# 1 Introduction

This document specifies the most essential interfaces inside the TEMA test engine.

# 2 Model interface

The model interface hides test modelling formalism. This allows using several modelling formalisms with the test engine and even mixing them.

## 2.1 Classes and methods

`MODEL` class

> `loadFromFile`(filelikeobj)
> This method is called before any other methods.
> **Args:** filelikeobj is usually `FILE` object. If not, it should at least implement the methods that are used for reading the "file" contents in the `loadFromFile` (typically `read` or `readline`).
> **Returns:** -
> **Exceptions:** may pass IOErrors and other problems in reading the file.
>
> `getInitialState()`
>
> **Args:** -
> **Returns:** a `STATE` object that represents the starting state of the model.
> **Exceptions:** -
>
> `getActions()`
>
> **Args:** -
> **Returns:** a list of `ACTION` objects. The list contains all possible actions that may in the model.
> **Exceptions:** -

`STATE` class — represents a state of the model

> `getOutTransitions`()
> **Args:** -
> **Returns:** an iterable set of `TRANSITION` objects. The actions of the transitions specify the events that can occur in this state, and the destination states of the transitions specify the states after the events.
> **Exceptions:** -
>
> `equals`(`STATE` object)

**Args:** a `STATE` object
**Returns:** a boolean value that is true iff the state given as a parameter represents the same state as this object.
**Exceptions:** -

`__str__`()

`TRANSITION` class — represents a transition from a state to another in the model

`getAction`()
**Args:** -
**Returns:** the `ACTION` object associated with the transition.
**Exceptions:** -

`getSourceState`()
**Args:** -
**Returns:** the `STATE` object from which the transition can be executed. The state is the only state that returns the transition when its `getOutTransitions` is called.
**Exceptions:** -

`getDestState`()
**Args:** -
**Returns:** the `STATE` object to which the transition leads.
**Exceptions:** -

`equals`(`TRANSITION` object)
**Args:** a `TRANSITION` object
**Returns:** a boolean value that is true iff the transition given as a parameter represents the same transition as this object.
**Exceptions:** -

`__str__`()

`ACTION` class — represents an action

`toString`()
**Args:** -
**Returns:** a string representation of the action.
**Exceptions:** -

`isKeyword`()
**Args:** -
**Returns:** a boolean that is true iff the action is a keyword, that is, it should be sent to the SUT when the test is executed. Non-keyword actions can be exected without notifying the SUT.
**Exceptions:** -

`equals`(`ACTION` object)

**Args:** an ACTION object
**Returns:** a boolean value that is true iff the action given as a parameter represents the same action as this object.
**Exceptions:** -

`__str__()`

## 2.2 Example of usage

LSTS model interface:

```
m = lstsmodel.Model()
m.loadFromFile( file("exampledata/BobTelephone.lsts") )

istate = m.getInitialState()
print "The initial state of BobTelephone.lsts is: %s" % (str(istate),)

for t in istate.getOutTransitions():
    print "Action %s can be executed in the initial state" % (t.getAction(),)
```

## 2.3 Available implementations

- `lstsmodel.py` implements model interface for LSTS files.
- `parallelmodel.py` implements model interface for parallel composition of any objects that implement the model interface.
- `parallellstsmodel.py` implements model interface for parallel composition of LSTSes with `parallelmodel.py`.
- `tracelogmodel.py` implements model interface for test logs. This can be used for repeating or investigating a previously run test: the test log can be used as a linear test model in the new test run or simulation / visualisation tool.
- (not in real use at the time of writing this document) `prepostmodel.py` implements model interface for a precondition / postcondition formalism that is strongly based on Python.

# 3 Coverage interface

Coverage interface implements a function that maps sequences of executed transitions to objects that tell how much of the required coverage has been achieved.

## 3.1 Classes and methods

`Coverage` class — implements a test generation algorithm

`__init__`([covspec, [model=`Model` object]])

**Args:** Optional argument covspec, passes coverage language statement when requirement coverage in `clparser.py` is used. The other optional argument is the `Model` object.
**Returns:** -
**Exceptions:** -

> TODO: initialization of the coverage differs from other plugins. Testengine should be changed to accept arguments `--coverage=requirementcoverage --coverage-args="actions start_aw.* then actions kw.*"` and `--coverage=statecoverage`. The arguments now given to the constructor should be given to the coverage with the standard setParameter() function calls.

