 'p' & Bool(True)
>>> f

(p and true)

>>> True & 'p' & Bool(True)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for &: 'bool' and 'str'

>>> 'p' & Bool(True) & 'p'

((p and true) and p)

>>> ~('p' & Bool(True)) | And(~f,'b')

(not (p and true) or (not (p and true) and b))
```

For user convenience, the function `pyModelChecking.PL.LNot()` is also provided. This function returns a formula equivalent to logic negation of the parameter and minimise the number of outermost ¬.

```
>>> f = Not(Not(Not(And('p',Not('q')))))
>>> f

not not not (p and not q)

>>> LNot(f)

(p and not q)

>>> LNot(Not(f))

not (p and not q)
```

(continues on next page)

(continued from previous page)

```
>>> LNot(LNot(f))

not (p and not q)
```

Parsing Formulas

The module *pyModelChecking.PL* also provides a parsing class `pyModelChecking.PL.Parser` for propositional formula. Its objects read a formula from a string and, when it is possible, translate it into a corresponding `pyModelChecking.PL.Formula` objects.

```
>>> p = Parser()

>>> p('p and true')

(p and true)

>>> p('(~p and q) --> ((q | p))')

((not p and q) --> (q or p))
```

A complete description of the parser grammar is contained in class member `pyModelChecking.PL.Parser.grammar`

```
>>> print(p.grammar)

    s_formula: "true"     -> true
             | "false"    -> false
             | a_prop
             | "(" s_formula ")"

    u_formula: ("not"|"~") u_formula  -> not_formula
             | "(" b_formula ")"
             | s_formula

    b_formula: u_formula
             | u_formula ( ("or"|"|") u_formula )+ -> or_formula
             | u_formula ( ("and"|"&") u_formula )+ -> and_formula
             | u_formula ("-->") u_formula -> imply_formula

    a_prop: /[a-zA-Z_][a-zA-Z_0-9]*/ -> string
          | ESCAPED_STRING            -> e_string

    formula: b_formula

    %import common.ESCAPED_STRING
    %import common.WS
    %import WS
```

2.2.2 Temporal Logics Implementation

CTL* formulas can be defined by using the *pyModelChecking.CTLS* sub-module.

```
>>> from pyModelChecking.CTLS import *
```

Path quantifiers A and E as well as temporal operators X, F, G, U and R are provided as classes (see *CTLS sub-module* for more details). As in the case of propositional logics, these classes wrap strings and Boolean values as objects of the classes `pyModelChecking.CTLS.language.AtomicProposition` and `pyModelChecking.CTLS.language.Bool`, respectively.

```
>>> phi = A(G(
...         Imply(And(Not('Close'),
...                   'Start'),
...               A(Or(G(Not('Heat')),
...                    F(Not('Error')))))
...         ))
>>> phi

A(G(((not Close and Start) --> A((G(not Heat) or F(not Error))))))
```

As far as parsing capabilities and siplifying syntax concern, `pyModelChecking.CTLS` has the same facilities `pyModelChecking.PL` had and implements $CTL*$ specific version of both class `pyModelChecking.CTLS.Parser` and operators ~, &, and |.

```
>>> p=Parser()
>>> p('G(not Heat)') | p('A(F(not Error))')

(G(not Heat) or A(F(not Error)))
```

2.2.3 Model Checking Formulas

The sub-module also implements the CTL* model checking and fair model checking algorithms described in [CGP00].

```
>>> from pyModelChecking import Kripke
>>> K = Kripke(R=[(0, 1), (0, 2), (1, 4), (4, 1), (4, 2), (2, 0),
...               (3, 2), (3, 0), (3, 3), (6, 3), (2, 5), (5, 6)],
...            L={0: set(), 1: set(['Start', 'Error']), 2: set(['Close']),
...               3: set(['Close', 'Heat']),
...               4: set(['Start', 'Close', 'Error']),
...               5: set(['Start', 'Close']),
...               6: set(['Start', 'Close', 'Heat'])})
>>> modelcheck(K, psi)

set([0, 1, 2, 3, 4, 5, 6])

>>> modelcheck(K, psi, F=[6])

set([])
```

It is also possible to model check a string representation of a CTL* formula by either passing an object of the class `pyModelChecking.CTLS.Parser` or leaving the remit of creating such an object to the function `pyModelChecking.CTLS.modelcheck()`.

