2015 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

Boosting SystemC-based Testbenches with Modern C++ and Coverage-Driven Generation

Hoang M. Le, University of Bremen, Bremen, Germany (*hle@cs.uni-bremen.de*)

Rolf Drechsler, University of Bremen & DFKI GmbH, Bremen, Germany (*Rolf.Drechsler@dfki.de*)

***Abstract*—Constrained randomization and functional coverage are two major pillars of the Universal Verification Methodology (UVM), which is now also available for SystemC. In this paper, we present a novel implementation of randomization and coverage for SystemC. This implementation improves the current state-of-the-art in various ways: better usability for randomization based on modern C++ syntax, more flexible coverage models, and for the first time in SystemC-based verification, coverage-driven generation is available to automate coverage closure. Our approach automatically integrates uncovered coverage bins into the stimuli model as additional constraints, which enable the underlying constraint solver to generate values filling these coverage holes directly. Experiments on a practical verification scenario show that coverage closure is achieved much faster with our approach.**

Keywords—SystemC; UVM; constrained random verification; coverage-driven generation; C++11; constraint; coverage

I. INTRODUCTION

Constrained Random Verification (CRV) [1] has become the prevalent method for functional verification due to its advantages over traditional directed testing approaches. Creating a huge number of test vectors manually is not required anymore. Instead, these values will be automatically generated from the legal input space defined by a set of constraints. Furthermore, CRV can reveal bugs using corner-case stimuli that might have not been thought of. The verification process is driven by functional coverage, a user-defined metric intended to measure the thoroughness of the verification, i.e. to which extent the functionality of the *design under test* (DUT) has been verified by generated stimuli. The CRV methodology has been standardized by *Accellera Systems Initiative* (ASI) as the *Universal Verification Methodology* (UVM) [3], which is a SystemVerilog class library that enables the creation of structured and reusable testbench.

In the last few years, there has been much interest and effort in porting UVM to the standardized C++-based system-level language SystemC [2]. The two most notable implementations are the *System Verification Methodology* (SVM) [6] and more recently, UVM-SystemC [7]. SVM is available as a complete package integrating its own implementation of functional coverage and the CRAVE library [4], which is also separately available as open-source software at www.systemc-verification.org/crave, for constrained randomization. However, SVM is not fully compliant to the UVM standard. On the other hand, UVM-SystemC attempts to give users the same look and feel of UVM by providing identical constructs as defined in the standard. For UVM-SystemC, constrained randomization and functional coverage are provided as an extension library SCVX [8]. In addition to basic support for floating point data types, the user API offered by this library resembles the SystemVerilog API for randomization and coverage more closely than the previous effort by SVM (although CRAVE is still used as backend for randomization).

Despite these recent good efforts, there is still much room for improvements. In this paper, we present a novel implementation of randomization and coverage for SystemC improving the current state-of-the-art in the following aspects:

1. We show that the use of C++11, the latest major version of the ISO standard for C++ [11], enables even more SystemVerilog-like API for the randomization layer. This would have not been possible with any previously known implementations which are based on the previous standard C++03.
2. The existing coverage API in both SVM and SCVX focuses mostly on supporting coverage models that group sampled values of variables into ranges, and thus, is not flexible enough for more complex coverage models. For those, we propose an API extension towards the expression layer which is used in constraints.

3. This extension also lays the foundation for the last improvement, which targets a major challenge in CRV: automated coverage closure. As both constraint and coverage can now be described using the same expression layer, it is possible to integrate coverage expressions directly into the constraint solving process. This effectively enables *Coverage-driven Generation* (CDG) for the first time for SystemC-based verification.

The implementation is built on top of CRAVE 2.0 [5] and will also be available as open-source software.

II. A Practical Example

We start with the description of a practical example that will be used throughout the paper to demonstrate our solutions. The example involves the verification of a SystemC TLM model of the interrupt controller core IRQMP from Cobham Gaisler [12]. In this particular scenario, the interrupt forwarding functionality of the core is triggered by writing to its interrupt level register (*level_reg*) and interrupt force register (*force_reg*). Each interrupt, indicated by a number from 1 to 15, is forced if the corresponding bit of *force_reg* is set, respectively. The interrupt controller prioritizes the forced interrupts and forwards the unmasked interrupt with highest priority to its output port, which is connected to a LEON3 core. Each interrupt is assigned to one of two levels (0 or 1) as set by the *level_reg*, i.e. the level of interrupt k is equal to the value of bit k of *level_reg*. Interrupts of level 1 are higher prioritized than interrupts of level 0. Within each level, an interrupt with larger index has higher priority. The interrupt controller must be verified with multiple different inputs, which is achieved via constrained randomization of *level_reg* and *force_reg*.

