2018 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

Using Constraints for SystemC AMS Design and Verification

Thilo Vörtler, Karsten Einwich, COSEDA Technologies GmbH, Dresden, Germany
(thilo.voertler|karsten.einwich@coseda-tech.com)

Muhammad Hassan, Daniel Große, DFKI GmbH, Bremen, Germany
(muhammad.hassan|daniel.grosse@dfki.de)

***Abstract*— In this paper we discuss how constraints can be applied for the design and verification of mixed-signal virtual prototypes based on SystemC AMS [1][2]. In particular, we propose two kinds of constraints w.r.t. mixed-signal designs: parameter constraints, and runtime constraints. We show that the CRAVE constraint solver has been extended to handle *real value* constraints for analog systems. Furthermore, we explain how constraints can be easily used to modify the DUT for modeling parameters and other kinds of variations. Using a DC-DC converter circuit example we show how our flow can be used to modify the DUT based on constraints and to generate random stimuli for verification in a UVM-SystemC environment.**

Keywords—Constraints, parameter variations, real value constraints,SystemC AMS, UVM;

I. INTRODUCTION

The complexity of modern electronic systems is still growing due to new application areas like autonomous vehicles and the growing interconnection between devices in the *Internet of Things* (IoT). SystemC AMS as design language allows to model such complex systems on different levels of abstractions. Use cases range from fast virtual prototypes for early software development and algorithm design to Register-Transfer-Level implementations comparable with hardware description languages such as VHDL and SystemVerilog [4]. The AMS part of SystemC allows to model analog behavior through different models of computation like *Timed Data Flow* (TDF) and *Electrical Linear Networks* (ELN).

Constraint-based randomization has long been applied for the functional verification of digital circuits and systems in verification languages like ***e*** or SystemVerilog. By describing a range of possible input stimuli, a constraint solver can randomly generate valid scenarios. Constraint randomization is thereby often used together with verification methodologies like the *Universal Verification Methodology* (UVM) [3]. UVM has also been made available for SystemC [6], which requires the use of external constraint solving libraries. The widely used libraries are SCV [5] and CRAVE [7] . In practice the use of constrained randomization techniques is currently limited to the verification of digital circuits. This has several reasons:

- Accurate behavior models of analog system parts are required for functional verification, as transistor level netlists are too slow to simulate a complete system.
- Automatically verifying the correctness of analog behavior is harder than in the digital domain due to the continuous time behavior of signals, especially a simple comparison for equality is not possible.
- Constraints in SystemVerilog cannot be solved during elaboration of the DUT. Therefore, it is not possible to modify DUT parameters like gain, cut off-frequencies or impedances by constraints. Analog DUT parameters are due to production tolerances blurred and these tolerances – in opposite to digital circuits – cannot be abstracted and usually directly influence the functionality. Thus, DUT parameter tolerances must be considered during the functional system verification.

As a consequence, to bring constrained-random for AMS design and verification to life, we propose the following new additions:

This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project CONVERS under contract no. 16ES0656.