```
>>> modelcheck(K, psi_str)

set([0, 1, 2, 3, 4, 5, 6])

>>> modelcheck(K, psi_str, parser=parser)

set([0, 1, 2, 3, 4, 5, 6])
```

Analogous functionality are provided for *CTL* and *LTL* by the sub-modules `pyModelChecking.CTL` and `pyModelChecking.LTL`, respectively.

CHAPTER 3

API

3.1 Models API

pyModelChecking provides implementations for *directed graph* and *Kripke structures*.

3.1.1 Graph API

It is used to define *directed graphs* and provides a method to compute the strogly connected components of a directed graph.

class pyModelChecking.graph.**DiGraph**(*V=None*, *E=None*)

Bases: `object`

A class to represent directed graphs.

A *directed graph* is a couple (V,E) where V is a set of vertices and E is a set of edges (i.e., pairs of vertices). If (s,d) in E, then s and d are said *source* and *destination* of (s,d).

add_edge(*src*, *dst*)

Add a new edge to a DiGraph

Parameters

- **src** – the source node of the edge
- **dst** – the destination node of the edge

add_node(*v*)

Add a new node to a DiGraph

Parameters

- **self** (`DiGraph`) – the DiGraph object
- **v** – a node

clone()
Clone a DiGraph

Returns a clone of the DiGraph

Return type *DiGraph*

edges()
Return the edges of a DiGraph

Returns a list of edges of the DiGraph

Return type list

edges_iter()
Return the edges of a DiGraph

Returns the generator of edges of the DiGraph

Return type generator

get_reachable_set_from(*nodes*)
Compute the reachable set

Parameters **nodes** (`a container of nodes`) – the set of nodes from which the reachability should be evaluated

Returns the set of the reachable nodes

Return type set

get_reversed_graph()
Build the reversed graph

Returns the reversed graph

Return type *DiGraph*

get_subgraph(*nodes*)
Build the subgraph that respects a set of nodes

Returns the subgraph that respects :param nodes:

Return type *DiGraph*

next(*src*)
Return the next of a node

Given a DiGraph (V, E) and one of its node v, the *next* of $v \in V$ is the set of all those nodes v' that are destination of some edge $(v, v') \in E$.

Returns the set of nodes $\{v'|(v, v') \in E\}$

Return type set

nodes()
Return the nodes of a DiGraph

Returns the list of the nodes of the DiGraph

Return type list

sources()
Return the sources of a DiGraph.

The *sources* of a DiGraph G are the nodes that are sources of some edges in G itself.

Returns a generator of all the nodes that are sources of some edges

Return type generator

`pyModelChecking.graph.compute_SCCs(G)`

Compute the strongly connected components of a DiGraph

This method implements a non-recursive version of the Nuutila and Soisalon-Soinen's algorithm ([ns94]) to compute the strongly connected components of a DiGraph.

Parameters **G** (`DiGraph`) – the DiGraph object

Returns a generator of the sets of nodes of the strongly connected components of the DiGraph

Return type generator

3.1.2 Kripke API

It is used to define *Kripke structures*.

class `pyModelChecking.kripke.Kripke(S=None, S0=None, R=None, L=None)`

Bases: `pyModelChecking.graph.DiGraph`

A class to represent Kripke structures.

A Kripke structure is a directed graph equipped with a set of initial nodes, S0, and a labelling function that maps each node into the set of atomic propositions that hold in the node itself. The nodes of Kripke structure are called *states*.

`clone()`

Clone a Kripke structure

Returns a clone of the current Kripke structure

Return type *Kripke*

`get_fair_states(F)`

Return a set of states from which leaves a fair path.

Parameters **F** (*a container*) – a container of fairness constraints

Returns the set of states from which leaves a fair path

Return type set

`get_substructure(V)`

Return the sub-structure that respects a set of states

The sub-structure of a Kripke structure (V', E', L') that respects a set of states V is the Kripke structure (V, E, L) where $E = E' \cap (V imes V)$ and $L(v) = L'(v)$ for all $s \in V$.

Parameters **V** (*set*) – a set of states

Returns the sub-structure that respects V

Return type *Kripke*

`label_fair_states(F)`

Label all the fair states by a new atomic proposition.

This method labels all the states from which a fair path exists by using a new atomic proposition that means "there exists a fair path from here". The new label is returned.

Parameters **F** (*a container*) – a container of fairness constraints

Returns a new label that means "there exists a fair path from here"

Return type str

labelling_function()
Return the labelling function

Returns the labelling function

Return type dict

labels(*state=None*)
Get the atomic propositions labelling either a state or the whole structure

