Simplified stimuli generation for scenario and assertion based verification

Luca Piccolboni
Department of Computer Science
University of Verona, Italy
Email: luca.piccolboni@studenti.univr.it

Graziano Pravadelli
Department of Computer Science
University of Verona, Italy
Email: graziano.pravadelli@univr.it

***Abstract*—Simulation-based approaches that require to drive the design under verification (DUV) to specific conditions, like for example, scenario-based testing and dynamic assertion-based verification (ABV), cannot rely on generic coverage-driven stimuli generators. On the contrary, constraint-based generation must be adopted. In this context, among several solutions, the Universal Verification Methodology (UVM) and the SystemC Verification Library (SCV) represent the main alternatives. However, their powerfulness is paid in term of easiness of use. In fact, their application generally requires to write complex pieces of code to specify the constraints that must be satisfied by the stimuli generator to produce the desired sequences of values. More is the complexity of setting up an effective stimuli generator, more is the risk of failing to capture the right behaviour and/or having a longer verification time. To overcome these problems, the paper presents a framework and a corresponding language for the automatic generation of stimuli that requires to write intuitive and compact directives representing the desired constraints. The approach is independent from the language adopted for the DUV implementation and it works for both embedded hardware as well as embedded software.**

I. Introduction

The generation of high-quality stimuli represents the basic step for all simulation-based techniques, which are applied throughout the abstraction levels of the embedded system design flow to verify both functional and non-functional (i.e., aging, timing, power, thermal) properties. A low-quality set of stimuli causes a false sense of safety because it is able to exercise only a small part of the DUV behaviours. In this case, testing techniques fail to discover bugs, power estimation approaches generate incomplete and incorrect results, design exploration methodologies provide non-optimal solutions, etc. Thus, several *static* and *dynamic* approaches have been defined, at different abstraction levels, to guarantee the efficient generation of effective and as much exhaustive as possible sequences of stimuli [1].

Static approaches aim at exploring the whole DUV state space in a rigorous way [2]. They are exhaustive, but they require to encode and traverse the DUV by exploiting mathematical formalisms (e.g., transition systems, symbolic encoding, etc.) that generally lead to the state explosion problem for large designs. A faster and less risky alternative for state explosion, is represented by dynamic approaches [3]. In this case, random or probabilistic techniques are adopted to generate stimuli, whose quality is measured through coverage metrics, like for example, code coverage or fault coverage. The main drawback of these techniques is represented by the lack of exhaustiveness, which is only partially compensated by the possibility of quickly generating a huge amount of stimuli, hoping that they cover the most of DUV behaviours. Semi-formal techniques have been proposed too, that try to mix the use of both dynamic and static algorithms [4]. The first to rapidly stimulate easy-to-reach states, the second to cover corner cases.

Independently from the static or dynamic approach, stimuli generation is mainly performed independently from a specific objective, and it is generally guided by some structure-oriented coverage-driven process whose goal is to create a set of stimuli that achieves an high coverage of the DUV behaviours [5]. In fact, coverage metrics evaluate how well the code is exercised, rather than how well design functionalities are stressed.

However, in several cases, after an initial execution of a coverage-driven stimuli generator, designers, verification engineers and testers need to create few initialization sequences to bring the DUV into particular states where it is possible to check specific conditions. This is particularly true, when verification and/or exploration are performed on the basis of scenarios [6], where test conditions, data to be used for testing, and the expected results are provided. The same happens when dynamic ABV is addressed [7], where assertion checkers must be stimulated in a non vacuous way. For example, people that perform verification of embedded system applications are generally required to follow a test plan. This contains a list of checks which are formalized into (temporal) assertions to be verified on specific scenarios. As a practical example, let us consider the following directive extracted from the test plan of an industrial mixing machine:

*"Bring the machine in a situation where T_cold=*5 °C*, T_pipe=*20 °C*, T_hot=*40 °C*, and T_set=*25 °C*, then linearly increase T_pipe= till* 30 °C*, and then activate the water discharge and check if the final water flow is composed only by water arriving from the water supply network and from the hot water container."*

To test the previous directive, verification engineers must put the mixing machine in the correct state (where the temperatures are as required), and then they must drive the simulation to respect that *T_pipe* increases its value linearly till 30 °C. It is almost impossible to reproduce such a situation by means of a traditional structure-oriented stimuli generator, whose aim is just to maximize the targeted coverage. On the contrary, constraint-based approaches, which explicitly generate stimuli that respect specific constraints, should be adopted [8].

