Verification of a CAN bus model in SystemC with functional coverage

Gilles B. Defo, Christoph Kuznik, Wolfgang Müller
University of Paderborn/C-LAB
Faculty of Electrical Engineering
Computer Science and Mathematics
33102 Paderborn, Germany
{defo, kuznik, wolfgang}@c-lab.de

***Abstract*—Many heterogeneous embedded systems, for example industrial automation and automotive applications, require hard-real time constraints to be exhaustively verified - which is a challenging task for the verification engineer. To cope with complexity, verification techniques working on different abstraction levels are best practice. SystemC is a versatile C++ based design and verification language, offering various mechanisms and constructs required for embedded systems modeling. Using the add-on SystemC Verification Library (SCV) elemental constrained-random stimuli techniques may be used for verification. However, SCV has several drawbacks such as lack of functional coverage. In this paper we present a functional coverage library that implements parts of the IEEE 1800-2005 SystemVerilog standard and allows capturing functional coverage throughout the design and verification process with SystemC. Moreover, we will demonstrate the usability of the approach with a case study working on a CAN bus model written in SystemC.**

I. Introduction

In several domains like industry automation and automotive systems, the verification of hard real-time constraints and verification of functional and non-functional properties for integrated system behaviors is essential. For example, machine-aided assembly lines, where process steps of automatic units and human workers intertwine with each other require a safe and deterministic behavior for the team play. In automotive applications multiple field busses running in parallel, exchanging messages with each other need to be verified to guarantee safety-critical behavior. Verifying these systems is a time consuming and tedious task. To cope with always more complexity and to boost productivity, more efficient verification techniques and technologies were introduced through the last years like the notion of functional verification [1] which supports features such as verification of assertions, constrained-based random test pattern generation, and functional coverage. This article, focuses on the implementation of a functional coverage library in SystemC and applies it on the verification of a CAN bus case study.

Functional coverage is a user defined metric to investigate to which extend the functionality of a given design under test (DUT) has been verified by simulation runs. As such, functional coverage keeps track of value assignments and changes of expressions and conditions within the code. Thus, it is not verified if the DUT is working properly rather than just gives information of the quality of the test pattern with respect to the user-defined metrics. The task to verify the actual simulation behavior of the DUT functionality with the expected one still has to be accomplished by implementing a testbench. So, functional coverage can tell if a property was executed at the right time, in the right order and context. In the ideal case of a close coupling of a testbench and functional coverage analysis, the information can even be processed at runtime to steer the constrained random stimuli generation. While the IEEE 1800 SystemVerilog [2] standard and other Hardware Verification Languages (HVL) such as *e* incorporate functional coverage language features, these are neither available in the IEEE 1666 SystemC standard [3] nor in any publicly available SystemC library. To overcome this, we developed a SystemC functional coverage library and testbench and applied it to a SystemC model of a Controller Area Network (CAN) based on the standard described in [4].

In this article, we first introduce our SystemC implementation of the IEEE 1800-2005 SystemVerilog coverage language features in Section II. The fundamentals of the CAN Controller bus model are introduced in Section III. Section IV outlines the testbench architecture and components of the implemented Controller Area Network (CAN) case study. Thereafter, scenarios of the verification environment are discussed in detail in Section V. Section VI compares our approach with existing standards and related work in the area of functional coverage within SystemC before we close with a conclusion in Section VII.

II. SystemC Coverage Library

The SystemC functional coverage (SCFC) library partly implements the `covergroup` concept of the IEEE 1800-2005 SystemVerilog standard [2] via a singleton Factory class implemented in SystemC. In details, the library supports the definition of:

- bins to collect hits for variable ranges of the coverpoint
- default bins to capture all non-specified values
- ignore bins, i.e., values to ignore and disregard for coverage computation
- illegal bins, where a bin hit calls `sc_stop()`
- transition bins to define value sequences and evaluation with help of user-defined sample function

- cross bins to cover cross-products with binsof() and intersect() operators

following the instantiation hierarchy as defined by the SystemVerilog standard. The SCFC factory is the main facility of the library, providing all necessary setup and management API calls for the creation and administration of every element of the implemented covergroup concept. Moreover, the factory provides the administration of the coverage database in order to store collected functional coverage information. The database has to fulfill two requirements. It has (i) to capture already sampled coverage information prior to the next run and (ii) to save this data after the test, which is supported by a set of convenience API functions. The `set_coverage_db_name(`*name*`)` function defines where to store the collected data. By `load_coverage_db(`*name*`)`, this data may be loaded and evaluated at later steps. To compute the functional coverage of all covergroup types, the factory provides the method `get_coverage()`.

```
1)  SC_MODULE (cg)
2)  {
3)     SC_HAS_PROCESS(cg);
4)
5)     cg(sc_module_name instname /*,...*/ ):sc_module(instname)
6)     {
7)        ...
8)        covergroup = SVFAC->new_SVCovergroup(this, “cg”, instname /*,...*/ );
9)        ...
10)    }
11) }
12) ...
13) cg* cg_inst = new cg(“cg_inst”, /*...*/ );
```