Parameters **state** – either a state of the Kripke structure or None

Returns the atomic propositions that label either a *state*, whenever a parameter *state* is passed, or at least one state of the Kripke structure, otherwise

Return type set

next(*src*)
Return the next of a state

Given a Kripke structure $K = (S, S0, R, L)$ and one of its state s, the *next* of s in K is the set of all those states that are destination of some edges whose source is s itself i.e., $K.next(s) = \{s'|(s, s') \in R\}$.

Returns the set of nodes $\{s'|(s, s') \in R\}$

Return type set

replace_labelling_function(L)
Replace the labelling function

Parameters **L** (`dict`) – a new labelling function for this Kripke structure

Returns the former labelling function

Return type dict

states()
Return the states of a Kripke structure

Returns the states of the Kripke structure

Return type set

transitions()
Return the edges of a Kripke structure

Returns the set of edges of the Kripke structure

Return type set

transitions_iter()
Return an interator of the edges of a Kripke structure

Returns an interator of the set of edges of the Kripke structure

Return type iterator

3.2 Logics and Model Checking API

The implementations of specific languages and their model checking routines are contained in *pyModelChecking* sub-modules. CTL*, CTL, and LTL are handled by *CTLS sub-module*, *CTL sub-module* and *LTL sub-module*, respectively.

3.2.1 Propositional Logics sub-module API

It represents *Proposition Logics formulas*.

Language

class pyModelChecking.PL.language.**And**(**phi*)

Bases: *pyModelChecking.PL.language.LogicOperator*, pyModelChecking.language.And

Represents logic conjunction.

class pyModelChecking.PL.language.**AtomicProposition**(*name*)

Bases: *pyModelChecking.PL.language.Formula*, pyModelChecking.language.AlphabeticSymbol

The class representing atomic propositionic propositions.

clone()

Clones an atomic proposition

Returns a clone of the current atomic proposition

Return type PL.AtomicProposition

subformula(*i*)

Returns the i-th subformula.

Parameters **i** (*Integer*) – the index of the subformula to be returned

Raises **TypeError** – atomic propositions have not subformulas

subformulas()

Returns the list of all the subformulas.

Returns returns the empty list of the subformulas of the current formula

Return type list

class pyModelChecking.PL.language.**Bool**(*value*)

Bases: pyModelChecking.language.Bool, *pyModelChecking.PL.language.AtomicProposition*

The class of Boolean atomic propositions.

class pyModelChecking.PL.language.**Formula**(**phi*)

Bases: pyModelChecking.language.Formula

A class to represent propositional formulas.

Formulas are represented as nodes in labelled trees: leaves are terminal symbols (e.g., atomic propositions and Boolean values), while internal nodes correspond to operators and quantifiers. The ariety of internal nodes depends on the kind of operator or quantifier must be represented. For instance, the arity of a node representing the formula $not(p \vee True)$ is one because the formula has exclusively one sub-formula, i.e., $p \vee True$. On the contrary, this last formula has two sub-formulas, i.e., p and $True$, thus, the node representing it has two sons.

cast_to(*Lang*)

Casts the current object in a formula of a different class.

Parameters

- **self** (*class*) – this formula
- **Lang** – a class representing a language

Returns a syntactically equivalent formula in language represented by `Lang`

Return type Lang

wrap_subformulas (*subformulas, FormulaClass*)

Replaces subformulas of the current object

This method replaces the subformulas of the current object by using the `FormulaClass` objects representing the elements of :param subformulas:.

Parameters

- **self** (*PL.Formula*) – this object
- **subformulas** (*Container*) – the set of objects to be used as model for the replacement
- **FormulaClass** (*class*) – the final class of the new subformulas

class pyModelChecking.PL.language.**Imply** (**phi*)

Bases: *pyModelChecking.PL.language.LogicOperator*, `pyModelChecking.language.Imply`

Represents logic implication.

class pyModelChecking.PL.language.**LogicOperator** (**phi*)

Bases: *pyModelChecking.PL.language.Formula*, `pyModelChecking.language.LogicOperator`

A class to represent logic operator such as $\wedge$ or $\vee$.

class pyModelChecking.PL.language.**Not** (**phi*)

Bases: *pyModelChecking.PL.language.LogicOperator*, `pyModelChecking.language.Not`

Represents logic negation.

class pyModelChecking.PL.language.**Or** (**phi*)

Bases: *pyModelChecking.PL.language.LogicOperator*, `pyModelChecking.language.Or`