In this context, the UVM represents one of the most popular framework for stimuli generation [9]. An UVM testbench generates stimuli corresponding to a specific verification goal and it sends them to the DUV. Coverage monitors are added to measure progress and identify non-exercised functionalities. Checkers can also be included to identify undesired DUV behaviours. However, the use of UVM requires to write complex pieces of code. Such a complexity to set up an effective stimuli generator reflects in the risk of failing to capture the right behaviour and/or in a longer verification time. Moreover, UVM works only for SystemVerilog designs. This is partially compensated by the definition of different UVM dialects that cover also other hardware description languages (HDLs) like, for example, the System Verification Methodology (SVM)

978-1-4799-4711-9/14/$31.00 ©2014 IEEE

for SystemC [10]. Another possibility is represented by SCV which provides pseudo-random generators and an integrated constraint solver based on BDDs for SystemC verification [11]. However, similar considerations to the UVM case apply also for SCV, concerning the complexity of implementing advanced generators. Moreover, several limitations affect SCV [12].

To overcome UVM and SCV complexity and being HDL-free, this paper is intended to present a framework for the constraint-based automatic generation of stimuli that:

- requires to write very few lines of directives representing the desired constraints by using a simple specification language;
- works independently from the language adopted for the DUV implementation;
- can be applied for both embedded hardware as well as embedded software;
- and creates sequences of stimuli suited for supporting verification and exploration methodologies that require to mimic specific situations, which are necessary, for example, in scenario-based testing or dynamic ABV.

A preliminary work in this direction, based on SCV, has been presented in [13], but with respect to [13] the current paper refines and extends the specification language, replaces the SCV generation engine with a more flexible C++ SMT-based approach to remove the SystemC dependency, and provides a larger set of experimental results.

The rest of the paper is organized as follows. Section II proposes a specification language to define the behaviour of the desired sequence of stimuli. Section III describes how stimuli generators are implemented on the basis of the specification language defined in Section II. Section IV compares the proposed approach with respect to the UVM facilities related to stimuli generation. Section V is devoted to experimental results. Finally, Section VI draws some concluding remarks.

II. Stimuli specification

The stimuli generation engine relies on a specification language that allows the user to easily define the sequence of values that must be generated for each input line of the DUV. The specification language is defined by means of a context-free grammar whose syntax is defined in Fig. 1. Non-terminal symbols are written between chevrons in lower-case letters. Terminal strings are enclosed in single quotes. Finally, strings starting with a capital letter and enclosed in chevrons are tokens that can be considered terminal symbols.

Given an input line, the specification of the sequence of values that must be generated is defined by a *named_generator* with *kind* OUTPUT. The *kind* TEMP is used to define stimuli generators that create sequences of values representing the basis for more complex OUTPUT generators. The *type* is used to define the size of the input line associated to the generator. The *name* is the identifier of the generator. The behaviour of the generator is specified by a *generator* instance among the following:

- *constant*: it is used to generate a sequence whose elements correspond to the numeric constant *Value*;
- *uniform:* it creates a sequence of values uniformly distributed between *Range_min* and *Range_max*;
- *reference:* it generates the same values of the *named_generator* identified by the argument *Name* and it is used to create dependencies among different generators;

```
⟨named_generator⟩ ::= ⟨kind⟩ ⟨Name⟩ ⟨generator⟩
⟨kind⟩ ::= ‘OUTPUT’ | ‘TEMP’ ⟨type⟩ ⟨Precision⟩
⟨type⟩ ::= ‘FLOAT’ | ’ | ‘INT8’ | ‘UINT16’ | ‘INT16’ | ‘UINT32’ | ‘INT32’ | ‘UINT64’ | ‘INT64’
⟨generator⟩ ::=⟨constant⟩ | ⟨uniform⟩ | ⟨reference⟩ | ⟨sequence⟩ | ⟨sum⟩ | ⟨product⟩ | ⟨uminus⟩ | ⟨delay⟩ |
               ⟨range_restrict⟩ | ⟨time_expand⟩ | ⟨function⟩ | ⟨input⟩ | ⟨constraint⟩
⟨constant⟩ ::= ‘Constant’ ⟨Value⟩
⟨uniform⟩ ::= ‘Uniform’ ⟨Range_Min⟩ ⟨Range_Max⟩
⟨reference⟩ ::= ‘Reference’ ⟨Name⟩
⟨sequence⟩ ::= ‘Sequence’ ⟨Do_Loop⟩ ⟨sequence_list⟩ ‘end’
⟨sequence_list⟩ ::= ⟨sequence_pair⟩ | ⟨sequence_list⟩ ⟨sequence_pair⟩
⟨sequence_pair⟩ ::= ⟨generator⟩ ⟨Duration⟩
⟨sum⟩ ::= ‘Sum’ ⟨generator⟩ ⟨generator⟩
⟨product⟩ ::= ‘Product’ ⟨generator⟩ ⟨generator⟩
⟨uminus⟩ ::= ‘UMinus’ ⟨generator⟩
⟨delay⟩ ::= ‘Delay’ ⟨Delay_Amount⟩ ⟨Initial_Value⟩ ‘end’ ⟨generator⟩
⟨range_restrict⟩ ::= ‘Range_Restrict’ ⟨Range_Min⟩ ⟨Range_Max⟩ ⟨generator⟩
⟨time_expand⟩ ::= ‘Time_Expand’ ⟨Factor⟩ ⟨generator⟩
⟨function⟩ ::= ‘Function’ ⟨Ftype⟩ ⟨Initial_Value⟩ ⟨Offset⟩
⟨input⟩ ::= ‘Input’ ⟨Name⟩ ⟨Do_Loop⟩
⟨constraint⟩ ::= ‘Constraint’ ⟨constraint_list⟩ ‘end’ ⟨Expression⟩ ‘;’
⟨constraint_list⟩ ::= ⟨constraint_pair⟩ | ⟨constraint_list⟩ ⟨constraint_pair⟩
⟨constraint_pair⟩ ::= ⟨generator⟩ ⟨Name⟩
```