Fig. 1. Definition and instantiation of a `covergroup` in SystemC

Figure 1 shows the abstract principles of the definition and instantiation of a `covergroup` using our library. A covergroup is created using a SystemC `SC_MODULE` macro which implicitly results in a C++ class definition. The `covergroup` class is defined as a `friend`, allowing access to variables of the surrounding class. In line 5, the constructor of the covergroup is defined. The method `new_SVCovergroup(SVRef scm, char* name, const char* instname,/*...*/)` creates a new covergroup type `name` and a associated first instance called `instname`. The actual instantiation is shown in line 13. The instance `cg_inst` (with the parameter `cg_inst`) is constructed via the constructor of `SC_MODULE cg`. This also leads to the creation of the covergroup type `cg` to which the instance is assigned.

The pointer `scm` is used both for the type and the instance and saves the hierarchical relation, pointing to the first instance. `scm` is a reference of the `SC_MODULE` of the covergroup, whereas SVRef is a just macro for `void*`. `name` identifies the name of the covergroup, `instname` the name of the instance of the covergroup. Moreover, additional parameters are possible and indicated by the `...` characters. The constructor allows the definition of a `SC_METHOD sample()` method in line 7. This sensitivity method allows defining on what signal the sampling of the covergroup is performed, for example, on a rising or falling edge of a certain signal. This feature is necessary if the covergroup shall automatically cover variable in dependence of a clock signal. So far the library has support for automatically covering signals of type `sc_signal<int>` or via an explicit API `sample()` function call. Once the covergroup is created, the actual coverage elements such as coverpoints, cross-coverpoints, bins, transition bins, illegal bins etc. can be defined starting in line 9. The order of the definition is the same as in the SystemVerilog standard and has to be respected.

In the following, two more detailed examples of the SCFC library coverage features shall be illustrated. In Figure 2 an example from the SystemVerilog specification is shown. Here, the covergroup `cg` has one defined coverpoint `v_a` with four associated bins definitions and a default bin. In this example, the bins are not derived automatically (for every possible value) rather than they are defined to capture certain variable ranges. Additionally, the coverage computation only considers the defined ranges. `Bins a` covers the range 0 to 63, and the single value 65. The notation `bins b[]` creates as many bins as values within the specified interval. The same holds for `bins c[]`. `bins others[]` holds all non-specified values. These values are not considered during coverage computation.

```
1)  bit [9:0] v_a;
2)
3)     covergroup cg @(posedge clk);
4)        coverpoint v_a
5)        {
6)           bins a = {[0:63], 65};
7)           bins b[] = {[127:150], [148:191]};
8)           bins c[] = {200, 201, 202};
9)           bins d = {[1000:1023]};
10)          bins others[] = default;
11)       }
12) endgroup
```

Fig. 2. A SystemVerilog `covergroup`

```
1)  SC_MODULE (cg)
2)  {
3)  public:
4)     SC_HAS_PROCESS(cg);
5)
6)     cg(sc_module_name instname, CGt309* outer) : sc_module(instname)
7)     {
8)        SC_METHOD(sample);
9)        sensitive_pos << outer->clk;
10)       dont_initialize();
11)
12)       covergroup = SVFAC->new_SVCovergroup(this, "cg", instname, outer);
13)       SVFAC->new_SVCoverpoint(this, "v_a", &(outer->v_a));
14)             SVFAC->new_SVBins(this, "a", 1, 4, 0, 63, 65, 65);
15)             SVFAC->new_SVBins(this, "b", AUTOBINS, 4, 127, 150, 148, 191);
16)             SVFAC->new_SVBins(this, "c", AUTOBINS, 6, 200, 200, 201, 201,
                                                                    202, 202);
17)             SVFAC->new_SVBins(this, "d", 1, 2, 1000, 1023);
18)             SVFAC->add_default("others");
19)    }
20)};
```

Fig. 3. SystemC implementation of the SV covergroup from Figure 2

The corresponding SystemC implementation is shown in Figure 3. `AUTOBINS` is the equivalent to the SystemVerilog `[]` notation and creates as many bins as present values in the defined range. The definition of a certain number of bins is done via the third parameter of `new_SVBins()`. The default bin is defined in line 18. The `add_default()` method computes all non-covered values and adds them to the default bin. Bins which are created after this call will be ignored.