Represents logic non-exclusive disjunction.

pyModelChecking.PL.language.**get_symbols** (*alphabet*)

3.2.2 CTLS sub-module API

It represents *CTL* formulas* and provides model checking methods for them.

Language

class pyModelChecking.CTLS.language.**A** (*phi*)

Bases: *pyModelChecking.CTLS.language.PathQuantifier*, `pyModelChecking.language.AlphabeticSymbol`

A class representing CTL* A-formulas.

get_equivalent_non_fair_formula (*fairAP*)

get_equivalent_restricted_formula ()

Return an equivalent formula in the restricted syntax.

This method returns a formula that avoids A, F, R, $\wedge$ and $\rightarrow$ and is equivalent to this formula.

Returns a formula that avoids A, F, R, $\wedge$ and $\rightarrow$ and is equivalent to this formula

Return type CTLS.Not

symbols = ['A']

class pyModelChecking.CTLS.language.**And**(**phi*)

Bases: *pyModelChecking.CTLS.language.LogicOperator*, *pyModelChecking.PL.language.And*

get_equivalent_restricted_formula()

Return an equivalent formula in the restricted syntax.

This method returns a formula that avoids A, F, R, $\wedge$ and $\rightarrow$ and is equivalent to this formula.

Returns a formula that avoids A, F, R, $\wedge$ and $\rightarrow$ and is equivalent to this formula

Return type CTLS.Not

class pyModelChecking.CTLS.language.**AtomicProposition**(*name*)

Bases: *pyModelChecking.PL.language.AtomicProposition*, *pyModelChecking.CTLS.language.StateFormula*

The class representing atomic propositionic propositions.

get_equivalent_non_fair_formula(*fairAP*)

get_equivalent_restricted_formula()

Return an equivalent formula in the restricted syntax.

This method returns a formula that avoids A, F, R, $\wedge$ and $\rightarrow$ and is equivalent to this formula. Since this is a method of the class `CTLS.AtomicProposition`, a clone of the current object is always returned.

Returns a clone of the current atomic proposition.

Return type CTLS.AtomicProposition

class pyModelChecking.CTLS.language.**Bool**(*value*)

Bases: *pyModelChecking.PL.language.Bool*, *pyModelChecking.CTLS.language.AtomicProposition*

The class of Boolean atomic propositions.

class pyModelChecking.CTLS.language.**E**(*phi*)

Bases: *pyModelChecking.CTLS.language.PathQuantifier*, `pyModelChecking.language.AlphabeticSymbol`

A class representing CTL* A-formulas.

get_equivalent_non_fair_formula(*fairAP*)

get_equivalent_restricted_formula()

Return an equivalent formula in the restricted syntax.

This method returns a formula that avoids A, F, R, $\wedge$ and $\rightarrow$ and is equivalent to this formula.

Returns a formula that avoids A, F, R, $\wedge$ and $\rightarrow$ and is equivalent to this formula

Return type CTLS.E

symbols = ['E']

class pyModelChecking.CTLS.language.**F**(**phi*)

Bases: *pyModelChecking.CTLS.language.TemporalOperator*, `pyModelChecking.language.AlphabeticSymbol`

get_equivalent_restricted_formula()

Return an equivalent formula in the restricted syntax.

This method returns a formula that avoids A, F, R, $\wedge$ and $\rightarrow$ and is equivalent to this formula.

Returns a formula that avoids A, F, R, $\wedge$ and $\rightarrow$ and is equivalent to this formula

Return type CTLS.U

symbols = ['F']

class pyModelChecking.CTLS.language.**Formula**(**phi*)

Bases: *pyModelChecking.PL.language.Formula*

A class to represent CTL* formulas.

Formulas are represented as nodes in labelled trees: leaves are terminal symbols (e.g., atomic propositions and Boolean values), while internal nodes correspond to operators and quantifiers. The ariety of internal nodes depends on the kind of operator or quantifier must be represented. For instance, the arity of a node representing the formula $not(A(pUq) \vee True)$ is one because the formula has exclusively one sub-formula, i.e., $A(pUq) \vee True$. On the contrary, this last formula has two sub-formulas, i.e., $A(pUq)$ and $True$, thus, the node representing it has two sons.

get_equivalent_non_fair_formula(*fairAP*)

is_a_state_formula()

Returns True if and only if the object represents a state formula.

This method should return True if and only if the current object represents a state formula. Since this is a general class meant to represent both state and path formulas, an implementation for this method cannot be provided. Thus, any call to it raise a `RuntimeError`.

Parameters **self** (*CTLS.Formula*) – this formula

Raises **RuntimeError** – this method cannot be implemented by this general class

class pyModelChecking.CTLS.language.**G**(**phi*)

Bases: *pyModelChecking.CTLS.language.TemporalOperator*, `pyModelChecking.language.AlphabeticSymbol`