III. C++11-based Syntax For Constrained Randomization

```
struct irqmp_regs : public rand_obj {
  randv<unsigned> level_reg;
  randv<unsigned> force_reg;

  irqmp_regs(rand_obj* parent_obj)
      : level_reg(this), force_reg(this) {
    constraint(level_reg() < (1 << 16));
    constraint((level_reg() & 1) == 0);
    constraint((force_reg() & 0xFFFF0001) == 0);
  }
};
```

```
struct irqmp_regs : public scvx_rand_object {
  scvx_rand<unsigned> level_reg;
  scvx_rand<unsigned> force_reg;

  scvx_constraint level_reg_cstr;
  scvx_constraint force_reg_cstr;

  irqmp_regs(scvx_name)
      : level_reg("level_reg")
      , force_reg("force_reg")
      , level_reg_cstr("level_reg_cstr")
      , force_reg_cstr("force_reg_cstr") {
    level_reg_cstr((level_reg() < (1 << 16))
                   && ((level_reg() & 1) == 0));
    force_reg_cstr((force_reg() & 0xFFFF0001) == 0);
  }
};
```

Figure 1 Constrained object in CRAVE (left) and SCVX (right)

Figure 1 shows two equivalent sets of constraints for the described registers, specified using CRAVE and SCVX, respectively. The constraints specify that only the bits from position 1 to 15 are to be randomized.The other bits are declared as reserved in the specification and always set to 0 in our stimuli. The constraints for *level_reg* and *force_reg* are essentially the same but expressed in two different ways. For C++-based constrained randomization, two kinds of objects are needed: randomized variable of primitive types (*randv* and *scvx_rand*) and randomized object container (*rand_obj* and *scvx_rand_object*). It is essential to build a hierarchy of these objects. As can be seen, CRAVE requires the parent object to be specified directly for each constructed object. In SCVX, each object must be given a name (preferably identical to the C++ variable name), so that the hierarchy can be automatically built (SystemC itself employs a similar technique). While the CRAVE scheme is more compact, the named object hierarchy of SCVX offers an important advantage that allows the randomized objects and constraints to be identified and manipulated dynamically during runtime by their names. However, it is still very inconvenient that multiple separated steps are needed to properly construct an object. For example, an SCVX constraint must be first declared as a member variable, which is then initialized with a name in the constructor's initialization list, and finally assigned to a constraint expression in the constructor body. C++11 provides a solution to this problem by support *in-place initialization* of member variables. In combination with the new *brace initialization* notation, the same set of constraints can be expressed in our new implementation as shown on the left hand side of Figure 2.

```
struct irqmp_regs : public crv_sequence_item {
  crv_variable<unsigned> level_reg { "level_reg" };
  crv_variable<unsigned> force_reg { "force_reg" };

  crv_constraint level_reg_cstr { "level_reg_cstr",
    level_reg() < (1 << 16),
    (level_reg() & 1) == 0
  };

  crv_constraint force_reg_cstr { "force_reg_cstr",
    (force_reg() & 0xFFFF0001) == 0
  };

  irqmp_regs(crv_object_name) { }
};
```

```
struct irqmp_regs : public crv_sequence_item {
  CRV_VARIABLE(unsigned, level_reg);
  CRV_VARIBALE(unsigned, force_reg);

  CRV_CONSTRAINT(level_reg_cstr,
    level_reg() < (1 << 16),
    (level_reg() & 1) == 0
  );
  CRV_CONSTRAINT(force_reg_cstr,
    (force_reg() & 0xFFFF0001) == 0
  );

  irqmp_regs(crv_object_name) { }
};
```

Figure 2 C++11-based syntax for constrained randomization without macros (left) and with macros (right)

As can be seen, the named hierarchy scheme is kept, but both variables (*crv_variable*) and constraints (*crv_constraint*) can now be initialized in one single step, which is more elegant and resembles SystemVerilog syntax very closely. If the use of C macros is allowed, it is possible to avoid typing a name twice (e.g. level_reg and "level_reg") as shown on the right hand side of the figure.