In Figure 4, an example for the original SystemVerilog notation for `ignore` and `illegal` bins is displayed. `Ignore` bins specify values or ranges which shall be ignored while covering the expression, even if they are included in the declaration of other bins. `Illegal` bins specify values on which the simulation shall be halted. In this case, a user-defined error message is shown. A cross-coverpoint allows computing the cross product on all values of given coverpoints. As this may result in a large data sets, it is useful to limit the product on certain bins. Therefore, this example makes just use of the two operators `binsof` and `intersect`. The `binsof` operator refers to the bins of an existing coverpoint. By the `intersect` expression the possible candidates can be limited using specific values or intervals.

```
1)  bit [3:0] a, b;
2)
3)  covergroup cg (int bad)  @(posedge clk);
4)     cross a, b
5)     {
6)        ignore_bins ign_vals = binsof(a) intersect {5, [1:3]};
7)        illegal_bins ill_vals = binsof(b) intersect {bad};
8)     }
9)  endgroup
```

Fig. 4. SystemVerilog `ignore` and `illegal` bins

The corresponding implementation of the `illegal` and `ignore` concept within the SCFC library is shown in Figure 5. To define a cross-coverpoint, a coverpoint has to be declared first. Using the methods `new_illegal_SVCrossBins` and `new_ignore_SVCrossBins` the respective bins are created. Moreover, the cross product is limited using the `intersect` operator and an expression consisting of boolean operators `!`, `&&`, and `||`, which are also supported by the SystemC library. Transition bins are support in two ways. First, it is possible to define a custom sample function whereas return value true leads to a bin hit. Second, transition vectors are assigned to a transition bin and are elaborated with a vector matching class.

In summary, our SystemC functional coverage implementation allows the definition of coverpoints with bins, transition bins, illegal bins, ignore bins, default bins as well as cross-coverpoints. As it is implemented as a library and based on the factory concept, the solution is highly flexible and can be easily combined with existing SystemC designs to leverage functional coverage. Compared to SystemVerilog, the current implementation does not support clocking block signals, conditional guards to avoid sampling and wildcards in transition bins.

```
1)  cg (sc_module_name instname, outer* o, int bad) : sc_module(instname)
2)  {
3)     SC_METHOD(sample);
4)        sensitive_pos << o->clk;
5)        dont_initialize();
6)
7)     covergroup = SVFAC->new_SVCovergroup(this, "cg", instname, o);
8)        SVCoverpoint* a = SVFAC->new_SVCoverpoint(this, "a", o->sig);
9)        SVCoverpoint* b = SVFAC->new_SVCoverpoint(this, "b", o->sig);
10)       SVFAC->new_SVCrossCoverpoint("ab");
11)          SVFAC->add_SVCoverpoint_to_cross(a);
12)          SVFAC->add_SVCoverpoint_to_cross(b);
13)          SVCrossBins* ign_vals = SVFAC->new_ignore_SVCrossBins
                                  ("ign_vals", binsof_intersect(a, 4, 5, 5, 1, 3));
14)          SVCrossBins* ill_vals = SVFAC->new_illegal_SVCrossBins
                          ("ill_vals", SVFAC->binsof_intersect(b, 2, bad, bad));
15) }
```

Fig. 5. SystemC implementation of Figure 4

III. Controller Area Network (CAN) model

In our case study we use a CAN controller model implemented in SystemC. The architecture of the model is depicted in Figure 6. This architecture was motivated by the Standalone CAN controller SJA1000 from Philips Semiconductors [5]. The implemented controller mainly consists of four modules: a *Message Buffer* module, an Interface Management Logic module (*IML*), a Bit Stream Processing (*BSP*) module and an Error Management Logic (*EML*) module. Furthermore, the CAN simulation model also contains a so-called bus module (CAN Bus) used as an abstraction of the physical bus.

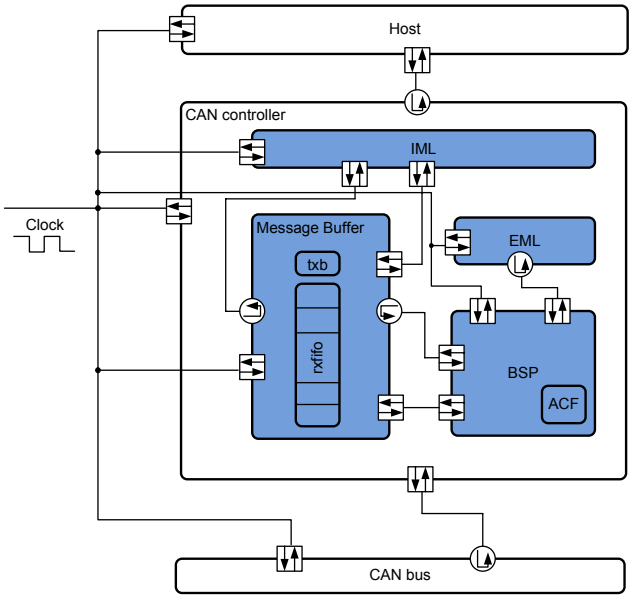

Fig. 6. CAN bus model

- Message buffer is used to store messages (CAN frames) coming from or going to the bus. In the current implementation, it consists of two simple buffers for incoming and outgoing frames.

