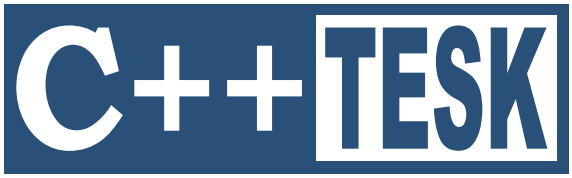

C++TESK Testing ToolKit: **Test Engines**

Version 1.0, 04/09/2013

C++TESK Testing ToolKit can be downloaded from the page http://forge.ispras.ru/projects/cpptesk-toolkit.

Please let us know about your proposals and problems while using C++TESK Testing ToolKit sending them to cpptesk-support@ispras.ru. The forum http://hw-forum.ispras.ru can be also used for such a purpose.

Content

Introduction

This document describes *test engines* included into library of C++TESK Testing ToolKit (hereafter C++TESK). *Test engine* is a core of *stimulus generator*, which is a component of *test system* being responsible for construction of stimulus sequence (*test actions*) for *target system*. Stimuli can be represented as parameterized target system operation calls. Sequence of stimuli is called *test sequence*. *Test scenario*, being input information for test engine, is a high-level specification of test defining available for test stimuli. Test sequence is constructed as a result of interpretation of given scenario by test engine. It should be noticed that all test engines have the same interface which let different test engines be used for execution of given test scenario.

Library of C++TESK includes two test engines: `fsm` (*Finite State Machine*) and `rnd` (*Random*), located in namespace `cpptesk::ts::engine`. The former (`fsm`) makes traversing of finite state machine *state graph* described in implicit form in test scenario. The criterion of test completeness in this case is visiting of all states reachable from initial state (it means also traversing of all arcs outgoing from reachable states). Test engine `rnd` constructs test sequence *randomly* selecting random either stimulus or set of available stimuli at each work step. Test is finished when given work steps are passed. More detailed information about test engines can be found in correspondent chapters of this document.

Before reading the rest part of this document, it is recommended to read about development of test scenario by means of C++TESK. This information is contained, e.g., by document «*C++TESK Hardware Edition: Quick Reference*» (chapter «*Development of test scenario*»).

Test scenario structure

From test engine point of view, test scenario consists of two main parts: (1) *stimulus iterator* and (2) *state calculation function*.

Stimulus iterator

Stimulus iterator is set by *scenario methods* of scenario class. Each scenario method iterates parameters of some stimulus using typical loop-constructions being available in C++ (`for`, `while` etc.). Iterations are performed by means of *iteration variables* which are the fields of *iteration context* being a parameter of scenario method. Iteration variables can be accessed by means of macro `CPPTESK_ITERATION_VARIABLE(`*`name`*`)`. Macro `CPPTESK_ITERATION_BEGIN` precedes iterations and macro `CPPTESK_ITERATION_END` finishes them. Implementation of *stimulus application to target system* is located inside of set of loops in block `CPPTESK_ITERATION_ACTION{...}`. Stimulus application finishes by calling macro `CPPTESK_ITERATION_YIELD(`*`verdict`*`)`.

Typical scenario method has the following structure.

```
bool MyScenario::ScenarioMethod(AbcCtx &ctx) {
    int &a = CPPTESK_ITERATION_VARIABLE(a);
    ...
    int &z = CPPTESK_ITERATION_VARIABLE(z);

    CPPTESK_ITERATION_BEGIN
    for(a = 0; a < Na; a++)
    ...
    for(z = 0; z < Nz; z++) {
        CPPTESK_ITERATION_ACTION {
            ...
            CPPTESK_ITERATION_YIELD(...);
        }
    }
    CPPTESK_ITERATION_END
```

```
}
```

Scenario method *ScenarioMethod* for given reference model state (specification state) *ModelState* produces stimulus set, where each stimulus is identified by integer numbers from range [0, N_{max}(*ScenarioMethod*, *ModelState*)-1], where N_{max}(*ScenarioMethod*, *ModelState*) is the number of executions of block `CPPTESK_ITERATION_ACTION` for scenario methods of given reference model state.