IV. EPXRESSION-BASED FUNCTIONAL COVERAGE

After the stimuli for the interrupt controller have been constrained, we continue with the definition of a coverage model. We require that each interrupt k between 1 and 15 must be forwarded at least once. An interrupt will be forwarded if one of the following conditions holds:

1. It is the forced interrupt with largest index on level 1
2. It is the forced interrupt with largest index on level 0 with no other forced interrupts on level 1.

These two scenarios should be covered at least once in the coverage model. Please note that each scenario involves values of both variables *level_reg* and *force_reg*. Now, assuming the coverage API provided by either SVM or SCVX will be used, first, we need to define two coverpoints for *level_reg* and *force_reg*, respectively. Each coverpoint consists of a collection of coverage bins, which define ranges for the value of the variable to be sampled. Because both variables are involved in the scenarios, we need to define a new coverpoint which is the cross coverage of the coverpoints for *level_reg* and *force_reg*. While theoretically possible, it is rather challenging to come up with the necessary ranges for the coverage bins to capture the scenarios.

Description	Formulation as Expression
Set of forced interrupts on level 1	*forced_level_1 = (force_reg & level_reg)*
Set of forced interrupts on level 0	*forced_level_0 = (force_reg & ~level_reg)*
Interrupt k is forced on level 1	*((forced_level_1 >> k) & 1) == 1*
Interrupt k is forced on level 0	*((forced_level_0 >> k) & 1) == 1*
No interrupt is forced on level 1	*forced_level_1 == 0*
Index of every forced interrupt on level 1 is smaller than or equal to k	*forced_level_1 < (2 << k)*
Index of every forced interrupt on level 0 is smaller than or equal to k	*forced_level_0 < (2 << k)*

Table 1 Subexpressions used in the formulation of the two conditions

On the other hand, the two scenarios can be formulated easily using logic expressions. With the help of the subexpressions defined in Table 1, the formulation is as follows. Interrupt k is the forced interrupt with largest index on level 1, iff it is forced on level 1 and the index of every forced interrupt on level 1 is smaller than or equal to k, which corresponds to the following conjunction:

$$(((forced\_level\_1 >> k) \& 1) == 1) \&\& (forced\_level\_1 < (2 << k)).$$

Similarly, interrupt k is the interrupt with largest index on level 0 with no other forced interrupts on level 1, iff it is forced on level 0 and the index of every forced interrupt on level 0 is smaller than or equal to k and no interrupt is forced on level 1:

$$(((forced\_level\_0 >> k) \& 1) == 1) \&\& (forced\_level\_0 < (2 << k)) \&\& (forced\_level\_1 == 0).$$

Our implementation of coverage provides a very convenient way to describe these formulations. As can be seen in Figure 3, the same expression layer, which is used in constraints, is also employed to specify coverpoints. First, we declare two variables *lr* and *fr*, which are at the time of construction unrelated to *level_reg* and *force_reg*. Then, we define two coverpoints *fwd_lvl_1* and *fwd_lvl_0* to capture scenarios where interrupt on level 1 and level 0 will be forwarded, respectively. In the constructor body, for each interrupt k, its coverage bin as an expression is added to the corresponding coverpoint accordingly. Two helper functions *forced_lvl_1* and *forced_lvl_0* are used to avoid repeating subexpressions.

```
struct my_covergroup : public crv_covergroup {
  crv_variable<unsigned> lr { "lr" };
  crv_variable<unsigned> fr { "fr" };

  crv_coverpoint fwd_lvl_1 { "fwd_lvl_1" };
  crv_coverpoint fwd_lvl_0 { "fwd_lvl_0" };

  expression forced_lvl_1() { return make_expression(fr() &  lr()); }
  expression forced_lvl_0() { return make_expression(fr() & ~lr()); }

  my_covergroup(crv_object_name) {
    for (int k = 1; k < 16; k++) {
      fwd_lvl_1.bins( (((forced_lvl_1() >> k) & 1) == 1) && (forced_lvl_1() < (2 << k)) );
      fwd_lvl_0.bins( (((forced_lvl_0() >> k) & 1) == 1) && (forced_lvl_0() < (2 << k)) && (forced_lvl_1() == 0) );
    }
  }
};
```