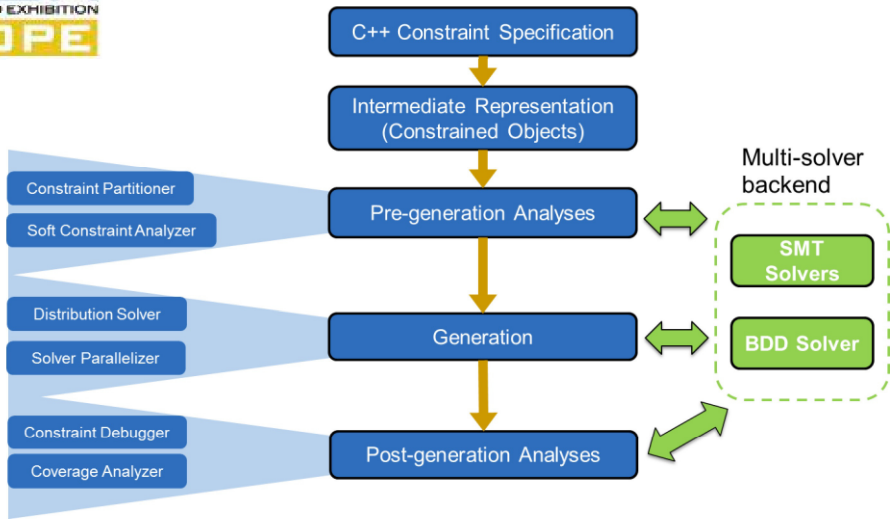

Figure 1: CRAVE Architecture

- *Parameter constraints* change the structure of the DUT and can be used to model complex parameter variations. For example, it can be expressed that the temperature shall be in a range from 0 to 80°C, however, all elements shall have only a difference of 10°C during a simulation run.
- *Runtime constraints* are used to control the input to the DUT during simulation. For example, these constraints drive random stimulus to the DUT as done in UVM. These kinds of constraints are similar to the constraints known in the digital world like used in SystemVerilog based testbenches with the exception, that real values must be supported.

The paper is structured as follows. In Section II we introduce the CRAVE library and discuss real value extensions. Section III demonstrates how constraints can be used for modeling parameter variations and design space exploration using SystemC. This includes both, parameter and runtime constraints. A parallel simulation framework is introduced to speed up simulations. In Section IV a case study modeling a DC-DC converter is presented. It is demonstrated how the DUT can be modified using constraint randomization. Furthermore, constraints are used to generate input stimuli, and checkers will be used to automatically verify simulation results for regression runs. Finally, the paper is concluded in Section V.

II. The CRAVE Library and Real Value Extensions

CRAVE is an open-source constrained random verification environment developed primarily for SystemC[1,2]. CRAVE provides many improvements over SCV such as a better API for constraint specification and management, automatic constraint debugging, etc. (see [8] for more details). Most importantly, the constraint solver of CRAVE is based on modern *Satisfiability Modulo Theories* (SMT) solvers [9], which are well-known for their much better scalability in comparison to *Binary Decision Diagrams* (BDDs). CRAVE provides support for soft constraints, distribution constraints, and constraint partitioning among many other features.

The architecture of CRAVE is shown in Figure 1. The first step is constraint specification where different constraints (hard/soft constraints) can be specified on random variables. Afterwards, CRAVE transforms the given constraints into an intermediate representation. Then the constraints are checked for dependency on each other by a constraints partitioner module. If there does not exist any dependency between constraints, CRAVE automatically separates constraints into independent constraints sets. This solves the performance bottleneck as there could be thousands of constraints in the verification environment. Secondly, CRAVE analyzes the given constraints if they are *hard constraints*; must always be satisfied, or *soft constraints*; can be ignored when conflict arises with hard constraints. The essential use of *soft constraints* is to define the default behavior of the DUT. This default behavior can be modified later using class specializations. The soft constraints are assigned a unique priority based on the order of runtime constraint creation. The high level idea is that when there is a contradiction between constraints, the lower priority constraint is dropped. Further details on soft constraints and their uses can be found in [15]. During the *Generation* phase, CRAVE checks for defined distribution (user defined biases) on the constraints. The distributions reflect how comprehensively the solution space will be covered. Afterwards, CRAVE can use a multi-

1. http://www.systemc-verification.org/crave
2. https://github.com/agra-uni-bremen/crave

threaded environment to execute a BDD-based constraint solver and an SMT-based constraint solver in parallel for the same set of constraints. The results of the fastest solver are used for the solution space. This gives CRAVE the advantage to use the solver best suited for the given constraints. In the final step (*Post-Generation Analyzer)*, CRAVE uses *Constraints Debugger* and *Coverage Analyzer* to verify if all the constraints have been satisfied, and the coverage goals have been achieved.