get_equivalent_restricted_formula()

Return an equivalent formula in the restricted syntax.

This method returns a formula that avoids A, F, R, $\wedge$ and $\rightarrow$ and is equivalent to this formula.

Returns a formula that avoids A, F, R, $\wedge$ and $\rightarrow$ and is equivalent to this formula

Return type CTLS.Not

symbols = ['G']

class pyModelChecking.CTLS.language.**Imply**(**phi*)

Bases: *pyModelChecking.CTLS.language.LogicOperator*, *pyModelChecking.PL.language.Imply*

get_equivalent_restricted_formula()

Return an equivalent formula in the restricted syntax.

This method returns a formula that avoids A, F, R, $\wedge$ and $\rightarrow$ and is equivalent to this formula.

Returns a formula that avoids A, F, R, $\wedge$ and $\rightarrow$ and is equivalent to this formula

Return type CTLS.Not

class pyModelChecking.CTLS.language.**LogicOperator**(**phi*)

Bases: *pyModelChecking.CTLS.language.Formula*, *pyModelChecking.PL.language.LogicOperator*

A class to represent logic operator such as $\wedge$ or $\vee$.

is_a_state_formula()

Returns True if and only if the object represents a state formula.

This method returns True if and only if the current object represents a state formula.

Parameters **self** (*CTLS.LogicOperator*) – this formula

Returns True if and only if the object represents a state formula.

Return type bool

class pyModelChecking.CTLS.language.**Not**(**phi*)

Bases: *pyModelChecking.CTLS.language.LogicOperator*, *pyModelChecking.PL.language.Not*

get_equivalent_restricted_formula()

Return an equivalent formula in the restricted syntax.

This method returns a formula that avoids A, F, R, $\wedge$ and $\rightarrow$ and is equivalent to this formula.

Returns a formula that avoids A, F, R, $\wedge$ and $\rightarrow$ and is equivalent to this formula

Return type CTLS.Not

is_a_state_formula()

Returns True if and only if the object represents a state formula.

This method returns True if and only if the current object represents a state formula.

Parameters **self** (*CTLS.Not*) – this formula

Returns True if and only if this formula is a state formula.

Return type bool

class pyModelChecking.CTLS.language.**Or**(**phi*)

Bases: *pyModelChecking.CTLS.language.LogicOperator*, *pyModelChecking.PL.language.Or*

get_equivalent_restricted_formula()

Return an equivalent formula in the restricted syntax.

This method returns a formula that avoids A, F, R, $\wedge$ and $\rightarrow$ and is equivalent to this formula.

Returns a formula that avoids A, F, R, $\wedge$ and $\rightarrow$ and is equivalent to this formula

Return type CTLS.Or

class pyModelChecking.CTLS.language.**PathFormula**(**phi*)

Bases: *pyModelChecking.CTLS.language.Formula*

A class representing CTL* path formulas.

is_a_state_formula()

Returns True if and only if the object has type *StateFormula*.

This method returns True if and only if the current object has type `CTLS.StateFormula`. Since this is a method of the class `CTLS.PathFormula`, it always returns False.

Parameters **self** (*CTLS.PathFormula*) – this formula

Returns False

Return type bool

class pyModelChecking.CTLS.language.**PathQuantifier**(*phi*)

Bases: *pyModelChecking.CTLS.language.StateFormula*

A class to represent the path quantifiers A or E.

class pyModelChecking.CTLS.language.**R**(**phi*)

Bases: *pyModelChecking.CTLS.language.TemporalOperator*, pyModelChecking.language.AlphabeticSymbol

get_equivalent_restricted_formula()

Return an equivalent formula in the restricted syntax.

This method returns a formula that avoids A, F, R, $\wedge$ and $\rightarrow$ and is equivalent to this formula.

Returns a formula that avoids A, F, R, $\wedge$ and $\rightarrow$ and is equivalent to this formula

Return type CTLS.Not

symbols = ['R']

class pyModelChecking.CTLS.language.**StateFormula**(**phi*)

Bases: *pyModelChecking.CTLS.language.PathFormula*

A class representing CTL* state formulas.

is_a_state_formula()

Returns True if and only if the object has type *StateFormula*.

This method returns True if and only if the current object has type `CTLS.StateFormula`. Since this is a method of the class `CTLS.StateFormula`, it always returns True.

Parameters **self** (*CTLS.StateFormula*) – this formula

Returns True

Return type bool

class pyModelChecking.CTLS.language.**TemporalOperator**(**phi*)

Bases: *pyModelChecking.CTLS.language.PathFormula*

A class to represent temporal operators such as R or X.

class pyModelChecking.CTLS.language.**U**(**phi*)

Bases: *pyModelChecking.CTLS.language.TemporalOperator*, pyModelChecking.language.AlphabeticSymbol

get_equivalent_restricted_formula()