- Interface Management Logic (IML) is a module that transforms data and identifier received from the host into CAN frames. Following, the built frame is stored into the message buffer and is ready for transfer.
- Bit Stream Processing (BSP) is responsible for the reception (de-serialization) and transmission (serialization) of the CAN messages. It performs aspects of the CAN protocol like bitwise arbitration, (de-)stuffing, error detection (bit errors, form errors and acknowledge errors). Furthermore, the BSP performs acceptance filtering (ACF) and CRC checking.
- Error Management Logic (EML) is a module that implements fault confinements rules according the CAN specification described in [4]. This module consists of several counters to keep track of reception and transmission errors. The counters are updated depending of the kind of error that occurred (Bit Error, Stuff Error, Form Error, CRC Error, Acknowledgement Error).
- The CAN bus module is needed for simulation purposes, it is a SystemC channel, that was implemented to abstract the physical bus.

The implemented CAN controller is an abstract high level CAN model. Therefore, not all the features of the specification are supported. In the current implementation, there is no support of extended CAN frames. Only standard CAN frames are used for network communication. Figure 7 compares a

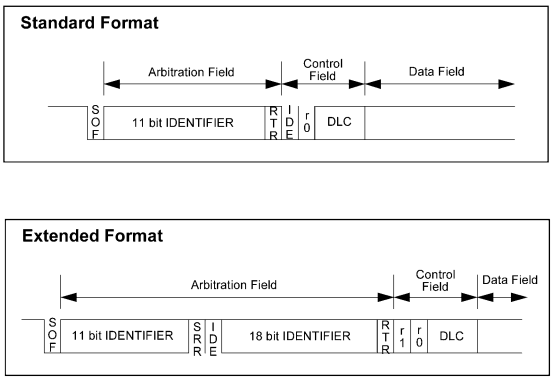

Fig. 7. Structure of a CAN message

standard and an extended CAN message format. As it can be seen, the main difference lies in the size of the arbitration and the control field. Furthermore, Bit Timing Logic (BTL) which is a module that handles bit timing in CAN systems is not implemented. Instead, the assumption is made that all controllers in the network are triggered by the same clock. Therefore, no synchronization is needed. For further information of the module Bit Timing Logic see [4]. Due to the absence of the BTL module, no overload frames are sent by the BSP.

IV. CAN Testbench

The general architecture of the implemented CAN is given in Figure 8. It comprises six components:

- Stimulus generator
- Transfer function
- Driver
- Design under test
- Coverage monitor
- Logging and Acceptance evaluation

The stimulus generator of the testbench is used for the input stimuli generation for the CAN controller (DUT). The transfer function module simulates the input and generates the expected output using a reference model of the CAN controller. The driver (adaptation) component converts the data received from the stimuli generator and drives it to the DUT. Another important component of the testbench is the coverage monitor, as it collects various coverage information based on the output of the stimulus generator and the output of the DUT for later scoreboarding. Note, that the coverage monitor is part of the testbench and autonomous for the functional coverage collection. The logging and acceptance module is needed for data logging and acceptance evaluation based on a scoreboard.

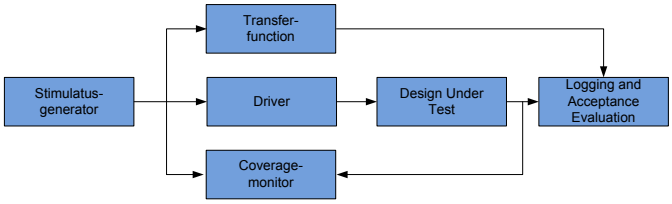

Fig. 8. testbench model

A. Stimulus generation

The stimulus generator creates specific or completely randomized messages for the Design Under Test based on the parameterization. It offers the possibility to generate both standard and extended CAN frames along CAN Specification 2.0 (see [4]. The choice is made based on a value that indicates with what percentage standard CAN frames will be generated throughout the simulation process. 100% means that only standard CAN frames shall be generated. This value can be set by a specific function within the testbench. Furthermore, it is possible to set a fixed ID for the CAN message that is generated. Here, the use of the stimulus generator is inspired by the SystemC Verification Library (SCV). Its application is as follows: After its instantiation it can be parameterized using appropriate function calls. New CAN frame objects are created by the function call *next()* and can be directly referenced in the generator. Furthermore, CAN frames which do not need stuff bits can also be generated by using the *next_no_stuff()* function call. Stuff bits are special bits needed for synchronization purposes during data transmission on a CAN bus. After five consecutive bits of the same value an inverted bit has to be inserted. For the generation of Stuff-bit-free CAN frames, the fields Arbitration, Control and Data are created without stuff bits. Thereafter, the CRC check is simulated to verify that no stuff bits are really expected. If this is the case, the message is stored in the generator and

can be used within test cases to investigate the behavior of the system for this specific case. The generated CAN frames are then passed to the reference model for simulation purpose and, after treatment of data by the adaptation module (Driver), to the actual model for data transmission.

B. Transfer function

The transfer function module receives all CAN frames generated by the stimulus generator as input. These are then taken to compute the expected output of the DUT in order to compare it with the actual output produced by the DUT later within the LA (Logging and Acceptance evaluation) module. Another task of the transfer function is to compute the transmission durations of the frames in clock ticks and stores them in the scoreboard of the LA module. Hereby, the length of the frame including the stuff bits that are inserted during transmission and the arbitration on the CAN bus are considered. The computation of the frame length is done by the so-called *StuffbitChecker*.