Aggregated stimulus iterator is obtained by "aggregation" of iterators defined in scenario methods. Test scenario stimulus iterator describes a set of all possible stimuli and introduces current test scenario-wide system of stimulus identification. If reference model state *ModelState* is fixed, stimuli are identified by integer numbers from range [0, N_{max}(*ModelState*)-1], where $N_{max}(ModelState)=\sum_{i=0,n-1}\{N_{max}(ScenarioMethod_i, ModelState)\}$. Numeration of stimuli accounts order of scenario method registration.

State calculation function

State calculation function is defined in *state calculation method* of scenario class. Type of return value can be scalar (`int`, `float` и т.п.) or of string type (`std::string`). Return value is interpreted as reference model state at some level of abstraction.

State calculation function is unnecessary test scenario component and used only by test engine `fsm`.

Test engines

Test engines are controlled by command line parameters; some of them are common for all engines.

- `--cpu` — running test host device identifier[1];
- `--length` — length of test sequence[2];
- `--print-progress` — print of test progress.

Common command line parameters

Parameter `cpu`

Parameter `--cpu` *`целое_число`* sets identifier of host device (computer, microprocessor or core) which runs current test. This parameter is used only in case of distributed among several host devices testing. The main purpose of this parameter is varying of test executions at different host devices. E.g., this parameter is used for random seed setting and in choosing of the following stimulus (graph arc) by test engine `fsm`.

Parameter `length`

Parameter `--length` *`integer`* restricts the length of test sequence generated by test engine. When the length is equal to the given number, test is finished.

Option `print-progress`

Option `--print-progress` turns on print of test execution progress (state graph exploration).

---

[1] This parameter is sensible only in distributed testing.

[2] In current version of C++TESK Testing ToolKit test engine `fsm` ignores parameter `--length`.

Default values

Option `--print-progress` is turned off by default, parameters `--cpu` and `--length` have the following values.

```
--cpu 0 --length 1000
```

Test engine `fsm`

Test engine `fsm` generates test sequence by finite state machine state graph exploration. The graph is represented implicitly in test scenario. The criterion of test completeness is visiting of all states being reachable from the initial state and traversing all arcs issuing from them.

Brief algorithm description

Test engine `fsm` works in the following way. At each step it has current state computed. There are two ways: (1) this state has non traversed arcs[3] and (2) all arcs issuing from this state have been traversed. It the first case test engine selects one of the non traversed arcs, applies correspondent stimulus to the target system and test keeps on going. The second case has two alternatives: (2.1) there are other states (from having been visited) with non traversed arcs, and (2.2) all known arcs have been traversed. In the first case test engine finds the way to the state with non traversed arcs. Eventually situation (2.1) becomes situation (1). In the second case test is finished.

Command line parameters

Test engine `fsm` supports the following command line options.

- `--randomize` — randomization of arc selection;
- `--nondeterministic` — support of nondeterministic arcs;
- `--dump-fsm` — saving of state graph in file after test finalizing:
  - `--full-graph` — saving of the whole graph;
  - `--forward-tree` — saving of spanning tree.

Option `randomize`

Option `--randomize` turns on randomized selection of the following arc. It means that if test engine has a choice which arc should be selected, test engine choose arc randomly. If this option is not set, arcs are selected in the order prescribed by their order in test scenario.

Option `nondeterministic`

Option `--nondeterministic` turns on support of *nondeterministic* graphs, i.e. graphs where several arcs labeled by the same stimulus can issue from the same state. The main difficulty in traversing of nondeterministic graphs is absence of possibility of determination which arc has been traversed by applying some or other stimulus. It makes finding of paths with non traversed arcs more difficult.

Setting of option `--nondeterministic` allows usage of nondeterministic arc during path finding (increasing class of supported graphs). If this option is not set, paths are constructed via deterministic arcs.

[3] It is more correct to speak about *stimuli* not *arcs* as in case of nondeterministic graph one stimulus can correspond to several arcs issuing from the same state.

Option `dump-fsm`

Option `--dump-fsm` is used for saving of state graph in file after test finalizing. File has fixed name `fsm.gv`; in this file data are represented in Graphviz format, which is an open source graph visualization package (http://www.graphviz.org).

Option `full-graph`

Option `--full-graph` (set together with `--dump-fsm`) saves state graph as a whole.