Fig. 1. Grammar of the stimuli specification language.

- *sequence*: it allows to create a generator composed of a sequence of different behaviours specified by different *generators*. Each generator in the sequence is active for a specific *Duration*. The *sequence* can repeat cyclically or the last *generator* can be used till the end of the generation, respectively when *Do_Loop* equals 1 or 0;
- *sum:* it sums the values returned by two *generators*;
- *products*: it multiplies the values returned by two *generators*;
- *uminus*: it returns the negative of the value returned by a *generator*;
- *delay*: it delays the generation of values according to the specified *generator* for *Delay_Amount* simulation instants. Before the *Delay_Amount* is elapsed the generator returns the constant *Initial_Value*;
- *range_restrict*: it restricts the range of values returned by the associated *generator* within the interval [*Range_min*, *Range_max*]. If the associated *generator* returns a value v outside such an interval, *range_restrict* returns *Range_Min* when $v < Range\_Min$ and *Range_Max* when $v > Range\_Max$. Otherwise it returns v;
- *time_expand*: it stretches horizontally the values returned by the associated *generator* by a given factor k. If the the associated *generator* behaves as a function of time $f(t)$, a time expansion of a factor k produces the behavior $g(t) = f(t/k)$;
- *function*: it allows to reproduce the behavior of a given function *Ftype* (e.g., sine, cosine, logarithm, etc.). In particular, the sequence of generated values is computed by $Ftype(Initial\_Value + t * Offset)$ where t represents the simulation time;
- *input*: it allows to recall the associated *generator* into a new generation. This is particularly useful when a sequence of stimuli previously generated is used as prefix (i.e., it acts as initialization for the DUV) of a new sequence;
- *constraint*: it creates values that satisfy complex constraints described by an *expression* representing a first-order formula, possibly, involving other *generators*.

```
OUTPUT gen_1
  Sequence 0
    Uniform −1 1 20
    Function sin 0 0.1 30
    Constant 0.24 1
  end
OUTPUT gen_2
  Range_Restrict 0 2
    Function log 1 0.1
OUTPUT gen_3
  Sum
    Constant −0.01
    Delay 1 0.01 end Reference gen_3
```

Fig. 2. Example of a stimuli specification using several *generators*.

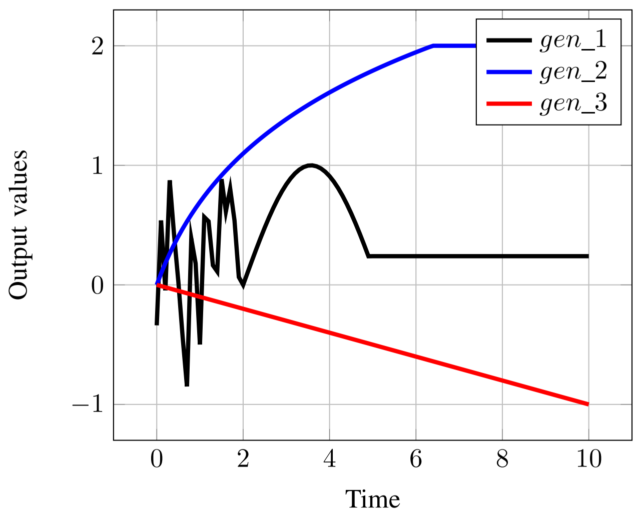

Fig. 3. Plot of stimuli created by the *generators* of Fig. 2.