Figure 3 Coverage model using expressions

To collect coverage of the generated stimuli, we need to connect an *irqmp_regs* object to an instance of the coverage model *my_covergroup*, which can be done shown in the following.

```
irqmp_regs regs("regs");

my_covergroup cg("cg");
cg.lr.bind(regs.level_reg);
cg.fr.bind(regs.force_reg);

assert(regs.randomize());
cg.sample();
```

After an object *regs* of type irqmp_regs and an object *cg* of type *my_covergroup* are created, we connect the variables *lr* and *fr* of the covergroup *cg* to *level_reg* and *force_reg* of *regs*, respectively. The connections are established via calling member function *bind()* of *crv_variable*. Then, after each call to *randomize()* to generate values for *level_reg* and *force_reg*, the function *sample()* can be invoked from *cg* to check whether these values fit into a specified coverage bin and increase the number of hits accordingly.

V. COVERAGE-DRIVEN GENERATION

As both constrained random stimuli and coverage model for the interrupt controller have been defined, we can now go ahead with the verification. In the default mode, uniform distribution is used to generate the values for *level_reg* and *force_reg*. The coverage model consists of two coverpoints and each of these contains 15 coverage bins (for each interrupt numbered from 1 to 15; cf. loop in Figure 3). After 100 iterations, 5 bins of coverpoint *fwd_lvl_1* and 14 bins of *fwd_lvl_0* are still uncovered. The results after 1000, 10000 and 20000 generated stimuli for each coverpoint are shown in Figure 4 and Figure 5, respectively. As can be seen, tests, where interrupts with lower index will be forwarded, are generally less frequently generated. The coverage bins of *fwd_lvl_1* are rather easy to hit in comparison with the bins of *fwd_lvl_0*. The last few bins of *fwd_lvl_0* are very hard to hit: after 1000, 10000 and 20000 iterations, the number of uncovered bins of *fwd_lvl_0 is* 9, 6 and 5, respectively.