Notice: options `--full-graph` and `--forward-tree` are incompatible.

Option `forward-tree`

Option `--forward-tree` (set together with`--dump-fsm`) saves only state graph spanning tree.

Notice: options `--forward-tree` and `--full-graph` are incompatible.

Default values

Options `--randomize`, `--nondeterministic`, and `--dump-fsm` are not set by default.

If option `--dump-fsm`, is set, state graph is saved as a whole by default.

Test engine `rnd`

Test engine `rnd` constructs test sequence *randomly* selecting random stimulus or random stimulus set from the list of allowed in test scenario stimuli at each work step. Test is finished when given number of stimulus is called.

Brief algorithm description

There are two modes of test engine `rnd` work. The first one is *sequential* (see chapter «*Option `sequential`*») and *parallel* (see chapter «*Option `parallel`*»).

In sequential mode all the stimuli, set up by test action iterator, are equitable. At each work step, identifier of stimulus is randomly selected from range [0, N_{max}(*ModelState*)-1], and correspondent stimulus is applied.

In parallel mode stimulus with identifier 0 has a special meaning playing role of delay stimulus. All the other stimuli are not allowed to shift simulation time (they are "immediate" actions without waiting of reactions for them.). In this mode at each work step random stimulus sequence (*multi stimulus*) is created. This sequence is finished by stimulus with identifier 0. In the other words, several stimuli are applied in parallel, and then some delay happens.

Size of multi stimulus depends on *load mode*. Test engine `rnd` supports four load modes.

- *minimum load mode* (see chapter «*Option `min-load`*»);
- *maximum load mode* (see chapter «*Option `max-load`*»);
- *fixed load mode* (see chapter «*Parameter `fix-load`*»);
- *variable load mode* (see chapter «*Option `var-load`*»).

The criterion of test completeness is passing of certain number of test steps (see chapter «*Parameter `length`*»).

Command line parameters

Test engine `rnd` supports the following command line parameters.

- `--sequential` — sequential mode;
- `--parallel` — parallel mode:
  - `--min-load` — minimum load mode;
  - `--max-load` — maximum load mode;
  - `--fix-load` — fixed load mode;
  - `--var-load` — variable load mode.

Option `sequential`

Option `--sequential` turns on *sequential mode* of test engine work. In this mode, test engine at each step selects random stimulus from the list of available in test scenario stimuli. Sequential mode is aimed for systems which does not support application of stimuli in parallel mode.

Notice: options `--sequential` and `--parallel` are incompatible.

Option `parallel`

Option `--parallel` turns on *parallel mode* of test engine work. In this mode, test engine at each step generates random stimulus sequence finished with stimulus with identifier 0 — delay stimulus[4]. All the stimuli except stimulus 0 are supposed to run immediately. Despite their attempt to run some or other operation, their real start is performed only after calling stimulus 0 when time is shifted. Test step in parallel mode is schematically showed in figure 1.

$$\ldots \rightarrow \left\{ \begin{array}{c} стимул\ i_1(t) \\ стимул\ i_2(t) \\ \ldots \\ стимул\ i_{n(t)}(t) \end{array} \right\} \rightarrow стимул\, 0\,(задержка) \rightarrow \ldots,\ i_k(t) \neq 0.$$

Figure 1. Test step in parallel mode

Notice: options `--parallel` and `--sequential` are incompatible.

Option `min-load`

Option `--min-load` turns on *minimal* load mode allowing not more than one stimulus at each work step. This mode is typically used during debug of test system.

Option `max-load`

Option `--max-load` turns on maximum load mode to apply all possible stimuli described in test scenario at each work step.

Parameter `fix-load`

Parameter `--fix-load` *`load_percentage`* sets *fixed* load mode to apply certain part of stimuli at each work step. E.g., if `--fix-load 50` is written, half of the whole stimulus set will be applied in parallel at each step.

[4] Typically this is a scenario method without iterations called `nop()` or `delay()`.

Option `var-load`

Option `--var-load` turns on variable load mode to let test system load be smoothly increased from minimum to maximum.

Default values

Option `--sequential` is used by default.

If option `--parallel` is used, variable load mode `--var-load` is used by default.