Return an equivalent formula in the restricted syntax.

This method returns a formula that avoids A, F, R, $\wedge$ and $\rightarrow$ and is equivalent to this formula.

Returns a formula that avoids A, F, R, $\wedge$ and $\rightarrow$ and is equivalent to this formula

Return type CTLS.U

symbols = ['U']

class pyModelChecking.CTLS.language.**X**(**phi*)

Bases: *pyModelChecking.CTLS.language.TemporalOperator*, pyModelChecking.language.AlphabeticSymbol

get_equivalent_restricted_formula()
Return an equivalent formula in the restricted syntax.

This method returns a formula that avoids A, F, R, $\wedge$ and $\rightarrow$ and is equivalent to this formula.

Returns a formula that avoids A, F, R, $\wedge$ and $\rightarrow$ and is equivalent this formula

Return type CTLS.X

symbols = ['X']

Model Checking

pyModelChecking.CTLS.model_checking.**modelcheck**(*kripke*, *formula*, *parser=None*, *F=None*)
Model checks any CTL* formula on a Kripke structure.

This method performs CTL* model checking of a formula on a given Kripke structure.

Parameters

- **kripke** (Kripke) – a Kripke structure.
- **formula** (*a type castable in a CTLS.Formula or a string representing a CTLS formula*) – the formula to model check.
- **parser** (*CTLS.Parser*) – a parser to parse a string into a CTLS.Formula.
- **F** (*Container*) – a list of fair states

Returns a list of the Kripke structure states that satisfy the formula.

3.2.3 CTL sub-module API

It represents *CTL formulas* and provides model checking methods for them.

Language

class pyModelChecking.CTL.language.**A**(*phi*)
Bases: *pyModelChecking.CTLS.language.A*, *pyModelChecking.CTL.language.StateFormula*

A class representing CTL A-formulas.

get_equivalent_non_fair_formula(*fairAP*)

get_equivalent_restricted_formula()
Return a equivalent formula in the restricted syntax.

This method returns a formula that avoids $\wedge$, $\rightarrow$, A, F, and R and is equivalent to this formula.

Parameters **self** (*CTL.E*) – this formula

Returns a formula that avoids $\wedge$, $\rightarrow$, A, F, and R and is equivalent to this formula

Return type CTL.StateFormula

pyModelChecking.CTL.language.**AF**(*psi*)
A shortcut to build $E(X(\psi))$.

This method returns the formula $E(X(\psi))$ where ψ is the method parameter.

Parameters **psi** (*CTL.StateFormula*) – a state formula

Returns the formula $E(X(\psi))$

Return type CTL.StateFormula

`pyModelChecking.CTL.language.`**`AG`**(*psi*)

A shortcut to build $A(G(\psi))$.

This method returns the formula $A(G(\psi))$ where ψ is the method parameter.

Parameters **psi** (*CTL.StateFormula*) – a state formula

Returns the formula $A(G(\psi))$

Return type CTL.StateFormula

`pyModelChecking.CTL.language.`**`AR`**(*psi*, *phi*)

A shortcut to build $A(R(\psi, \phi))$.

This method returns the formula $A(R(\psi, \phi))$ where ψ and ϕ are the method parameters.

Parameters

- **psi** (*CTL.StateFormula*) – a state formula
- **phi** (*CTL.StateFormula*) – a state formula

Returns the formula $A(R(\psi, \phi))$

Return type CTL.StateFormula

`pyModelChecking.CTL.language.`**`AU`**(*psi*, *phi*)

A shortcut to build $A(U(\psi, \phi))$.

This method returns the formula $A(U(\psi, \phi))$ where ψ and ϕ are the method parameters.

Parameters

- **psi** (*CTL.StateFormula*) – a state formula
- **phi** (*CTL.StateFormula*) – a state formula

Returns the formula $A(U(\psi, \phi))$

Return type CTL.StateFormula

`pyModelChecking.CTL.language.`**`AX`**(*psi*)

A shortcut to build $A(X(\psi))$.

This method returns the formula $A(X(\psi))$ where ψ is the method parameter.

Parameters **psi** (*CTL.StateFormula*) – a state formula

Returns the formula $A(X(\psi))$

Return type CTL.StateFormula

class `pyModelChecking.CTL.language.`**`And`**(**phi*)

Bases: *pyModelChecking.CTLS.language.And*, *pyModelChecking.CTL.language.StateFormula*

A class representing CTL conjunctions.

class `pyModelChecking.CTL.language.`**`AtomicProposition`**(*name*)

Bases: *pyModelChecking.CTLS.language.AtomicProposition*, *pyModelChecking.CTL.language.StateFormula*

A class representing CTL atomic propositions.

class pyModelChecking.CTL.language.**Bool**(*value*)

Bases: *pyModelChecking.CTLS.language.Bool*, *pyModelChecking.CTL.language.StateFormula*

A class representing CTL Boolean atomic propositions.

class pyModelChecking.CTL.language.**E**(*phi*)

Bases: *pyModelChecking.CTLS.language.E*, *pyModelChecking.CTL.language.StateFormula*

A class representing CTL E-formulas.

get_equivalent_non_fair_formula(*fairAP*)

get_equivalent_restricted_formula()

Return a equivalent formula in the restricted syntax.