Previous generators create a sequence of values independently from the simulation time and from the mechanism adopted to notify the passing of time. They assume a discrete-time model where new values are provided to the input lines at each timestamp. The definition of what a timestamp is depends on the abstraction level of the DUV and on the simulation engine. For example, in case of an HW register transfer level (RTL) model, timestamps are represented by clock cycles, at transaction level model (TLM), they correspond to the starting/ending of transactions, in the context of embedded SW, they are related to the iteration loop of the control application, etc.. Our framework just needs to know a couple of parameters: the length of the simulation and the duration of the timestamp. Thus, for example, to generate a sequence of stimuli for an RTL simulation of 10 seconds with a clock cycle of 100 ms, the user can specify the couple (10000, 100) and in this way a sequence composed of 100 values is generated.

A. Examples of generators

The primary goal of the previous generators is to describe the behaviour of digital/analog inputs from environmental probes which typically equip embedded systems. In particular, the proposed specification language allows the definition of a library of generators to solve recurring patterns of use whose definition is quite difficult, like, for example, linear filters, tracking trends for physical parameters, etc..

Some examples to show the effectiveness and the easiness of use of the proposed language in specifying the behaviour of different stimuli generators are described hereafter.

Fig. 2 shows the use of several *generators* that create the three sequences of stimuli depicted in Fig. 3. The sequence corresponding to *gen_1* is obtained by sequencing tree behaviors through the *sequence* generator such that the generated values respect a uniform distribution between -1 and 1 till simulation instant 20, then a sine function till simulation instant 30, and finally a constant. Then, *gen_2* shows the use of *range_restrict* to create a logarithmic waveform with an upper bound represented by the value 2. Finally, *gen_3* generates a decreasing sequence of values by using *delay* and *reference* to reproduce the effect of the function $f(t) = f(t-1) - 0.01$, where $f(0) = 0.01$. The corresponding plot is shown in Fig. 3 where the couple (10000, 100) has been used to specify the simulation time and the timestamp length.

```
TEMP FLOAT 0 filter_parameter Constant 0.75
TEMP FLOAT 0 temp1 Uniform −1 1
OUTPUT filter
  Sum
    Product
      Sum
        Constant 1
        UMinus Reference filter_parameter
      Reference temp1
    Product
      Reference filter_parameter
      Delay 1 0 end Reference filter
```

Fig. 4. Specification of a linear filter.

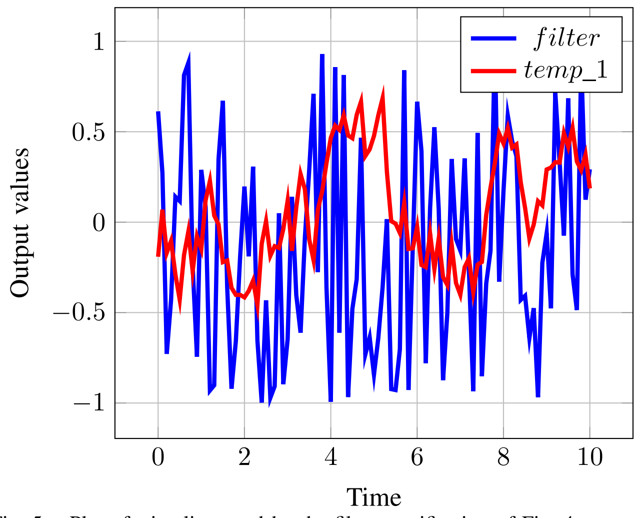

Fig. 5. Plot of stimuli created by the filter specification of Fig. 4.

An interesting example is shown in Fig. 4, where a generator to implement the behavior of a linear filter is defined. *temp_1* is a uniform generator returning value in [-1,1]. *filter* is the linear filter that uses *temp_1* according to the following expressions:

$$filter(t) = 0;$$
$$filter(t+1) = 0.25 * temp\_1(t+1) + 0.75 * filter(t).$$

The corresponding plot is shown in Fig. 5.

The use of the *input* generator is shown in Fig. 6. The sequence created by the *gen_4* generator is composed of a prefix including the first 40 elements produced by the *prev_sequence* generator, which is supposed to have been previously defined and possibly used for a different verification run. The *input* generator is used to import the values provided by *prev_sequence* into the *prev* TEMP generator. Then, the sequence is completed according to a sine function specified by the *post* TEMP generator. Finally, *prev* and *post* are referenced by *gen_4* to create the final sequence. The corresponding plot is shown in Fig. 7.