`markExecuted`(`Transition` object)

**Args:** `Transition` object that is considered executed.
**Returns:** -
**Exceptions:** -

`getPercentage`()

**Args:** -
**Returns:** a coverage status object whose equality to 1.0 must be testable and which should be comparable to other coverage status objects. This can be a floating point number, for example.
**Exceptions:** -

`pickDataValue`(set of possible data values)

If the coverage prefers using some data values over other, this method returns the value in the set. Otherwise it should return None. The returned value can be considered to be covered.
**Args:** a set of values that can be converted to string
**Returns:** None or a chosen value converted to string
**Exceptions:** -

`push`()

Stores the current coverage data to a stack. If the data is stored, `markExecuted`() and `pickDataValue`() can be called without affecting the stored data. Calling `pop`() restores the state of the previous `push`(). Push and pop calls can be nested. The purpose of these methods is to save memory, because test generation

algorithms can avoid making of deep-copies of coverage objects. Coverage class implementations can use more sophisticated data structures to store the current state.
**Args:** -
**Returns:** -
**Exceptions:** -

`pop()`
Restores the coverage object to the stored state that is on the top of the stack and removes the state from the stack.
**Args:** -
**Returns:** -
**Exceptions:** IndexError (pop from emtpy list)

## 3.2 Example of usage

```
# init
c = coveragemodule.Coverage("actions .*", modelobject)
# see which action gives the best coverage
scores = []
current_state = modelobject.getInitialState()
for t in current_state.getOutTransitions():
    c.push()
    c.markExecuted(t)
    scores.append((c.getPercentage(), t.getAction()))
    c.pop()
scores.sort()
print "best score: %s with action %s" % (scores[-1],)
```

Yet this **would** be better init:

```
c = coveragemodule.Coverage()
c.setParameter("model", modelobject)
c.setParameter("requirement", "actions start_aw.* then actions kw.*")
```

## 3.3 Command line usage

Command line arguments related to coverage module

- `coverage` The coverage module. (If not given, the proper module is guessed based on the `coveragereq`)
- `coveragereq` Coverage language expression (REQUIRED)
- `coveragereq-args` Optional arguments for coverage module

For example:

```
--coverage='clparser' \
--coveragereq='actions .*' \
--coveragereq-args=''
```

### 3.4 Available implementations

- `clparser` Coverage language. Example of the language: `"actions .*"`
- `findnewcoverage` Implements another ("findnew") coverage language. Example of the language: `"findnew state"`
- `altercoverage` Coverage module implementing "find a new action word on another application" kind of heuristic. Example of the accepted language: `"app1 app2 app3 random"`. The last word in the language is either a number $N = 2, 3, ...$ for $N$-coverage of applications or `random`.

## 4 Guidance interface

Test generation algorithms are written behind the guidance interface. The algorithms are given a test model (that implements the model interface) and a coverage object (that implements the coverage interface) as input. In addition to them, several algorithm-specific parameters can be passed to the object from the command line.

### 4.1 Classes and methods

GUIDANCE class — implements a test generation algorithm

`setTestModel`(MODEL object)
**Args: Returns: Exceptions:**

`addRequirement`(COVERAGE object)
**Args: Returns: Exceptions:**

`prepareForRun`()
**Args: Returns: Exceptions:**

`suggestAction`(STATE object)
**Args:** a STATE object from the model interface.
**Returns:** ACTION object (an action must always be returned)
**Exceptions:**

`markExecuted`(TRANSITION object)
**Args: Returns: Exceptions:**

### 4.2 Example of usage

Command line arguments

```
--guidance-args='lookahead:15,randomseed:hop'
```

cause the test engine core to call

1. `setParameter`("lookahead","15")
2. `setParameter`("randomseed","hop").

## 4.3 Available implementations

- `randomguidance.py` implements a guidance that returns a random transition among the transitions leaving the given state.

- `wrandomguidance.py` weighted random guidance. When transition objects in the model interface implement optional `getWeight`() method, this guidance algorithm can be used for selecting randomly an outgoing transition where the probability of becoming selected depends on the weight of the transition.

- `gameguidance.py` calculates $n$ steps forward from the given state and returns the transition that leads to the best coverage after the $n$ steps.

- `gameguidance-t.py` threading version of the previous: the possible steps are being continuously calculated in a separate thread in the background.

- `greedyguidance.py` finds the shortest path improving coverage. Requires a `getExecutionHint` method from its coverage module.

- `weightguidance.py` finds the best path up to a given search depth. Uses `transitionPoints` method of coverage object if it is available. Similar to `gameguidance` with a few extra parameters.

- `tabuguidance.py` tries to execute actions/states/transitions that haven't been executed before.