This method returns a formula that avoids $\wedge$, $\rightarrow$, A, F, and R and is equivalent to this formula.

Parameters **self** (*CTL.E*) – this formula

Returns a formula that avoids $\wedge$, $\rightarrow$, A, F, and R and is equivalent to this formula

Return type CTL.StateFormula

pyModelChecking.CTL.language.**EF**(*psi*)

A shortcut to build $E(F(\psi))$.

This method returns the formula $E(F(\psi))$ where ψ is the method parameter.

Parameters **psi** (*CTL.StateFormula*) – a state formula

Returns the formula $E(F(\psi))$

Return type CTL.StateFormula

pyModelChecking.CTL.language.**EG**(*psi*)

A shortcut to build $E(G(\psi))$.

This method returns the formula $E(G(\psi))$ where ψ is the method parameter.

Parameters **psi** (*CTL.StateFormula*) – a state formula

Returns the formula $E(G(\psi))$

Return type CTL.StateFormula

pyModelChecking.CTL.language.**ER**(*psi*, *phi*)

A shortcut to build $E(R(\psi, \phi))$.

This method returns the formula $E(R(\psi, \phi))$ where ψ and ϕ are the method parameters.

Parameters

- **psi** (*CTL.StateFormula*) – a state formula
- **phi** (*CTL.StateFormula*) – a state formula

Returns the formula $E(R(\psi, \phi))$

Return type CTL.StateFormula

pyModelChecking.CTL.language.**EU**(*psi*, *phi*)

A shortcut to build $E(U(\psi, \phi))$.

This method returns the formula $E(U(\psi, \phi))$ where ψ and ϕ are the method parameters.

Parameters

- **psi** (*CTL.StateFormula*) – a state formula
- **phi** (*CTL.StateFormula*) – a state formula

Returns the formula $E(U(\psi, \phi))$

Return type CTL.StateFormula

`pyModelChecking.CTL.language.`**EX**(*psi*)

A shortcut to build $E(X(\psi))$.

This method returns the formula $E(X(\psi))$ where ψ is the method parameter.

Parameters **psi** (*CTL.StateFormula*) – a state formula

Returns the formula $E(X(\psi))$

Return type CTL.StateFormula

class `pyModelChecking.CTL.language.`**F**(**phi*)

Bases: *pyModelChecking.CTLS.language.F*, *pyModelChecking.CTL.language.PathFormula*

A class representing CTL F-formulas.

class `pyModelChecking.CTL.language.`**Formula**(**phi*)

Bases: *pyModelChecking.CTLS.language.Formula*

A class representing CTL formulas.

class `pyModelChecking.CTL.language.`**G**(**phi*)

Bases: *pyModelChecking.CTLS.language.G*, *pyModelChecking.CTL.language.PathFormula*

A class representing CTL G-formulas.

class `pyModelChecking.CTL.language.`**Imply**(**phi*)

Bases: *pyModelChecking.CTLS.language.Imply*, *pyModelChecking.CTL.language.StateFormula*

A class representing CTL implications.

class `pyModelChecking.CTL.language.`**Not**(**phi*)

Bases: *pyModelChecking.CTLS.language.Not*, *pyModelChecking.CTL.language.StateFormula*

A class representing CTL negations.

class `pyModelChecking.CTL.language.`**Or**(**phi*)

Bases: *pyModelChecking.CTLS.language.Or*, *pyModelChecking.CTL.language.StateFormula*

A class representing CTL disjunctions.

class `pyModelChecking.CTL.language.`**PathFormula**(**phi*)

Bases: *pyModelChecking.CTL.language.Formula*, *pyModelChecking.CTLS.language.PathFormula*

A class representing CTL* path formulas.

class `pyModelChecking.CTL.language.`**R**(**phi*)

Bases: *pyModelChecking.CTLS.language.R*, *pyModelChecking.CTL.language.PathFormula*

A class representing CTL R-formulas.

class pyModelChecking.CTL.language.**StateFormula**(**phi*)
Bases: *pyModelChecking.CTL.language.Formula*, *pyModelChecking.CTLS.language.StateFormula*

A class representing CTL* state formulas.

class pyModelChecking.CTL.language.**U**(**phi*)
Bases: *pyModelChecking.CTLS.language.U*, *pyModelChecking.CTL.language.PathFormula*

A class representing CTL U-formulas.

class pyModelChecking.CTL.language.**X**(**phi*)
Bases: *pyModelChecking.CTLS.language.X*, *pyModelChecking.CTL.language.PathFormula*

A class representing CTL X-formulas.

Model Checking

pyModelChecking.CTL.model_checking.**modelcheck**(*kripke*, *formula*, *parser=None*, *F=None*)
Model checks any CTL formula on a Kripke structure.

This method performs CTL model checking of a formula on a given Kripke structure.

Parameters

- **kripke** (Kripke) – a Kripke structure.
- **formula** (*a type castable in a CTL.Formula or a string representing a CTL formula*) – the formula to model check.
- **parser** (*CTL.Parser*) – a parser to parse a string into a CTL.Formula.
- **F** (*Container*) – a list of fair states

Returns a list of the Kripke structure states that satisfy the formula.

3.2.4 LTL sub-module API

It represents *LTL formulas* and provides model checking methods for them.

Language

class pyModelChecking.LTL.language.**A**(**phi*)
Bases: *pyModelChecking.LTL.language.StateFormula*, *pyModelChecking.CTLS.language.A*