The StuffBitChecker is part of the transfer function module. It automatically checks the number of stuff bits of each frame generated by the stimulus generator and stores it with its position. For this, it iterates through the relevant fields of the frame (SOF, ARB, CTRL, DATA, CRC). Furthermore, CRC-Check is simulated for the computation of Stuff bits present in the CRC field. Afterward the results of the computation can be accessed by the LA Module or by the Coverage monitor.

All frames present in the system are stored in a vector and processed according to their respective IDs. This guarantees that only those messages compete as in the arbitration phase of the CAN network.

C. Driver

Since the message objects produced by the stimulus generator are standard data types (`int`, `char`), and because of the fact that the CAN model uses `sc_logic` vectors to store the individual message elements, the driver module makes use of a message converter to convert the generated values into CAN-frames before driving them to the DUT.

D. Coverage monitor

The coverage monitor is used in the testbench model to gather coverage information for message objects produced by the generator and for messages received from the DUT. For this, we define a functional coverage metric with help of our functional coverage library and make use of it in several test scenarios. The coverage metric monitors the desired values and performs a functional coverage analysis. An example is to cover the IDs of the generated CAN frames of the stimulus generator. A termination condition can be set based on the information from the coverage analysis like for instance when a specific number of IDs within a certain interval have been generated.

```
received transfer function message:

data         -->  01011110

arbitration -- > 548(1000100100)

generation_time   --> 0 s

start_time        -- > -
```

Fig. 9. transfer function

```
received controller message:

data         -- > 01011110

arbitration -- > 548(1000100100)

generation_time   -- > -

start_time        --> 5
```

Fig. 10. logging and acceptance evaluation

E. Logging and Acceptance evaluation

The LA module in the testbench is used for system analysis and for logging of generated and received CAN frames. These are the messages produced by the generator and the received by the receiving controller within the DUT are logged. The second task of the LA module is the comparison and verification of the output provided by the Transfer function module with the actual output of the DUT. Figures 9 and 10 depict such a log-file generated by the LA module. As shown in the figure, only the dynamic part (i.e., parts that were automatically generated) of the messages are logged with some additional timing information.

Figure 9 shows the message produced by the stimulus generator, and simulated by the transfer function. This file contains the generation time, the calculated arrival time in clock ticks and the ID controller sending of this message. Figure 10 shows the message after its transmission over the CAN bus. It contains the time stamp at which the controller started the message transfer and the time stamp at which the message was received. These log files can be used for manual review of DUT behavior. Furthermore, the LA module monitors the bus traffic in order to rebuild the transmitted CAN-frames. Hereby, the main focus is to test and verify the behavior of the BSP module by investigating aspects such as the *recognition of stuff bits* and correct behavior during the *arbitration* phase. These are stored in a map with the appropriate CAN frame along with the position of stuff bits for each message. The stuff bits identified by the BSP module are compared with those calculated by the transfer function and taken into account during the acceptance evaluation. The frames calculated by the transfer function and the frames transmitted on the CAN bus are stored in a scoreboard and can later be used to conduct the acceptance evaluation. In order to check the reliability of the design under the conditions provided by the test cases, it is possible to alter various metrics such as the percentage of error free message receptions or the

maximum latency.

V. Scenarios

For verification of the CAN bus model and usability study of the SystemC coverage library, we have defined and implemented several test scenarios. First the relevant signal valuations of the DUT have to be identified such as ID fields, CRC and Stuff Bits in order to cover the associated expressions within the simulation. Moreover, with help of the functional coverage library the DUT control path logic can be efficiently covered using transition bins. In the following subsection, we first give a briefly overview about these scenarios and how we made use of the functional coverage features of our developed library. There, we also summarize Scenarios V-A to V-D, as scenario V-E is a superset of all previously iterative developed scenarios, applying most coverage library features.

A. Sender/Receiver Scenario

This scenario verifies the CAN data packet transmission via the CAN bus from a single CAN controller to two CAN controllers. The controller which intends to send data starts to send a data packet with a certain CAN message ID onto the CAN bus. Two other controllers are configured in a way to accept this CAN message ID. In the test run, the IDs are randomized and the SystemC functional coverage library is used to determine the coverage of the ID variable. Though this use case quite simple, it allows the efficient combination of random constrained stimuli and functional coverage analysis. The developer has easy access to information like how much of the overall variable values have been assigned during the test run.

B. Bus Arbitration Scenario

In this scenario, three CAN controllers start to drive a CAN packet onto the CAN bus model at the same time. One controller sends the message for the receiving CAN controller, the other two send data with different CAN IDs. So despite there is not collision in this test, it verifies the wiring of the CAN bus model and the bus arbitration. Again, the SystemC coverage library is used to cover the CAN ID and to emphasize verification efforts to corner cases. Additionally, the control flow logic of the controller is covered via transition bins.