CRAVE is not only limited to discrete values i.e., bool, integers, bit-vectors, and sc_int/sc_uint, as used in digital systems. Rather we have extended CRAVE in the most recent version to support *real* values as well, i.e., double data type. This extension covers the analog, as well as mixed-signal systems. Because analog mixed-signal systems use input signals with values in decimals, e.g., sine wave, and CRAVE without real value support cannot handle it. We have selected the Z3 SMT solver [10] as backend because of its ability to solve real valued constraints efficiently, and its ability to provide similar performance as original CRAVE. Our real value extension maps the real value constraints to Z3 *QF_FP* logic, before initiating the solver. An example is shown in next sections.

CRAVE has been integrated with the UVM-SystemC environment [12]. This can be used for automated verification, and regression testing. The randomized solution space of CRAVE can be used in UVM sequences and transactions (using different CRAVE outputs in each sequence), and CRAVE coverage can be used as UVM functional coverage subscriber. The UVM-SystemC environment can be also extended to work with SystemC AMS, so that analog mixed signal systems can also be verified. The synchronization between UVM sequences and transactions with SystemC AMS model of computation is thereby crucial. As it is not feasible to create new transactions for each computation point of an AMS cluster, transactions then contain data for multiple time stamps or switch modes in sources (e.g. the frequency of a sine wave). Therefore, the transactions have to be available before they are used in the AMS cluster. For more details on the synchronization, we refer the readers to [14].

III. Using Constraints within SystemC AMS for Design and Verification

The main focus of the SystemC AMS language is the creation of virtual prototypes that can be reused at later stages in the design process. Thereby, architectural exploration is an important task. The results of this process define the specification for the further design process. In addition, SystemC AMS can also be used to create test environments, which contain system-level tests that also have to run correctly with the finally implemented design.

We propose two kinds of constraints types: parameter constraints, and runtime constraints, as mentioned in Section I, depending on their purpose and when they are used within the SystemC simulation. Figure 2 depicts how *parameter constraints* and *runtime constraints* are applied within the SystemC simulation schedule and both are described in the following subsections.

A. Parameter Constraints

These constraints model parameters which determine the structure of the DUT and cannot be altered during normal simulation. Typical parameters in SystemC AMS can be resistance values of resistors or parameters of other electronic components, which are set during object construction. Modifying these parameters can be done before the DUT is created in the *sc_main* or afterwards during the *before_end_of_elaboration* callbacks, which still allows modification of the SystemC design hierarchy. An example of such a constraint is shown in the following code snippet.

```
class parameter_constraints : public crv_sequence_item {
  crv_variable<double> R1; //Resistor 1
  crv_variable<double> R2; //Resistor 2
  crv_constraint  R1_val{ 1 < R1 (), R1 () < 3.3  }; // 1KOhm to 3.3Kohm
  crv_constraint R2_val{ 0.5 < R2(), R2() < 4.7 }; // 0.5Kohm to 4.7Kohm
  crv_constraint R_overall{ R2()+R1() < 4.7e3 }; // Restrict sum
  parameter_constraints(crv_object_name) {}
};
```

