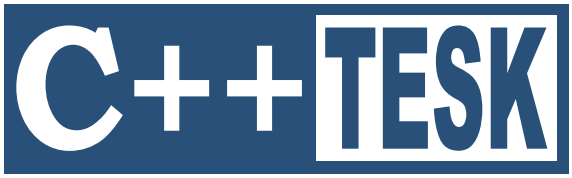

C++TESK Hardware Edition: **Whitepaper**

Version 1.0, 16/09/2011

C++TESK Hardware Edition is a part of C++TESK Testing ToolKit, which 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 Hardware Edition and 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.

Introduction

In this paper, the basic facilities of C++TESK, a C++ based toolkit for simulation-based hardware verification, are described. The toolkit implements the model-based approach to verification of HDL[1]-models, which means that all verification tasks, like stimulus generation, reaction checking and coverage tracking, are carried out employing a design model. The key feature of C++TESK is scalability – the toolkit can handle complex designs by using abstract models and/or parallelizing verification on computer clusters.

Simulation-based verification is known to be the main means for ensuring functional correctness of hardware designs of industrial size and complexity. A lot of methods, tools and technologies have appeared to overcome the ever-growing system complexity and to bring a higher level of automation to verification. C++TESK developed at ISPRAS is one of them. Like all tools intended for creating verification environments, so-called testbenches, it provides facilities for defining stimulus generators, reaction checkers and coverage trackers.

C++TESK is a C++ based toolkit, which implies that testbench components are developed in pure C++ (the toolkit's core is an open-source C++ library). In this regard, it is similar to SystemC, but has a more specialized application domain. The C++ language is used for several reasons. First of all, most engineers are familiar with it. Second, microprocessor instruction set simulators (ISS) are usually written in C/C++ (thus, it is possible to reuse ISS components as reference models for verification). Third, there are many C++ programming tools (compilers, debuggers, profilers, etc.) and libraries (STL, Boost, etc.) that can be used for free.

Besides the usability achieved by employing C++, the toolkit has many advantages comparing with the existing solutions. These advantages include hardware modeling at different abstraction levels (from a functional, untimed level up to a cycle-accurate one), automated generation of stimulus sequences based on state graph exploration (state enumeration), support for used-defined coverage criteria including temporal coverage, verification parallelization on computer clusters (using distributed graph exploration), and some others. The rest of the paper describes the C++TESK facilities more in detail.

Hardware Modeling and Reaction Checking

The central part of C++TESK is a library of hardware modeling primitives. The library allows developing reference models of hardware designs at different abstraction levels (untimed, time-approximate and time-accurate models) and composing complex models from simple ones using data transmission channels (thus, C++TESK supports transaction-level modeling, TLM). Hardware component is modeled as a class declaring input and output interfaces and stimulus processing operations. An example is given below (bold font indicates C++TESK macros).

```
MODEL(MyModel) {
public:
    DECLARE_INPUT   (in_iface);        // input interface(s)
    DECLARE_OUTPUT  (out_iface);       // output interface(s)
    DECLARE_STIMULUS(operation);       // operation(s)
    ...
};
```

Stimulus processing operations are modeled as methods with a fixed signature (an input interface and a message). Within operations, in addition to common C++ statements, special constructs are used to model time and reaction dispatching.

```
DEFINE_STIMULUS(MyModel::operation) {
    START_STIMULUS();                  // starts an operation
```

[1] HDL (Hardware Description Language) — class of languages used for description of hardware designs. Verilog and VHDL are the most famous among them.

```
    ...                                 // emulates a one-cycle
    CYCLE();                            // time delay
    SEND_REACTION(out_iface, out_msg); // produces a model reaction
    STOP_STIMULUS();                    // stops an operation
}
```

Adaptation of a reference model for co-simulation with the design under verification (DUV) is done in a descendant class (so-called model adapter) by defining input and output interface adapters. An input interface adapter (being launched in **START_STIMULUS**) serializes the input message into the input signals distributed in time. An output interface adapter (being launched in **SEND_REACTION**) waits until either the design reaction is detected or time limit is reached and, then, deserializes the output signals into the output message.

Roughly speaking, reaction checking is done as follows. Every time when a model calls **SEND_REACTION**, it puts a model reaction into the reaction queue associated with the corresponding output interface and returns. When a design reaction is received, it should be associated with one of the model reactions stored in the reaction queue (if a model is accurate, the reaction queue should contain exactly one model reaction; for abstract models there can be several reactions though). Choosing a model reaction corresponding to a design reaction is carried out by the output interface's reaction arbiter. As soon as the correspondence is found, the model reaction and design reactions are compared. If they are not equal, the bug is reported.

