pyModelChecking Documentation

Release 0.2.0

Alberto Casagrande

Apr 19, 2019

Contents

pyModelChecking is a simple Python model checking package. Currently, it is able to represent *Kripke structures*, *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 $s' \in S$
- $L : S \to AP$ 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 Temporal Logics

1.2.1 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]). 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 **A** ("for all paths") and **E** ("for some path") and the five temporal operators **X** ("at the next step"), **G** ("globally"), F ("in the future"), **U** ("until"), and **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 $\bot$
- an atomic proposition

- $\neg\varphi_1$, $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$, 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 are given with respect to a *Kripke structure*. If K is a Kripke structure, s one of its states, and φ a state formula, we write $K, s \models \varphi$ (to be read "K and s satisfy φ") 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]).

1.2.2 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 $\bot$
- an atomic proposition
- $\neg\varphi_1$, $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$, 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:

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

1.2.3 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 $\bot$
- an atomic proposition
- $\neg\varphi_1$, $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$, 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.formula import *
>>> AtomicProposition('p')

p

>>> Bool(True)

True
```

Moreover, the *pyModelChecking.language* sub-module implements the logic operators $\land$, $\lor$, $\rightarrow$ and $\neg$ by mean of the classes `pyModelChecking.formula.And`, `pyModelChecking.formula.Or`, `pyModelChecking.formula.Imply` and `pyModelChecking.formula.Not`, respectively. These classes automatically wrap strings and Boolean values as objects of the classes `pyModelChecking.formula.AtomicProposition` and `pyModelChecking.formula.Bool`, respectively.

```
>>> 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)
```

For user convenience, the function `pyModelChecking.formula.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)

>>> LNot(LNot(f))

not (p and not q)
```

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 ref:*CTLS sub-module<ctls_api>* 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))))))
```

In order to simplify the use of the library, a parsing class `pyModelChecking.CTLS.Parser`: has been implemented. Its objects read a formula from a string and, when it is possible, translate it into a corresponding `pyModelChecking.CTLS.Formula` objects.

```
>>> parser = Parser()
>>> psi_str = 'A G ((not Close and Start) --> ' +
...           'A(G(not Heat) or F(not Error)))'
>>> psi = parser(psi_str)
>>> psi

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

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.

3.1.2 Kripke API

It is used to define *Kripke structures*.

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 CTLS sub-module API

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

Language

Model Checking

3.2.2 CTL sub-module API

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

Language

Model Checking

3.2.3 LTL sub-module API

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

Language

Model Checking

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.