C. Priorities & Arbitration Scenario

This scenario combines V-A and V-B to verify that messages with lower IDs, and a higher priority, win the bus arbitration. Here, at least two sender and receivers are created. In the fourth controller scenario, the first sender controller sends a CAN message to the other controllers. The second sender also sends a message with a higher ID. So this model verifies the bus arbitration in a multi-sender and receiver CAN network. The coverage library is again used to cover simulated IDs. Illegal bins have been specified to capture disallowed variable assignments as they should not occur during the simulation.

D. Messages Coverage 1

Scenario V-D is an extensive test case where a large set of controllers is applied. In fact, for every possible binary combination of the ID field (which is used for arbitration in the CAN bus standard) a CAN controller is configured and instantiated. This leads to 2032 CAN Controllers each sending messages with the same ID as their Controller ID. Additionally, all controllers act as receivers once they sent their messages or in case of lost arbitration, accepting messages with arbitrary ID value. This scenario enables extensive coverage collection with help of the SystemC coverage library. During the simulation, the received CAN messages ID are sampled (covered), resulting in one bin for every occured message ID. With help of this collected data, the coverage of the input data can be calculated.

Figure 11 shows the constructor of the test class. The SystemC functional coverage library is instantiated and a database file for the collected coverage information is specified via `set_coverage_db_name()`. For a proper working coverage, the factory is initialized via `init_factory()`. Additionally, the CAN bus itself, the transfer function component and the CAN messages generator are instantiated. These initializing steps of the verification environment are based on SCV Library behaviors [6].

```
SC_MODULE(CAN_Test_4)
{
        SC_CTOR(CAN_Test_4): C4("C4")
        {
                SC_METHOD(receive);
                        dont_initialize();
                        sensitive_neg << C4;
                SC_METHOD(sample);
                        dont_initialize();
                        sensitive_neg << C4;
                fac = SCFC_Factory::init();
                fac->set_coverage_db_name("CAN_Test_4_coverage_db");
                fac->init_factory();
                bus          = new can_bus("bus", controller_to_use);
                bus->clock(C4);
                generator = new TB_Generator("CAN_Test_4_Generator");
                transfer_function = generator->get_transfer_function();
                transfer_function->Clock(C4);
                this->create_controller();
        }
```

Fig. 11. Constructor of test scenario V-D

E. Messages Coverage 2

Figure 12 shows the constructor of Scenario five. First, relevant variables are defined and initialized. The variable `message_counter` is a counter variable and represents how many messages are present in the system. If a new message is generated it is incremented. Once a message is received it is decremented. This indicates how many messages have to be driven to the bus simultaneously in the arbitration phase. Variable `stuff_bit_count_sig` is a `sc_signal` from type `int`. It is used for the coverage computation and represents the stuff bits in a generated CAN message. The boolean `compute_coverage` indicates if the coverage has to be computed within the specific clock cycle or not. It is

```
SC_MODULE(CAN_Test_5)
{
        SC_CTOR(CAN_Test_5): C5("C5")
        {
                srand(time(NULL));        // initialize randomizer
                message_counter = 0;
                stuff_bit_count_sig.write(-1);
                compute_coverage = false;
                fac = SCFC_Factory::init();
                fac->set_coverage_db_name("CAN_Test_5_coverage_db.db");
                fac->init_factory();
                SC_METHOD(generate_and_transmit_new_message);
                        dont_initialize();
                        sensitive << C5;
                SC_METHOD(sample);
                        dont_initialize();
                        sensitive << C5;
                bus          = new can_bus("bus", controller_to_use);
                bus->clock(C5);
                generator = new TB_Generator("CAN_Test_5_Generator");
                transfer_function = generator->get_transfer_function();
                transfer_function->Clock(C5);
                this->create_controller();
        }
...
```

Fig. 12. Constructor of test scenario V-E

```
void sample() {
        if (compute_coverage) {
                SCFC_Covergroup* cgType = fac->get_SCFC_Covergroup_type(this,
                        "Stuff_Bit_Covergroup");
                if (cgType != NULL)
                        cgType->sample();
                else cout << "ERROR: SCFC_Covergroup to sample not found\n";
                compute_coverage = false;
        }
        if (transfer_function->data_ready_sig.read() == true) {
                stuff_bit_count_sig.write(transfer_function->get_stuff_bit_count());
                compute_coverage = true;
                transfer_function->data_ready_sig.write(false);
        }
        for (int i=0; i<controller_to_use; i++) {
                can_controller* ctrl = ((can_controller*)(controller_port_array[i]));
                if (ctrl->rx_data_available_sig.read()) {
                        ctrl->rx_data_available_sig.write(false);
                        checker_map[i]->log_message_from_bus();
                        message_counter--;
                }
        }
        if(!compute_coverage && fac->get_SCFC_Covergroup_type(this,
                        "Stuff_Bit_Covergroup")->get_coverage() >= 60) {
                cout << "stopped simulation at " << sc_time_stamp() << " with stuff-bit-coverage
                        reached " << fac->get_SCFC_Covergroup_type(this,
                        "Stuff_Bit_Covergroup")->get_coverage() << " percent\n";
                finished();
        }
}
```

Fig. 14. `sample()` method of test scenario V-E

always set to `true` once the transfer function has determined the stuff bits of a generated CAN message.