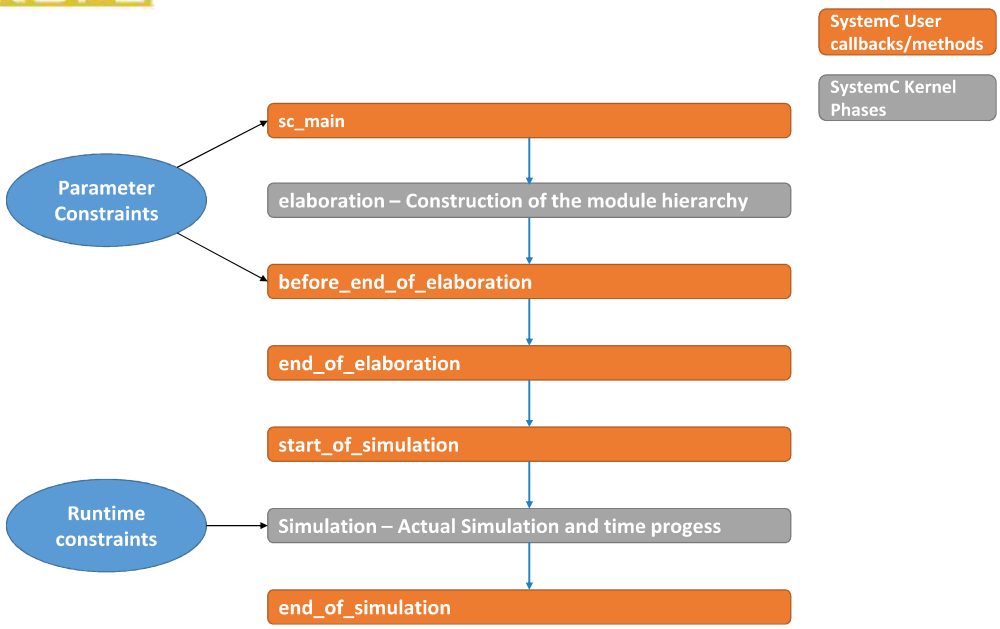

Figure 2: Constraint Solving within SystemC simulation phases

In this example constraints are used to model the allowed resistor values and to specify a correlation between these values. The values of the resistors *R1* and *R2* are restricted to a certain range of values. Using the constraint *R_overall* the sum of the resistor values is also limited. For each run of the SystemC AMS simulation a new set of parameters based on the constraint solver result is generated, that influences the behavior of the DUT. Hereby, it is important to also support real value constraints as most parameters within AMS systems rely on analog values, as shown in the example in Section IV.

B. Runtime Constrainsts

Runtime constraints shape the stimuli at the inputs of a system. The randomization of input values is the basis of UVM, where constraints are used to specify the range and relations of possible input values of a system. This allows to describe general test scenarios, where it is possible to increase coverage by running the simulation longer i.e. increasing the number of test scenarios.

In contrast to parameter constraints, the constraints are solved during the simulation phase of the SystemC simulation, similar to constraints used in SystemVerilog to randomize class objects. Whenever new stimulus is applied to the DUT the randomization of a stimulus object is triggered and new values are assigned. An example of such a constraint is shown in the following code excerpt:

```
class rfmixer_tx: public uvm_randomized_sequence_item
{
public:
  UVM_OBJECT_UTILS(rfmixer_tx)
  crv_variable<double> rf_signal_freq; //Random Variable
  crv_variable<double> rf_signal_level; //Random Variable
  crv_variable<double> lo_signal_freq; //Random Variable
  crv_variable<double> lo_signal_level; //Random Variable
  crv_constraint  lo_level{ lo_signal_level ()  >  (rf_signal_level  + 0.7) }; // 0.7v
  crv_constraint rf_level{ 1.0 < rf_signal_level (), rf_signal_level ()  <  2.5 }; //
1.0v to 2.5v swing)
  crv_constraint rf_freq { 0.5 < rf_signal_freq (), rf_signal_freq () < 15}; // (0.5 MHz
to 15 MHz)
  crv_constraint lo_freq { 0.2 < lo_signal_freq (), lo_signal_freq () < 13, lo_sig-
nal_freq () < rf_signal_freq ()};
  rfmixer_tx(const std::string& name) {}
};
```

