Dynamic Fault Injection Library Approach for SystemC AMS

Thomas Markwirth, Fraunhofer IIS - Design Automation Division - EAS, Dresden, Germany
(*Thomas.Markwirth@eas.iis.fraunhofer.de*)

Paul Ehrlich, COSEDA Technologies GmbH, Dresden, Germany
(Paul.Ehrlich@coseda-tech.com)

Dominik Matter, Hella Aglaia Mobile Vision GmbH, Berlin, Germany
(*Dominik.Matter@hella.com*)

***Abstract*—This paper introduces a fault injection library for SystemC/SystemC AMS which can be used to dynamically integrate failure structures into arbitrary SystemC/SystemC AMS descriptions. The injection is realized at the beginning of a test case run by dynamically reconnecting netlists without changing the DUT model itself. The approach was successfully validated on a model of a battery management system (BMS). Additionally for the same system, the benefits of the proposed fault library are shown and discussed for Hardware-in-the-Loop systems during lab validation.**

Keywords—Verification, Validation, SystemC AMS, Fault injection, HiL, Hardware Acceleration

I. Introduction

There is an increasing complexity of systems consisting of analogue and digital hardware and embedded software, as the physical environment has to be taken into consideration during the verification and validation phase. However, verification is not only required to test the nominal behavior. It is also necessary to check the behavior in case of faulty components. This ensures not only the fulfilment of functional requirements but also the fulfilment of safety requirements. Verification tools and methods have to support the design of functional correct, robust, and safe systems. Especially, the automotive industry expects solutions that are in compliance with the ISO 26262 functional safety standard. The presented approach can be applied in the design process of the related systems.

A simulation approach based on SystemC/SystemC AMS is a good choice to validate the nominal behavior. It guarantees a high simulation speed while maintaining appropriate accuracy. Software development aspects can be included into the design and verification process. The approach also closes the bridge to Hardware-in-the-Loop (HiL) simulation and therefore to the lab validation.

So far, little effort was spent on improvement of the consideration of failure aspects and appropriate methods to model fault behavior. This paper, therefore, describes a new method to inject faulty behavior into SystemC/SystemC AMS descriptions of the nominal behavior without changing the models or netlists. Thus, the main advantage of the solution is that for a wide class of failures it is not necessary to modify the SystemC/SystemC AMS description that is provided for the nominal case. Faulty behavior can be handled on the level of test scenarios. The potential of the approach was investigated related to the specification phase of a battery management system and further tested in a HiL environment during the project IKEBA[*].

State of the art fault injection methods use approaches which typically integrate faulty behavior or failure structures directly into the model, which is used as DUT. For a certain test case the nominal or the faulty behavior is used. This approach has the disadvantage of no clear separation between functional and test description. On the other hand, there is a risk that functional and faulty behavior are not consistent.

* The R&D project IKEBA was partially funded by the German Ministry for Education and Research (BMBF) under the grant *16N12440*. The sole responsibility for the contents rests with the authors.

The paper is organized as follows: The second section describes the requirements for fault injection mechanism. The third section describes the proposed library exemplarily for the different SystemC/SystemC AMS domains and models of computation (MoC). Then in section four, the library is used together with hybrid HiL simulations. It is followed by a case study which is based on battery management system, in section five. Finally, section six and seven conclude the paper and give an outlook on future work.

II. Fault Injection Requirements

In order to enable usability for a wide range of different models/systems, a generic modelling approach has to be used. This provides the possibility to use the fault injection library for different model domains and MoCs. The fault injection approach should guarantee a clear and absolute separation between functional model (DUT) and test descriptions. This means, that the separation has to be realized for both aspects related to data management and runtime integration. Additionally, this gives the possibility to develop completely separate descriptions of the functional model (DUT), test environment and the fault description. Another key point should be a simple usability for own use cases. Therefore, predefined and unified interfaces have to be used. Last but not least, a central location for a failure configuration related to a certain test case has to exist.

III. Proposed Library

The introduced fault injection library uses a hierarchical approach. This means, the fault injection can be described at different abstraction levels. There are injection structures at the lowest level, e.g., in order to disconnect signals/ports of a DUT netlist and fault inject models. This reconnection is dynamically done before the simulation of the related test case is started and without changing the DUT model itself.

A. *SystemC backgrounds:*

The execution of a SystemC application is divided in two phases, elaboration and simulation [1]. During elaboration the module hierarchy which should be simulated in the second phase is created. In order to gain control of the different phases, the hardware description language SystemC provides callback functions. These callback functions are called by the simulator kernel and can be used to influence the execution. E.g., the callback function *before_end _of_elaboration* is called after creation of the module hierarchy, but before the port binding is completed. Therefore, the module hierarchy can still be extended or manipulated. This language feature is used in order to instantiate fault structures which will be injected into the DUT hierarchy.