Additionally, two `SC_METHOD` are defined which will be executed every clock cycle. In order to generate meaningful traffic for the CAN bus model, one method is utilized to generate and transmit new messages, as depicted in Figure 13.

```
...
void generate_and_transmit_new_message() {
        while (message_counter < max_number_of_concurrent_messages &&
                                message_counter < controller_to_use) {
                if ( ((rand() % 101) < 10) && !(message_counter == 0) ) return;
                int controller_select = rand() % (controller_to_use);
                while ( ((can_controller*)controller_port_array[controller_select])
                                ->tx_data_rdy_sig.read() == true)
                        controller_select = rand() % (controller_to_use);
                cout << "TB_CAN_Test_5: generate new random message\n";
                generator->set_ctrl_id(controller_select);
                generator->next_no_stuff();
                checker_map[controller_select]->get_message_from_transfer_function();
                controller_port_array[controller_select]->transmit(generator->message);
                message_map.insert( make_pair(controller_select, generator->message) );
                data_signal_map[controller_select]->write(true);
                message_counter++;
}
...
```

Fig. 13. CAN message randomizer of test scenario V-E

Here, the `sample()` method in Figure 14 can be subdivided into four tasks:

1) Start of the coverage analysis once the transfer function calculated the stuff bits of a new CAN message.
2) Check if the transfer function calculated stuff bits of a new CAN message.
3) Check if the CAN controller received a CAN message when delivering this message to the Logging and Acceptance evaluation model.
4) Stop in case of sufficient coverage (stop criterion).

The verification environment is depicted in Figure 15. First, a covergroup for the stuff bit coverage is defined. For every stuff bit class a transition bin is declared. The coverpoint relies on the `stuff_bit_count_sig` signal which is driven by the transfer function and contains the amount of stuff bits generated for the latest generated CAN message. The transitions bin makes use of that value in order to check if they are covered. After these steps, the coverage library is instantiated, configured by setting database filename, and initialized.

```
void generate_verification_environment() {
        fac->new_SCFC_Covergroup(this, "Stuff_Bit_Covergroup", "Stuff_Bit_Covergroup_inst",
                this);
        fac->new_SCFC_Coverpoint(this, "Stuff_Bit_Coverpoint", &stuff_bit_coun:_sig);
                fac->new_transition_SCFC_Bins(this, "stuffBitBins0",  stuffBitMethod0);
                :
                fac->new_transition_SCFC_Bins(this, "stuffBitBins18", stuffBitMethod18);
}
static int stuffBitCheck(sc_module* test5, int threshold) {
        CAN_Test_5* test = (CAN_Test_5*)test5;
        if (test->stuff_bit_count_sig.read() == threshold) {
                cout << "Threshold: " << threshold << " -> covered\n";
                return 1;
        }
        return 0;
}
static int stuffBitMethod0(sc_module* test5) {
        if (stuffBitCheck(test5, 0))
                return 1;
        return 0;
}
```

Fig. 15. Verification Environment (transition bins) in test scenario V-E

VI. Related Work

In the area of CAN bus testing, [7] validated a CAN IP and analyzed several aspects of verification methodologies to validate designs in the context of automotive systems development. However, the authors apply manual directed test pattern design and no constrained random stimuli generation and no functional coverage in the sense of functional verification and the SystemVerilog standard. Our studies have shown that for the identification of corner cases, constrained random techniques are much ahead of classical techniques. Moreover, it is not possible to define allowed sequences of variable transitions (e.g., via transition bins) or forbidden cases (e.g., via illegal

bins). In the area of verification libraries, the Open Verification Library (OVL) [8] is maintained by Accellera and provides checkers that may work as assertion, assumption or coverage point checkers. The most recent versions support SystemVerilog, Verilog and VHDL. Unfortunately, there is currently no support for SystemC. Additionally, except SystemVerilog the supported languages are working on RTL level, impeding verification on higher levels of abstraction on more abstract data types. In the area of functional coverage implementations with SystemC, [9] introduces a functional coverage prototype. Unfortunately, just a list of functions without any details is available. In contrast, our library is open with all details as open source. In [10] the authors introduce a verification framework based on the SystemC Verification Library (SCV) providing a coverage monitor library for functional coverage modeling. The implemented basic coverage operators range from logical `OR`, `equal`, `non-equal`, `greater`, etc. and are not as powerful as the IEEE 1800 SystemVerilog coverage features. In [11] the authors propose a coverage-driven verification methodology approach that uses the their own `bve_cover` class. This class has also been referenced in [12] and [13]. It is reported that the approach allows definition of (illegal) buckets (similiar to bins) and cross-coverage. Unfortunately, the authors do not given more details. In [14] a coverage driven testing policy is proposed whereas Property Specification Language (PSL) expressions which are converted to C++ are used to gather and inspect function coverage information. In [15] the author uses the callback facility of the SystemC SCV library to achieve functional equivalent of SystemVerilog value and transition coverpoints. Unfortunately, this approach does not include advanced features like cross coverage, illegal bins or default declaration.