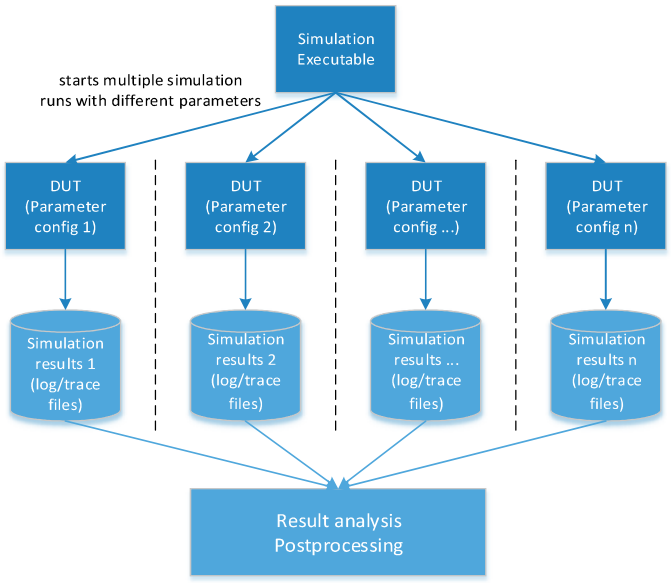

Figure 3: Running simulation in parallel with different Parameter configurations

In this example constraints are embedded using the UVM-SystemC compatibility layer of CRAVE, which allows it to use constraints and random variables within UVM transactions and sequences. Therefore, transactions are derived from *uvm_randomized_sequence_item* instead of *uvm_sequence_item*. The randomized transactions are then sent to the UVM driver and driven via a virtual interface to the DUT.

C. *Parallel Simulation Framework*

Simulation performance is always a bottleneck during system design. Automatic parallelization of SystemC AMS simulations is hard due to the close coupling of processes in the event driven kernel. Therefore, an approach to increase productivity, and speed up verification coverage is to run simulations in parallel for different parameters or input stimulus, rather than decrease simulation time for a single simulation by paralyzing the model. Therefore, we developed a general parallel simulation framework that allows it to easily run test cases with different sets of parameters. This is also possible as the SystemC AMS simulator reference implementations allow license free simulations. In Figure 3 the general idea is shown, from the simulation executable the simulation is spawned to several application processes using a fork mechanism. Each simulation runs has its own SystemC kernel and uses a different random number seed. By giving each run a different seed different random parameters or stimuli sets are created leading to different simulation results.

In the following code excerpt the use of the framework for generating parameter constraints is demonstrated. Before the DUT is created, the parallel simulation is initiated, with a call to *sca_statistics_start*. This call spawns the specified number of 20 parallel simulations, in *Monte Carlo* simulation mode, where each run has its own random number seed. The returned *statistics_handle* contains information, which can be used to identify the run such as a run number. Afterwards, the CRAVE object shown in Section III.A is instantiated and randomized. As each run has its own seed, randomization will return different values. These values are then passed to a parameter object *p_dut*, which is used to construct the DUT and then used in the simulation.

```
int sc_main(int argc, char* argv[]) {
  // spawn 20 parallel runs
  statistics_handle h = sca_statistics_start(MONTE_CARLO, 20);
  // use constraints and  run them
  parameter_constraints c("parameter_constraints");
  c.randomize();
  // Create parameters and assign randomized values
  dut::params p_dut;
  p_dut.p_R1 = c.R1;
  p_dut.p_R2 = c.R2;
```

```
  …
  // Bind to DUT
  dut* i_dut;
  i_dut = new dut("i_dut", p_dut);
  …
};
```

IV. DC-DC CONVERTER CIRCUIT EXAMPLE

To demonstrate the feasibility we consider a DC-DC converter circuit. We build the DUT with different parameters using CRAVE, apply UVM-SystemC for verification and use the proposed parallel simulation. The top level schematic diagram of the system is shown in Figure 4. The system consists of a digital controller, a motor, and DC-DC converter combining SystemC AMS timed data flow (TDF), electric linear network (ELN), and digital event driven model of computation.