A class representing LTL A-formulas.

class pyModelChecking.LTL.language.**And**(**phi*)
Bases: *pyModelChecking.LTL.language.PathFormula*, *pyModelChecking.CTLS.language.And*

A class representing LTL conjunctions.

class pyModelChecking.LTL.language.**AtomicProposition**(*name*)
Bases: *pyModelChecking.CTLS.language.AtomicProposition*, *pyModelChecking.LTL.language.PathFormula*

A class representing LTL atomic propositions.

class pyModelChecking.LTL.language.**Bool**(*value*)
Bases: *pyModelChecking.CTLS.language.Bool*, *pyModelChecking.LTL.language.PathFormula*

A class representing LTL Boolean atomic propositions.

class pyModelChecking.LTL.language.**F**(**phi*)
Bases: *pyModelChecking.LTL.language.PathFormula*, *pyModelChecking.CTLS.language.F*

A class representing LTL F-formulas.

class pyModelChecking.LTL.language.**Formula**(**phi*)
Bases: *pyModelChecking.CTLS.language.Formula*

A class representing LTL formulas.

class pyModelChecking.LTL.language.**G**(**phi*)
Bases: *pyModelChecking.LTL.language.PathFormula*, *pyModelChecking.CTLS.language.G*

A class representing LTL G-formulas.

class pyModelChecking.LTL.language.**Imply**(**phi*)
Bases: *pyModelChecking.LTL.language.PathFormula*, *pyModelChecking.CTLS.language.Imply*

A class representing LTL implications.

class pyModelChecking.LTL.language.**Not**(**phi*)
Bases: *pyModelChecking.LTL.language.PathFormula*, *pyModelChecking.CTLS.language.Not*

A class representing LTL negations.

class pyModelChecking.LTL.language.**Or**(**phi*)
Bases: *pyModelChecking.LTL.language.PathFormula*, *pyModelChecking.CTLS.language.Or*

A class representing LTL disjunctions.

class pyModelChecking.LTL.language.**PathFormula**(**phi*)
Bases: *pyModelChecking.LTL.language.Formula*, *pyModelChecking.CTLS.language.PathFormula*

A class representing LTL path formulas.

class pyModelChecking.LTL.language.**R**(**phi*)
Bases: *pyModelChecking.LTL.language.PathFormula*, *pyModelChecking.CTLS.language.R*

A class representing LTL R-formulas.

class pyModelChecking.LTL.language.**StateFormula**(**phi*)
Bases: *pyModelChecking.LTL.language.Formula*, *pyModelChecking.CTLS.language.StateFormula*

A class representing LTL state formulas.

class pyModelChecking.LTL.language.**U**(**phi*)
Bases: *pyModelChecking.LTL.language.PathFormula*, *pyModelChecking.CTLS.language.U*

A class representing LTL U-formulas.

class `pyModelChecking.LTL.language.`**`X`**(**phi*)

Bases: *pyModelChecking.LTL.language.PathFormula*, *pyModelChecking.CTLS.language.X*

A class representing LTL X-formulas.

Model Checking

`pyModelChecking.LTL.model_checking.`**`modelcheck`**(*kripke*, *formula*, *parser=None*, *F=None*)

Model checks any LTL formula on a Kripke structure.

This method performs LTL model checking of a formula on a given Kripke structure.

Parameters

- **kripke** (`Kripke`) – a Kripke structure.
- **formula** (*a type castable in a LTL.Formula or a string representing a LTL formula*) – the formula to model check.
- **parser** (*LTL.Parser*) – a parser to parse a string into a LTL.Formula.
- **F** (*Container*) – a list of fair states

Returns a list of the Kripke structure states that satisfy the formula.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Bibliography

[P77] A. Pnueli. “The temporal logic of programs.” In Proceedings of the 18th Annual Symposium of Foundations of Computer Science (FOCS), 1977, 46-57

[BMP83] M. Ben-Ari, Z. Manna, A. Pnueli. The temporal logic of branching time. Acta Informatica 20(1983): 207-226

[CE81] E. M. Clarke, E. A. Emerson. “Design and synthesis of synchronization skeletons using branching time temporal logic.” In Logic of Programs: Workshop. LNCS 131. Springer, 1981.

[CE80] E. M. Clarke, E. A. Emerson. “Characterizing correcteness properties of parallel programs using fixpoints.” In Automata, Languages, and Programming. LNCS 85:169-181. Springer 1980.

[CES86] E. M. Clarke, E. A. Emerson, A. P. Sistla. “Automatic verification of finite-state concurrent systems using temporal logic specifications.” ACM Transactions on Programming Languages and Systems 8(2): 244-263. 1986.

[CGP00] E. M. Clarke, O. Grumberg, D. A. Peled. “Model Checking” MIT Press, Cambridge, MA, USA. 2000.

[Bryant86] Randal E. Bryant. “Graph-Based Algorithms for Boolean Function Manipulation”. IEEE Transactions on Computers, C-35(8):677–691, 1986.

[ns94] E. Nuutila and E. Soisalon-Soinen. “On finding the strongly connected components in a directed graph.”, Information Processing Letters 49(1): 9-14, (1994)

Python Module Index

c

g

k

l

p

Index

A

B

C

D

E

F

G

I

K

L

M

N

O

P

R

S

T

U

W

X