A final example, in Fig. 8, is related to the use of the *constraint* generator. The generator *gen_5* imposes that each

```
 OUTPUT prev_sequence
    Sequence 1
       Constant −1 5
       Constant 1 5
   end

TEMP FLOAT 0 prev Input prev_sequence 0
TEMP FLOAT 0 post Function sin 1 0.1
OUTPUT gen_4
   Sequence 0
      Reference prev 40
      Reference post 1
   end
```

Fig. 6. Specification of a stimuli sequence with an initialization prefix imported from a previous generation.

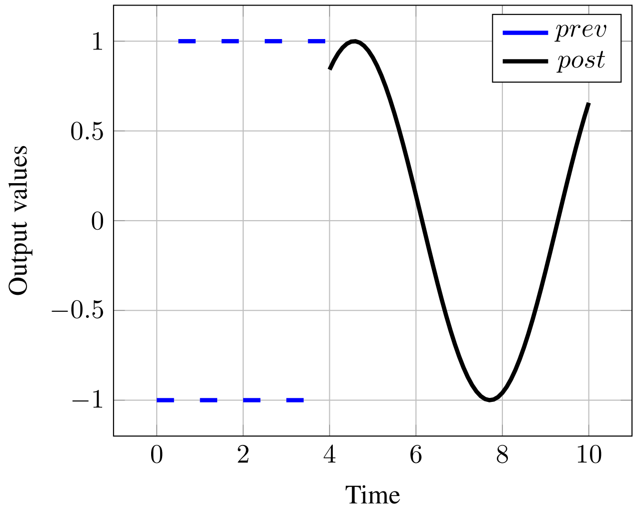

Fig. 7. Plot of the generator gen_4 defined in Fig. 6.

returned value v satisfies the constraint $v > x$ and $v < y$, where x and y are specified by a couple of *function* generators which represent the allowed upper and lower bounds for v. The corresponding plot is shown in Fig. 9.

III. Stimuli generation

Given a file containing the specification of a set of generators according to the syntax described in Section II, the corresponding sequence of stimuli is generated by a C++ engine. In particular, only *constant*, *uniform*, *function* and *constraint* generate values, while *reference*, *sequence*, *sum*, *product*, *uminus*, *delay*, *range_restrict*, *time_expand* and *input* are used to reuse, combine or modify the values returned by other generators.

The C++ engine straightforward implements the *constant* and *function* generators, by returning, respectively, the required constant value or a sequence of values respecting the specified function at varying of time. Several standard C functions are already built-in, while the engine can be extended by adding further used-defined functions.

On the contrary, the implementation of the *uniform* and *constraint* engines relies, respectively, on the Random-Boost library [14], and the MathSAT5 constraint solver [15].

The Random-Boost library provides a set of powerful engines that generate random (integer and real) numbers according to the most popular probabilistic distributions, like, for example, the uniform distribution. Custom distributions can be defined as well. Moreover, by using the seed provided by the *random_device* class, we are guaranteed that the generated numbers are actually random, even in the case they are generated at a very high frequency. The same assurance cannot

```
OUTPUT LowerBound Function log 1 0.1
OUTPUT UpperBound
   Product
      Function log 1.1 0.1
      Constant 2
OUTPUT gen_5
   Constraint
      Reference LowerBound x
      Reference UpperBound y
   end (assert (and (> this x) (< this y)));
```

Fig. 8. Specification of a stimuli sequence by means of the *constraint* generator.

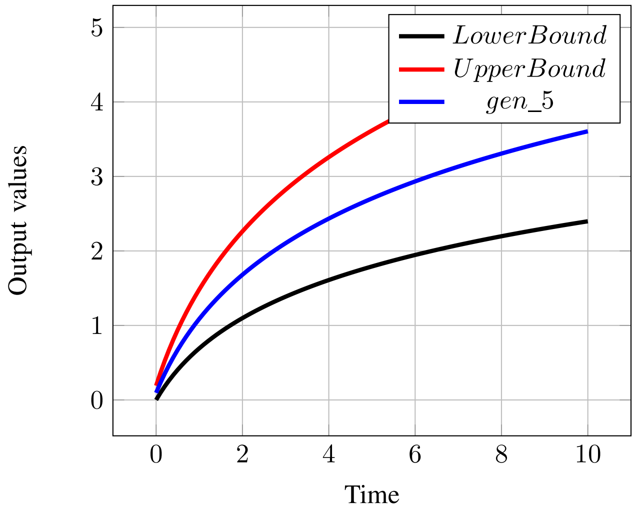

Fig. 9. Plot of the generator defined in Fig. 8.

be obtained by using the classical *rand* function of the standard C library that in case of two very close consecutive calls can provide the same value, even when the *srand* function is called to initialize the seed.