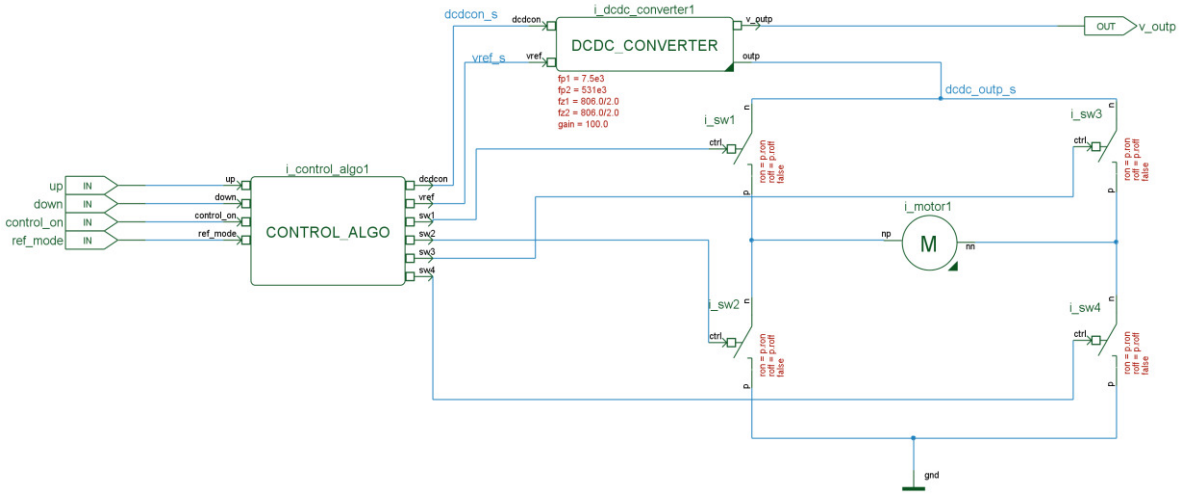

Figure 4: Top level schematic of the motor system with controller and DC-DC converter

The controller decodes the inputs from the system direction inputs (port *up* and *down*) and sets the motor direction switches *i_sw1 – i_sw4* and thereby, controlling the rotation direction of motor. Additionally, the controller sets the reference voltage for the DC-DC converter and turns it on.

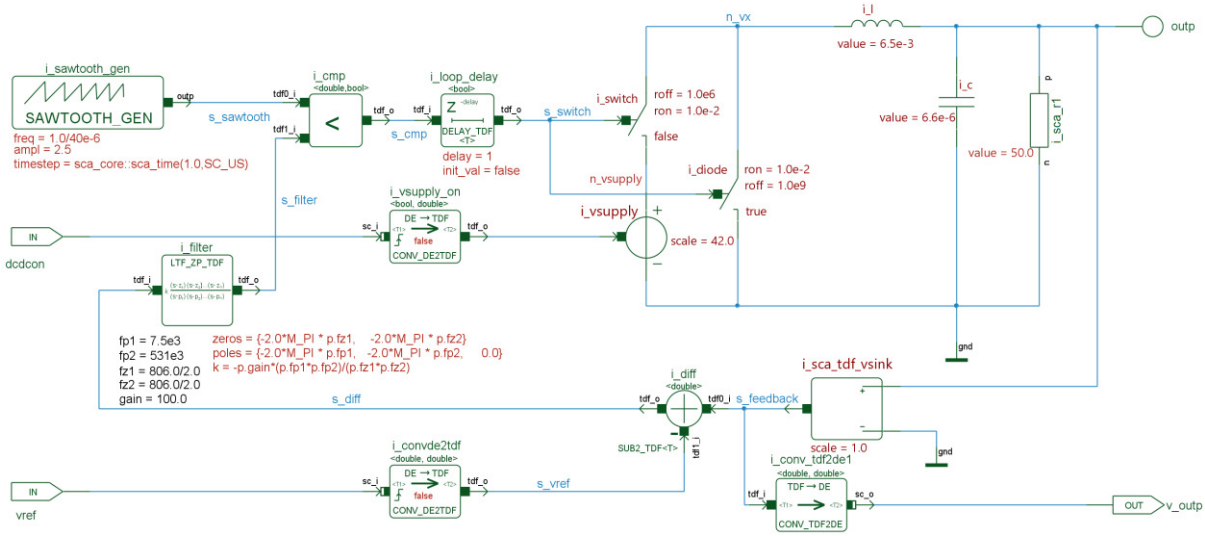

Figure 5: DC-DC converter schematic used for SystemC AMS simulation

In Figure 5 the schematic of the DC-DC converter circuit is shown. It uses the principle of a buck converter where a voltage is stepped down. The general idea of the circuit is to use a saw tooth generator source to generate a PWM signal that turns on and off a voltage source. When the voltage source is connected the current through the inductor increases and produces an opposing voltage, effectively reducing the voltage of the voltage source. To control the output voltage, the output voltage is constantly measured, filtered and compared. The comparison is done against the voltage created by the saw tooth generator to control a PWM signal with a certain frequency, controlling the request output voltage *vref*.