Reaction arbitration is a powerful technique that makes it possible to use abstract order-inaccurate reference models for simulation-based verification. C++TESK has a library of ready-to-use reaction arbiters covering various verification purposes. The simplest one is a FIFO arbiter, which chooses the first model reaction stored in the reaction queue.

Scenario Description and Stimulus Generation

Verification scenario in C++TESK is specified as a state machine whose state corresponds to abstract state of the DUV, while transitions are stimuli. A special component, called engine or traverser, interprets a scenario and generates a stimulus sequence by exploring the corresponding state graph (the purpose is to try each transition in each state reachable from initial). Scenario is described in a separate class; specifications of transitions are grouped into so-called scenario methods.

```
SCENARIO(MyScenario) {
    MyStateType get_state();            // scenario state
    bool scenario_method(Context &ctx); // scenario method(s)
    ...
    MyModel &duv;                       // model adapter
};
```

Each scenario method should be organized as a co-routine: it iterates stimulus parameters and applies a stimulus. After each invocation it should return control to the engine. Let us consider a scenario method example.

```
bool MyScenario::scenario_method(Context& ctx) {
    IBEGIN // enters an iteration section
    // IVAR(x) accesses iteration variable named x
    for(IVAR(x) = -1; IVAR(x) <= 1; IVAR(x)++) {
        IACTION {
            MyMessage in_msg(x);  // applies a stimulus
            duv.start(&MyModel::operation, duv.in_iface, in_msg);
            YIELD(duv.verdict()); // returns a verdict
        }
    }
    IEND   // exits an iteration section
}
```

It should be noticed that C++TESK graph exploration engine supports non-deterministic state machines, which is especially important when abstract reference models are used (abstraction is a frequent cause of indeterminacy). Besides the graph-based engine, C++TESK has an engine that constructs a sequence by applying the randomization techniques.

Coverage Definition and Tracking

C++TESK supports user-defined test coverage which is described in a reference model and is tracked during simulation. The resulting coverage is summarized in verification reports. Coverage structure is specified using some set of macros. As an example let us consider the sign coverage having three elements (negative, zero and positive).

```
DEFINE_ENUMERATED_COVERAGE(SignCoverage, "Sign coverage", (
    (NEGATIVE, "Negative"), // coverage item: an identifier
    (ZERO,     "Zero"),     // and a human-readable name
    (POSITIVE, "Positive")  // used in coverage reports
));
```

The toolkit implements several operations with coverage structures: aliasing, composition and partial composition. Aliasing constructs coverage with different type of elements, but the same coverage elements (i.e. the identifiers and human readable names are the same). Composition builds Cartesian product of two coverage structures. The composed coverage enumerates elements that are ordered pairs of the elements of the operand coverage structures. Unreachable elements can be excluded from the coverage using the partial composition.

C++TESK also supports defining and tracking temporal coverage, which is specified as a set of temporal sequences. Each sequence defines a pattern of interaction with the DUV (events, their order and delays between them). If the pattern is recognized, then the corresponding situation is covered. A temporal sequence example is given below.

```
// after stimulus S is applied, reactions R1, R2, R1 and R2, should
// appear one after the other with 1-2 cycles delay between them
if_then(S) << any_delay() <<
    (R1 << delay(1, 2) << R2 << delay(1, 2)).repeat(2)
```

Verification Parallelization

A useful facility implemented in C++TESK is that each testbench can be executed on a computer cluster in parallel. The approach significantly speeds up verification, shrinks bug detection time and accelerates the design process in whole. Parallelization is done dynamically without using static information on a verification scenario. From the perspective of engineers parallelization is fully transparent – development of a testbench does not depend on how it will be executed (on one computer, on several computers or on a computer cluster). Moreover, it is not more difficult to launch a testbench in a distributed environment than on a single computer.

The key idea used for parallelization is distributed graph exploration. All testbench instances explore the same state graph and share information about traversed parts of the graph. The engine remains the same, but there are several sources of traversed arcs. Each testbench instance has a build-in component, called synchronizer, responsible for exchanging information with other instances. Synchronizers of all instances are interconnected into a virtual communication network, which allows a state graph's arc traversed by one instance to be known to all other instances (thus, they will not traverse it by themselves wasting no time to duplicate work that has been already done).

Parallelization has been used for simulation-based verification of various hardware designs. Depending on the design complexity and verification purposes, model graphs included from thousands to millions of nodes and up to several millions of arcs. Testbench execution has been performed on 1-150 computers. We have conducted a number of experiments and have measured the parallelization efficiency **T(1)/(n·T(n))**, where **T(n)** is time of testbench execution on n computers. The experiments show that if the communication topology is chosen correctly, the parallelization efficiency always exceeds 0.8 (we used "ring" for 8 or less computers and "two-dimensional torus" for 9 or more computers).