On the other hand, the generation of random stimuli that respect a constraint expressed by a first-order formula requires the adoption of a constraint solver. In our case, the MathSAT5 Satisfiability Modulo Theories (SMT) solver has been exploited. Differently from a SAT solver, which works only for the theory of Boolean values, an SMT solver can solve constraints according to several theories like integer, real, array, etc.. MathSAT5 has been integrated in the proposed environment by means of the SMT-Lib 2.0 library [16]. Such a library represents a common interface for the most popular SMT solvers. In this way, MathSAT5 could be substituted, in the future, with more efficient constraint solvers without changing the grammar of the *constraint* generator.

IV. Comparisons with UVM

UVM represents the most popular framework for testbench generation. It is based on the SystemVerilog language, but UVM Multi-Language (UVM-ML), a modular solution for integrating verification components written in different languages, is also available with some restrictions [17].

An UVM testbench is composed of reusable verification environments called verification components. A verification component is an encapsulated, ready-to-use, configurable verification environment for an interface protocol, a design submodule, or a full system. It allows to stimulate the DUV by generating constrained-random stimuli, measure the DUV coverage and monitor assertion checkers.

In the following, the UVM facilities concerning constrained-random stimuli are compared, by means of an example, against the stimuli generation framework

```
class tridomix_trans extends uvm_sequence_item;
  rand int tset;
  rand int tcold;
  rand int tcold_der;
  int unsigned delta_A = 2;

  function new();
    super.new();
  endfunction

  // constraints
  constraint lower_bound {
    tset > (tcold - delta_A); }
  constraint upper_bound {
    tset < (tcold + delta_A); }
  constraint derivate {
    abs(tcold_der) <=1 ; }
endclass

class my_driver #(type REQ = uvm_sequence_item)
         extends uvm_driver #(REQ);
  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction

  task run_phase(uvm_phase phase);
   REQ req;
   forever begin
     // to get a transaction from the sequencer
     seq_item_port.get(req);
     // transaction fields are used on the DUV
     drive_item(req);
     // request completed
     seq_item_port.item_done();
   end
  endtask

  function drive_item(REQ req)
    // to be implemented
  end function
endclass

`define NUM_SEQS 1000
class env extends uvm_env;
  uvm_sequencer #(tridomix_trans) sqr;
  my_driver #(tridomix_trans) drv;
  tridomix_trans sequence[`NUM_SEQS];

  function new(string name, uvm_component parent);
    super.new(name, parent);
    // Creation of the sequencer
    sqr = new("tridomix_sequencer", this);
    // Creation of the driver
    drv = new("tridomix_driver", this);
    // Creation of the transaction sequence
    for (int i = 0; i < `NUM_SEQS; i++) begin
      sequence[i] = new("sequence");
    end
    // Connection between the driver and the sequencer
    drv.seq_item_port.connect(sqr.seq_item_export);
  endfunction

  task run_phase(uvm_phase phase);
    phase.raise_objection(this);
    for (int i = 0; i < `NUM_SEQS; i++) begin
      fork
        sequence[i].start(sqr, null);
        join_none #0;
      end
      wait fork;
      phase.drop_objection(this);
  endtask
endclass

module top;
  import user_pkg::*;
  import uvm_pkg::*;
  env e;

  initial begin
    e = new("env", null);
    run_test();
  end
endmodule
```

Fig. 10. UVM code to generate a test sequence for the mixing machine.

proposed in this paper. Let us consider, the following directive from the industrial test plan of a mixing machine.

It is necessary to verify that the machine moves from the state IDLE to the state SECOND_TEMP when all the following conditions are true:

```
OUTPUT tcold Uniform −20 5
OUTPUT tcold_der
    Constraint
    Constant 0.3 x
  end (assert (< this 0.3));
OUTPUT delta_A Constant 2
TEMP FLOAT 0 lower_bound
  Sum
    Reference tcold
    UMinus Reference delta_A
TEMP FLOAT 0 upper_bound
  Sum
    Reference delta_A
    Reference tcold
OUTPUT tset
  Constraint
    Reference lower_bound low
    Reference upper_bound up
  end (assert (and (< this up) (> this low)));
```

Fig. 11. Proposed specification directives to generate a test sequence for the mixing machine.

- $T_{cold} - Delta_A < T_{set}$;
- $T_{set} < T_{cold} + Delta_A$;
- $\mid derivative(T\_cold) \mid < 0.3$;

where T_{cold} is the temperature of the cold water measured by a probe, $Delta_A$ is a constant defined by the installer, and T_{set} is the required temperature set by the user.