VII. Conclusion & Outlook

In this article, we introduced a SystemC functional coverage library and testbench and applied it to a SystemC model of a Controller Area Network (CAN). The SystemC implementation of the coverage language features of IEEE 1800-2005 SystemVerilog, such as `covergroup`, allows functional coverage to be efficiently collected and evaluated within SystemC simulation runs. In advanced testbenches this coverage data may be efficiently used to steer the stimuli generators or the constraint solvers of constrained random variables into the right direction, targeting closure of the coverage gap. With our library, we could efficiently analyze the testbench for different configurations of CAN controllers. We defined several test scenarios, each utilizing the different features of the functional coverage library. With help of our coverage collection uncovered corner cases of the CAN testbench and the DUT could be identified which were not addressed by the initial verification environment. By improving the testbench we could successfully identify various design flaws and bugs of the CAN bus model. As such, the functional coverage collection helped to emphasize verification effort into the right directions during the development. For example, control path logic can be efficiently covered using `transition bins`, whereas forbidden cases may be modeled as `illegal bins`. Though SystemC CAN model is implemented at RTL level, test cases could be driven and managed by a high level testbench.

Future work, is the investigation of the impact of the Open Verification Methodology (OVM) to SystemC in combination with adding coverage support for arbitrary data types such as events on transaction level. Additionally, a closer database coupling to, for example, Accellera UCIS could enhance coverage results reuse and management.

Acknowledgements

This work was partly funded by the DFG Collaborative Research Centre 614 and by the German Ministry of Education and Research (BMBF) through the BMBF projects SANITAS (01M3088) and VERDE (01IS09012). We greatly appreciate the cooperation with the project partners.

References

[1] J. Bergeron, "Writing testbenches: Functional verification of HDL models." Kluwer Academic Publishers, 2003.

[2] "IEEE Standard for System Verilog-unified Hardware design, specification, and verification language," *IEEE STD 1800-2009*, 2009.

[3] Open SystemC Initiative, "IEEE Standard SystemC language reference manual," *IEEE Std 1666-2005*, 2006.

[4] Robert Bosch GmbH, "CAN Specification version 2.0", 1991.

[5] Philips Semiconductors, "SJA1000 stand-alone CAN controller," Data sheet, 2000.

[6] *SystemC Verification Library v1.0p2*, Open SystemC Initiative, 2006. [Online]. Available: http://www.systemc.org/downloads/standards/

[7] J. S. Antonio Souza and P. Domingues, "Functional verification of a CAN data layer implementation: a case study," *Brazil Semiconductor Technology Center (BSTC)*, 2003.

[8] Accellera Organization Inc., Open verification library (OVL). (2009, May) [Online]. Available: http://www.accellera.org/activities/ovl/

[9] R. Siegmund, U. Hensel, A. Herrholz, and I. Volt. (2004) Functional coverage prototype for SystemC-based verification of chipset designs. AMD Dresden Design Center.

[10] S. Park and S.-I. Chae, "A C/C++-based functional verification framework using the SystemC verification library," *Rapid System Prototyping, IEEE International Workshop on*, vol. 0, pp. 237–239, 2005.

[11] K. R. G. da Silva, E. U. K. Melcher, G. Araujo, and V. A. Pimenta, "An automatic testbench generation tool for a SystemC functional verification methodology," in *SBCCI '04: Proceedings of the 17th symposium on Integrated circuits and system design*. New York, NY, USA: ACM, 2004, pp. 66–70.

[12] G. S. Silveira, K. R. G. da Silva, and E. U. K. Melcher, "Functional verification of an mpeg-4 decoder design using a random constrained movie generator," in *SBCCI '07: Proceedings of the 20th annual conference on Integrated circuits and systems design*. New York, NY, USA: ACM, 2007, pp. 360–364.

[13] C. L. Rodrigues, K. R. G. da Silva, and H. N. Cunha, "Improving functional verification of embedded systems using hierarchical composition and set theory," in *SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing*. New York, NY, USA: ACM, 2009, pp. 1632–1636.

[14] Y. Lahbib, O. Missaoui, M. Hechkel, D. Lahbib, B. Mohamed-Yosri, and R. Tourki, "Verification flow optimization using an automatic coverage driven testing policy," in *Design and Test of Integrated Systems in Nanoscale Technology, 2006. DTIS 2006. International Conference on*, sept. 2006, pp. 94 –99.

[15] K. Schwartz, "A Technique for Adding Functional Coverage to SystemC," in *DVCON 2007*. Willamette HDL, Inc., 2007.