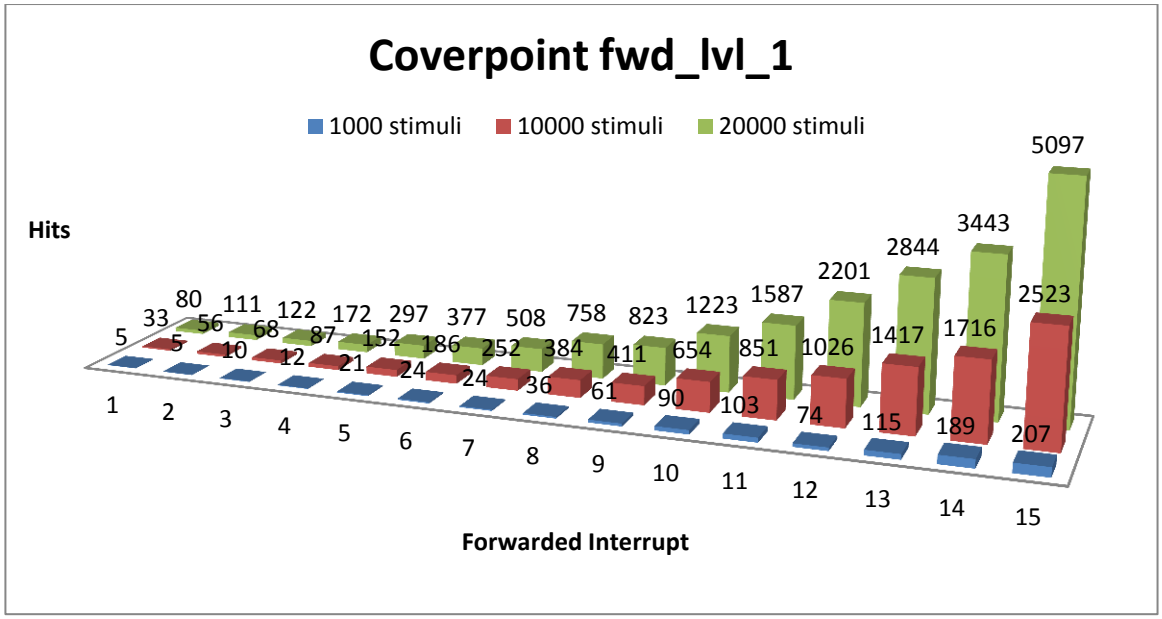

Figure 4 Coverpoint fwd_lvl_1

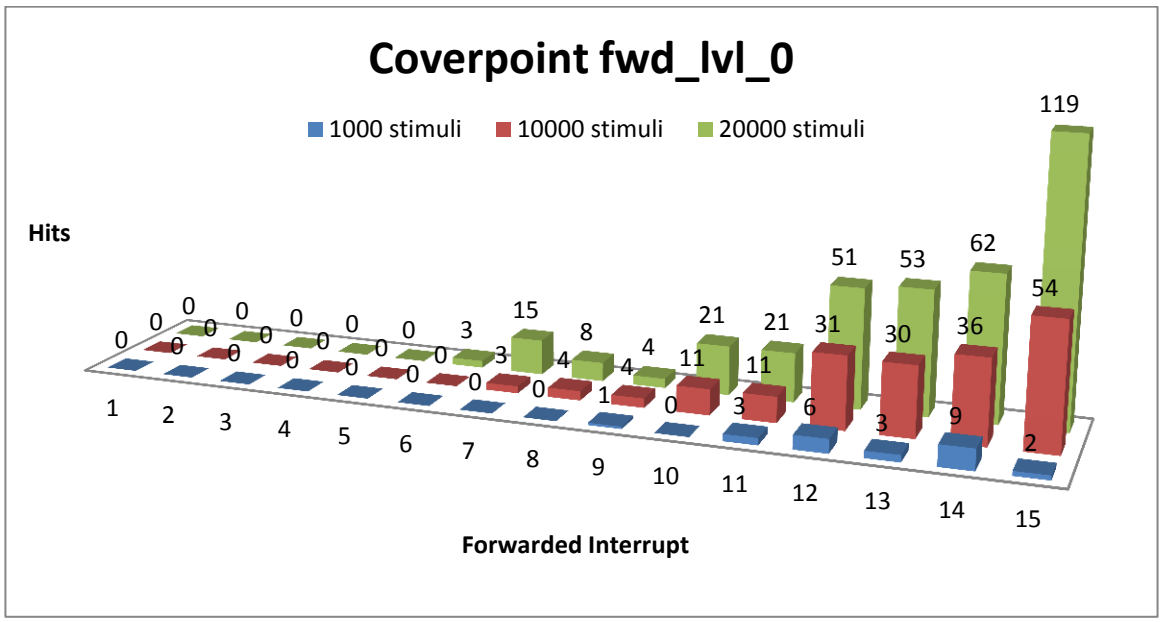

Figure 5 Coverpoint fwd_lvl_0

This is due to the well-known gap between stimuli model (i.e. constraints) and coverage model in practice. There are several ways for verification engineers to overcome this problem. The most obvious is to manually write a set of directed tests which fill the uncovered bins. However, this requires a lot of work and generally does not scale to large stimuli/coverage models. The second option is to add more constraints or to tune the distribution of variables to direct the generation towards the coverage holes. This solution also requires a lot of manual work and deep knowledge from verification engineers about both the stimuli and coverage model. The best option is that the underlying constraint solver can "understand" the coverage model and automatically generate values filling the uncovered bins. In this case, the stimuli generation process is said to be driven by the coverage model (hence, coverage-driven generation).

With the coverage model being formulated using the same expression layer used by the constraints, CDG can be directly enabled in our implementation. After the variables of the coverage model are bound to the stimuli model, users can start CDG by calling the function *goal()* (for our example: *regs.goal(cg)*). The uncovered bins will be identified and converted to a set of "cover"-constraints only used internally by the constraint solver. Each

time *randomize()* is called, the constraint solver tries to generate a new solution, which satisfies all specified constraints and *at least* one "cover"-constraint. This satisfied "cover"-constraint will then be removed from the set, because the corresponding bin is now covered. For the verification of the interrupt controller, CDG needs exactly 30 iterations to hit all 30 defined bins if it is enabled from the beginning. When CDG is enabled after 1000, 10000, and 20000 iterations, the number of additional iterations needed for coverage closure in each case is 9, 6, and 5, respectively. These numbers correspond directly to the number of uncovered bins described above. Clearly, with CDG, coverage closure can be achieved much faster.