B. *Reconnecting:*

Based on the language functionality which was described in the last section, the design environment COSIDE[1] already provides a library function *sc_reconnect_port*. The function takes references to the target port, the new signal and additionally a signal reference to return the previously connected signal. If this function is called during the *before_end_of_elaboration* callback, the references can be used in order to insert a switchable driver.

In principle, this approach is usable for all MoCs in a SystemC/ SystemC AMS system with different kinds of low level fault injection structure models. A low level structure model represents the lowest abstraction level of fault injection models in the library and could be a simple multiplexer, for example for non-conservative signals/ports. Afterwards, such a multiplexer enables the switching between the original and a new driver, for instance a new source (e.g. for stuck-at value) or another signal from inside or outside the DUT (e.g. for crosstalk).

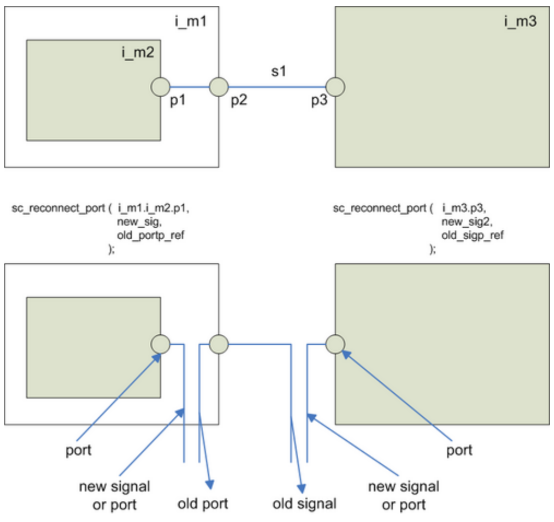

```
sc_reconnect_port (  i_m1.i_m2.p1,
                     new_sig,
                     old_portp_ref
                   );
```
```
sc_reconnect_port (  i_m3.p3,
                     new_sig2,
                     old_sigp_ref
                   );
```

Figure 1: Reconnecting approach, source: COSEDA Technologies GmbH

[1] – COSIDE is a design environment provided by the COSEDA Technologies GmbH

For linear networks, as one of the SystemC AMS MoCs, a switchable resistor is instantiated, instead. By using the references to the original connected node and the terminal, the resistance is connected between both. Afterwards, the resistance value is definable by a separate port.

Depending on the used fault model, the instantiation of an additional switchable resistance and a voltage source can be necessary in order to connect an alternative source to the target terminal. Additionally, it has to be considered that the MoC and the signal type of the source and the target have to be identical. Figure 2 shows the low level structures used for different MoCs.

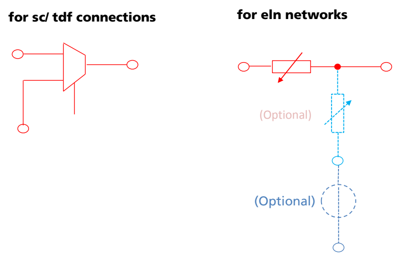
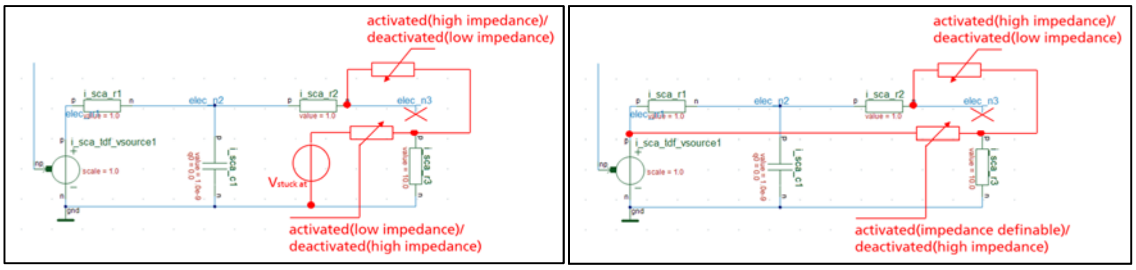


Figure 2: low level failure structures for different domains

C. Fault models:

The fault injection library includes a number of classical fault models, which are based on the low level injection structures, introduced in the previous sections. The fault models represent the next higher abstraction level in the library. Actually, classes for the following fault models are available: stuck-at (value or signal), crosstalk, bridging, open/ short, delay and glitch. It is planned to extend this enumeration. The figure below shows examples.



activated(high impedance)/ deactivated(low impedance)
activated(low impedance)/ deactivated(high impedance)
activated(high impedance)/ deactivated(low impedance)
activated(impedance definable)/ deactivated(high impedance)
i_sca_r1
i_sca_r2
i_sca_r3
i_sca_c1
i_sca_tdf_vsource1
elec_p2
elec_n3
V_stuck at

Figure 4: stuck at (value) and crosstalk examples for linear networks

The fault models are realized as template classes. This means, that they are usable for several signal types. The data type is the template argument and has to be defined during the instantiation. Contrary to the data type the MoC type is recognized automatically by identifying the signal/ port derivations. Therefore, the right low level structure is chosen automatically. During the instantiation of a fault model, an arguments set consisting of at least a string representing the name of the target port and its hierarchy are necessary. It therefore represents the location where the fault should be injected. Optional is a second string of an object, which should be the new driver. This new driver can also be realized by instantiating an additional module prior to the instantiation of the related fault model.