For this system an UVM-SystemC test bench was created, that randomly generates valid inputs to the controller and requests different voltages. A corresponding UVM sequence item is shown in the following code snippet. Randomized inputs are used to determine the direction of the motor and to turn the system on and off. The enumeration types supported by the CRAVE library are used to select the different possible input voltages.

```
CRAVE_BETTER_ENUM(ref_mode_e, DCDC_3V = 0, DCDC_5V = 1, DCDC_7V = 2, DCDC_9V = 3);

class dcdc_control_agent_dcdc_control_agent_tx: public uvm_randomized_sequence_item
{
public:
  UVM_OBJECT_UTILS(dcdc_control_agent_dcdc_control_agent_tx)
  crv_variable<bool> up;
  crv_variable<bool> down;
  crv_variable<bool> control_on;
  crv_variable<ref_mode_e> ref_mode;
  crv_constraint c_enabled{control_on() == true};
};
```

In addition to the input voltage a UVM monitor was created, which checks the stability of the generated output voltage. Therefore, we used the mixed signal checker framework [13], to specify checks for rise time and stability of the output voltage. Using the framework, we can run regressions which also checks the behavior of analog waveforms.

One characteristic of the circuit is that when changing the direction of the motor (change of signal *up* and *down*) a spike in the output voltage can be seen. The height of this spike depends on the load resistor *i_sca_r1*, but also influences the time till the desired voltage is reached. In Figure 6 simulation results are shown which illustrate this behavior. When the direction of the motor is changed a spike on the output voltage *v_outp* can be seen. In addition, results for varying the load are shown (signals *v_outp_run1*- *v_outp_run5*). We used *parameter constraints* as described in Section III.A to specify a range for this load. The range for the resistor was thereby between 10 and 100 ohm. Using the checker framework, we set a maximum voltage that must not be reached. This makes it possible to automatically find values for *i_sca_r1*, which stay in the specified voltage bound.

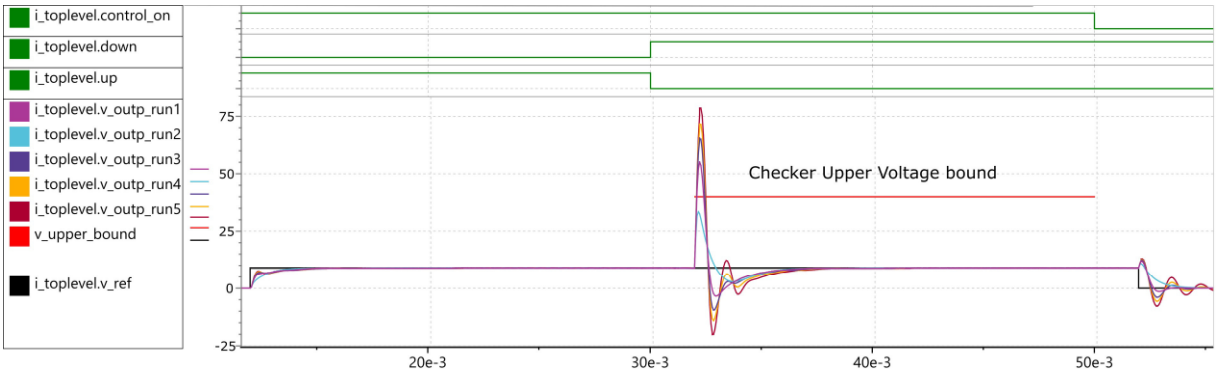

Figure 6: Simulation results for output voltage, for different parameters sets

In addition, by using the parallel simulation framework described in Section III.C we were able to parallelize the simulation runs making it possible to try a lot of different parameter sets. We could run a UVM sequence, testing several modes of the systems for 100 different DUT parameter sets in about 68 seconds on a standard quad core

(8 threads, 3GHz clock speed) laptop. Simulation using only 1 core took 255 seconds, equaling to a speed up of 3.75 times, which is close to the maximum possible speedup of 4.