The main limitation of the current CDG is that there must be a direct relation between the stimuli model and the coverage model. Furthermore, this relation must be expressible using the expression layer (Please note that a variable of the coverage model can be bound to an expression involving multiple variables of the stimuli model instead of a single variable as demonstrated in the example). Otherwise, other solutions such as machine learning are required (please see [10] for a survey). Recently, Cadence has integrated a slightly similar CDG [9] into the constrained random stimuli generator Intelligen. However, this solution is only available for users of Cadence's Specman and the coverage model is used to tune the distribution automatically, whereas our solution fills uncovered bins directly. At the time of writing, CDG only considers bins that have never been hit. We are currently working on extending CDG to also consider bins whose number of hits has not reached an user-defined threshold. The integration of machine learning techniques will be investigated in future work as well.

VI. CONCLUSIONS

In this paper, a new framework for SystemC-based verification has been presented. The framework provides support for both constrained randomization and functional coverage. Several novelties have been introduced in the framework. First, the use of modern initialization constructs from the latest major C++ standard C++11 enables a very convenient and SystemVerilog-like API for constrained randomization. This is not possible with state-of-the-art implementations which still use C++03. Second, in our frame work, randomization and functional coverage are built from the same expression layer. The use of expressions offers more flexibility for the specification of complex coverage models, which are not naturally expressible by a set of ranges. Last but not least, thanks to the shared expression layer, uncovered bins from the coverage model can be automatically converted to an additional set of constraints. These constraints directly drive the underlying solver to generate values that fill the coverage holes. To the best of our knowledge, there exists no other framework for SystemC-based verification with such a form of automated coverage closure. Furthermore, the framework will be made open-source facilitating further development (and standardization).

ACKNOWLEDGMENT

This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project EffektiV under contract no. 01IS13022E and by the German Research Foundation (DFG) within the Reinhart Koselleck project DR 287/23-1. The authors would like to thank Daniel Große for taking the time to review an early version of this paper. We are also grateful for the valuable comments and suggestions from the anonymous reviewers.

REFERENCES

[1] J. Yuan, C. Pixley and A. Aziz, Constraint-based verification, 1st ed. New York, NY: Springer, 2006.

[2] 'IEEE Standard for Standard SystemC Language Reference Manual', IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005), pp. 1-638, 2012.

[3] Accellera Systems Initiative, Standard Universal Verification Methodology (UVM), www.accellera.org/downloads/standards/uvm/.

[4] 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.

[5] H. M. Le and R. Drechsler, "CRAVE 2.0: The next generation constrained random stimuli generator for SystemC,". In Design and Verification Conference Europe (DVCON Europe), 2014.

[6] M. F.S. Oliveira, C. Kuznik, H. M. Le, D. Große, F. Haedicke, W. Mueller, R. Drechsler, W. Ecker, V. Esen, "The system verification methodology for advanced TLM verification,". In International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), pp. 313-322, 2012.

[7] M. Barnasconi, F. Pêcheux, T. Vörtler, and K. Einwich, "Advancing system-level verification using UVM in SystemC,". In Design and Verification Conference (DVCON), 2014.

[8] T. Vörtler, T. Klotz, K. Einwich, Y. Li, Z. Wang, M.-M. Louerat, J.-P. Chaput, F. Pecheux, R. Iskander, and M. Barnasconi, "Enriching UVM in SystemC with AMS extensions for randomization and functional coverage,". In Design and Verification Conference Europe (DVCON Europe), 2014.

[9] M. Teplitsky, A. Metodi, and R. Azaria, "Coverage driven distribution of constrained random stimuli,". In Design and Verification Conference (DVCON), 2015.

[10] C. Ioannides and K. I. Eder, "Coverage-directed test generation automated by machine learning -- a review,". In ACM Transactions on Design Automation of Electronic Systems (TODAES), Volume 17, Issue 1, January 2012.

[11] C++ Standardization – Current Status, https://isocpp.org/std/status.

[12] Cobham Gaisler, "GRLIB IP Core User's Manual (version 1.4.1 - b4156)", http://gaisler.com/products/grlib/grlib.pdf.