In the context of ABV, the verification of the previous directive requires: (i) to define an assertion that captures the desired intent, and (ii) to generate a set of stimuli sequences that fire the assertion to avoid its vacuous passing. Clearly, the assertion is fired only when the three conditions described in the test plan are satisfied.

The UVM code to create a stimuli sequence that satisfies the conditions reported in the test directive of the mixing machine is reported in Fig. 10. Class *tridomix_trans* is the basic transaction, which represents the data items to create the input values for the DUV according to the desired constraints. Class *my_driver* is the driver for the DUV. In particular, the function *drive_item*, whose implementation is not reported for saving space, uses values generated by the transaction to drive the DUV simulation. Class *env* is the verification environment that instantiates a sequencer, a driver and a sequence of transactions. The sequencer is an advanced stimulus generator that returns a random data item upon request from the driver according to constraints specified in the transaction. Finally, the module *top* instantiates the verification environment and runs the stimuli generation. On the contrary, Fig. 11 shows what a user should write to create a corresponding sequence of stimuli by means of the framework proposed in the previous sections.

By comparing Fig. 10 and Fig. 11 it appears that creating a sequence by means of the proposed specification framework is much more easier than implementing a corresponding UVM verification environment. Moreover, it is worth remembering that UVM works only for SystemVerilog hardware descriptions, while the proposed framework can be adopted for both HW and SW designs. On the other hand, UVM provides further features, like coverage measure and assertion monitors, which make it a comprehensive verification environment, while the proposed framework is only intended to provide an easy-to-use and effective stimuli generator to drive the DUV during specific simulation approaches like, for example, scenario-based testing and assertion-based verification.

DUV			St.-oriented		Our approach		
Name	Lines	Ass.	Act.	Time	Act.	Dir.	Time
Breadmaker	2006	20	30%	0.008s	100%	7	0.064s
RPC-3D-door	22043	63	98%	0.012s	100%	168	0.132s
DSC-2L	37545	14	35%	0.004s	100%	114	1.676s

TABLE I. Experimental results (Embedded SW applications).

DUV			St.-oriented		Our approach		
Name	Lines	Faults	Cov.	Time	Cov.	Dir.	Time
ADPCM	341	143	75.5%	0.008s	96.5%	102	0.448s
AM2910	622	156	48.7%	0.004s	85.3%	506	0.136s

TABLE II. Experimental results (Hardware components).

V. Experimental results

The effectiveness of the proposed framework has been evaluated in two ways. First, we evaluated its capability in the context of dynamic ABV of C++ embedded software applications. As briefly summarized in the introduction, dynamic ABV requires to stimulate the DUV with high-quality sequences such that assertion checkers can be actually activated to avoid vacuous passing of the corresponding assertions. The second experiments has been conducted to measure the quality of the proposed framework in the context of traditional fault simulation of SystemC RTL hardware components. In both cases, the framework has been compared, from the effectiveness point of view, with respect to a structure-oriented (coverage-driven) generator included into a commercial verification suite [18]. We do not report such a comparison with respect to UVM since our approach and UVM have the same capability from the point of view of (SAT/SMT) constraint-based generation. Thus, the most relevant comparison between UVM and our approach concerns the difference in the easiness of use and the application domain, as described in the previous section.

Table I presents the results in the context of dynamic ABV. Columns report benchmarks characteristics, i.e., the DUV name (Name), the number of code lines (Lines) and the number of defined assertions (Ass.), and results, in terms of percentage of fired assertions (Act.) and time required for stimuli generation (Time) by using both the structure-oriented engine and the proposed framework. The total number of directive lines written by means of the specification language defined in Section II is also reported (Dir.). To be fair, both the generators have been required to create sequences of the same length (thousands of test vectors). Our approach allows to activate all the assertions for all the considered benchmarks, while the structure-oriented generator achieves a good activation percentage only in one case. This is motivated by the fact that the activation of assertions for *Breadmaker* and *DSC-2L* depends on conditions which cannot be simply guaranteed by a structure-oriented generator. Indeed, for the most of assertions, an initialization sequence is required to bring the application in a particular state from which their activation is guaranteed. Such initialization sequences, reproducing the real environment where the application operates, can be easily obtained by reproducing the required behaviours with the specification language described in Section II. Execution times show that our approach is more expensive, in particular for *DSC-2L* where an extensive use of the SMT-based *Constraint* generator has been required. Indeed, generation time is almost negligible for all benchmarks.