V. CONCLUSIONS

The use of constraint randomization in the design flow is not only limited to constrained random verification. Constraints can also be used to model the possible design space of a mixed-signal circuit at the system level. Combined with automatic checking of correct and analog behavior and license free parallel simulation this creates a powerful solution to explore and verify more and more complex systems. In future work, we plan to use CRAVE for SystemC AMS testbench qualification [11]. CRAVE can also be used for automated test generation to exercise a specific mutant (bug). Furthermore, we plan to consider system-level verification of mixed-signal platforms, i.e. including software running for instance on top of RISC-V virtual prototypes such as [16, 17].

REFERENCES

[1] "IEEE Standard for Standard SystemC® Analog/Mixed-Signal Extensions Language Reference Manual," in IEEE Std 1666.1-2016, vol.,no.,pp.1-236, April 6 2016 doi: 10.1109/IEEESTD.2016.7448795

[2] "IEEE Standard for Standard SystemC Language Reference Manual," in IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005) , vol., no., pp.1-638, Jan. 9 2012 doi: 10.1109/IEEESTD.2012.6134619

[3] "Universal Verification Methodology 1.2 Class Reference," Accellera System Initiative (2014).

[4] "IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language," in IEEE Std 1800-2012 (Revision of IEEE Std 1800-2009) , vol., no., pp.1-1315, Feb. 21 2013 doi: 10.1109/IEEESTD.2013.6469140

[5] "SystemC Verification Library 2.0.1," Accellera Systems Initiative (2017), http://www.accellera.org/activities/working-groups/systemc-verification

[6] "UVM-SystemC Library 1.0", Accellera Systems Initiative (2017), http://www.accellera.org/activities/working-groups/systemc-verification

[7] http://www.systemc-verification.org/crave

[8] F. Haedicke, H. M. Le, D. Große, and R. Drechsler, "CRAVE: An advanced constrained random verification environment for SystemC,". In International Symposium on System-on-Chip, pp. 1-7, 2012.

[9] L. De Moura and N. Bjørner, 'Satisfiability Modulo Theories: Introduction and Applications', Commun. ACM, vol 54, iss 9, pp. 69--77, 2011.

[10] De Moura, L., & Bjørner, N. (2008, March). Z3: An efficient SMT solver. In International conference on Tools and Algorithms for the Construction and Analysis of Systems (pp. 337-340). Springer, Berlin, Heidelberg..

[11] Hassan, Muhammad, et al. "Testbench Qualification for SystemC-AMS Timed Data Flow Models." Design, Automation and Test in Europe, pp. 857-860. 2018.

[12] S. Gerth, D. Große "UVM goes random – Introducing CRAVE in UVM-SystemC", DVCon Europe 2016, Munich.

[13] Vörtler, T., Einwich, K. "Verification IP for Complex Analog and Mixed-Signal Behavior" DVCon Europe 2017, Munich.

[14] Barnasconi, M., Dietrich, M., Einwich, K., Vörtler, T., Lucas, R., Chaput, J.P., Pecheux, F., Wang, Z., Cuenot, P., Neumann, I., Nguyen, T., 2015. UVM-SystemC-AMS Framework for System-Level Verification and Validation of Automotive Use Cases. Design & Test, IEEE PP, 1. https://doi.org/10.1109/MDAT.2015.2427260

[15] M. Strickland, H. J. Zhang, J. Chen, D. Goswami, and A. Wakefield, "Soft Constraints in SystemVerilog: Semantics and Challenges,". In Design and Verification Conference (DVCON), 2012

[16] V. Herdt, D. Große, H. M. Le, and R. Drechsler. Extensible and configurable RISC-V based virtual prototype, Forum on Specification and Design Languages (FDL), 2018.

[17] https://github.com/agra-uni-bremen/riscv-vp