Table II reports a similar analysis for the fault simulation context. In this case, the number of assertions and the percentage of their activations are replaced by the number of faults (Faults) and the achieved fault coverage (Cov.). The fault model defined in [19] has been adopted. The proposed approach provides an higher fault coverage than the structure-oriented engine showing its effectiveness also for a "coverage-driven" goal. However, the number of directives that we needed to write for the *AM2910* micro controller is quite high compared to the complexity of the DUV. This is due to the necessity of mimic the behaviour of an *AM2910* program by means of the specification language of Section II, to accurately stimulate its instruction set architecture, which is composed of 16 instructions. On the other hand, for the same reason, the coverage achieved by the structure-oriented approach is very low due to its inability of simulating a real operating environment for the micro controller.

VI. Conclusions

The paper presented a stimuli specification language and a corresponding stimuli generation engine targeting the reproduction of specific conditions, as required, for example, by scenario-based testing and dynamic ABV. The language allows to intuitively write directives for the engine to generate constraint-based stimuli sequences that respect the desired conditions. In comparison to the popular UVM environment, our approach is intended to provide a simpler way of specifying sequence behaviours, it works independently from the abstraction level and from the DUV implementation language, and it can be used for the hardware as well as the software domain. In comparison to a structure-oriented engine we showed that our framework generates stimuli which more effectively activate assertions and achieves higher fault coverage.

References

[1] D. Pradhan and I. Harris, *Practical Design Verification*. Cambridge Univ Press, 2009.

[2] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. of OSDI*, 2008, pp. 209–224.

[3] S. M. Plaza, I. L. Markov, and V. Bertacco, "Random stimulus generation using entropy and XOR constraints," in *Proc. of ACM/IEEE DATE*, 2008, pp. 664–669.

[4] G. Di Guglielmo, M. Fujita, F. Fummi, G. Pravadelli, and S. Soffia, "EFSM-based model-driven approach to concolic testing of system-level design," in *Proc. of ACM/IEEE MEMOCODE*, 2011, pp. 201–209.

[5] S. Yang, R. Wille, D. Grobe, and R. Drechler, "Coverage-driven stimuli generation," in *Proc. of IEEE DSD*, 2012, pp. 525–528.

[6] C. Kaner, "An introduction to scenario testing," *Lecture Notes, Center for Software Testing Education and Research, Florida Institute of Technology*, 2003.

[7] D. Tabakov, "Dynamic assertion-based verification for SystemC," Ph.D. dissertation, 2010, Rice University.

[8] R. A. DeMillo and A. J. Offutt, "Constraint-based automatic test data generation," *IEEE Trans. Softw. Eng.*, vol. 17, no. 9, pp. 900–910, 1991.

[9] Accellera Organization, "Universal Verification Methodology," 2012. [Online]. Available: http://www.accellera.org/downloads/standards/uvm/uvm_users_guide_1.1.pdf

[10] M. F. Oliveira, C. Kuznik, H. M. Le, D. Grosse, F. Haedicke, W. Mueller, R. Drechsler, W. Ecker, and V. Esen, "The system verification methodology for advanced TLM verification," in *Proc. of IEEE/ACM/IFIP CODES+ISSS*, 2012, pp. 313–322.

[11] Accellera Organization, "SystemC Verification Library," 2012. [Online]. Available: http://www.accellera.org/activities/committees/ systemc-verification/

[12] D. Grosse, R. Ebendt, and R. Drechsler, "Improvements for constraint solving in the Systemc Verification Library," in *Proc. of ACM GLSVLSI*, 2007, pp. 493–496.

[13] G. Di Guglielmo and G. Pravadelli, "A testbench specification language for SystemC verification," in *Proc. of IEEE/ACM/IFIP CODES+ISSS*, 2012, pp. 333–342.

[14] "Random boost library." [Online]. Available: http://www.boost.org/

[15] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani, "The MathSAT5 SMT solver," in *TACAS*, ser. Lecture Notes in Computer Science, vol. 7795. Springer, 2013, pp. 93–107.

[16] C. Barrett, A. Stump, and C. Tinelli, "The SMT-LIB Standard: Version 2.0," in *Proc. of International Workshop on Satisfiability Modulo Theories*, 2010.

[17] "UVM -ML Open Architecture," 2013. [Online]. Available: http://forums.accellera.org/files/file/ 65-uvm-ml-open-architecture/

[18] "radCHECK." [Online]. Available: http://www.verificationsuite.com

[19] V. Guarnieri, G. Di Guglielmo, N. Bombieri, G. Pravadelli, F. Fummi, H. Hantson, J. Raik, M. Jenihhin, and R. Ubar, "On the reuse of TLM mutation analysis at RTL," *J. Electron. Test.*, vol. 28, no. 4, pp. 435–448, 2012.