The constructor of the fault model will call a function *get_object* by using the object name strings above. If these objects could be found successfully, the references of the expected objects are given back by this function. The *get_object* function with extended pattern matching is also a part of the COSIDE design environment and allows to find an object in the module hierarchy by searching its object name during the runtime.

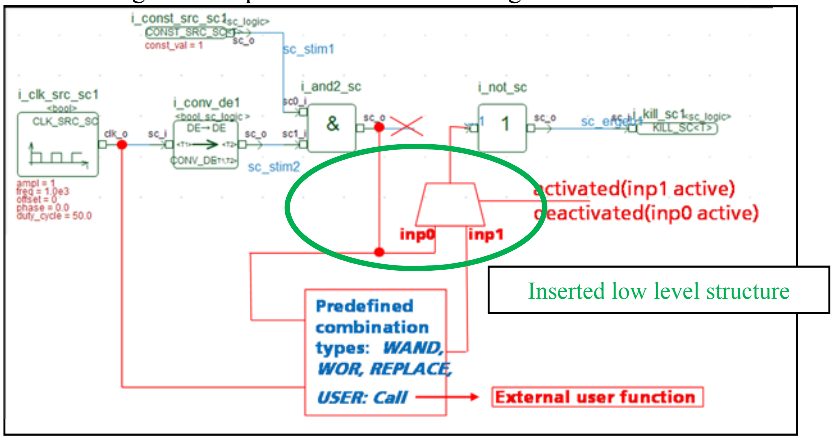


Figure 5: logical combination types for crosstalk in non-conservative networks

The fault model crosstalk allows, especially for digital or non-conservative signals/ ports, to define the kind of logical combination of original and new driver. Actually available are the logical combinations *WAND, WOR, ADD, SUB, REPLACE* and *USER*. The last one enables the possibility to call a user function, which realizes the combination.

D. Scenarios:

The highest abstraction level in the library is represented by the scenario. A scenario instantiates one or more fault models and allows to configure it/them. The library includes a scenario template class which gives the possibility to use the predefined fault models. The template argument of the scenario is the data type of the target port. Additionally, the constructor expects some arguments. The first constructor argument includes a vector of string pairs. As a scenario can handle more than one fault model instances, these pairs are the target and the source ports for each of the fault models. A second argument is the name of the desired fault model. The type of this argument is an enumeration, which includes all predefined fault models. Additional constructor arguments could be necessary, for example the logical combination in the crosstalk fault model. The fault scenario template includes the possibility to activate fault models permanently. Alternatively, a sequence for periodical activation/ deactivation of fault models can be configured. Additionally, the sweep of e.g. the voltage for the stuck-at fault models can be configured, too.

E. Applying the library and the stimulus model:

Creating a separate SystemC stimulus module is a proven approach to apply the fault injection library for specific investigations. The instantiation and configuration of one or more fault injection scenarios is the only task of such a stimulus model. Additionally, the stimulus module provides activation/ deactivation and, optional, tracing functionalities related to the fault. In most cases, the fault injection stimulus module is related to a specific test case. Therefore, it is suggested to create a special fault injection stimulus module for each test case which includes fault injection. Figure 7 shows an example of a fault injection stimulus model, which includes crosstalk and stuck-at scenarios

```
stimuli_fault_injection_spi(sc_core::sc_module_name nm, params pa = params() ) : p(pa)
{
  ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
  /// Create 1. Scenario: crosstalk from global clock signal to the chip select and mosi signals oft he SPI interface
  //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

  // define source and target ports
  port.push_back("*i_spi_slave1.mosi");
  port_q.push_back("*i_clk_src_sc1.clk_o");
  port.push_back("*i_spi_slave1.csq_in");
  port_q.push_back("*i_clk_src_sc1.clk_o");

  // ..and instantiate the scenario
  cross_f1 = new fault_scenario_template<bool >(std::make_pair(port, port_b), fault_injection_base::FAULT_CROSSTALK, fault_injection_base::REPLACE);

  ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
  /// Create 2. Scenario: SPI-clock: Stuck-at “true“
  ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
  port_c.push_back("*i_spi_slave1.sclk_in");
  port_d.push_back("");
  stuck_at_err1         = new fault_scenario_template<bool >(std::make_pair(port_c, port_d), fault_injection_base::FAULT_STUCK_AT);
  stuck_at_err1->fehler_mod[0]->set_voltage(double(1));     //Haftfehler für boolean Port auf Wert true

  fault_duration        = sc_core::sc_time(3.0, SC_US);

  stat_mode_val         = STAT_STATISTICAL_SUB;
  SC_THREAD(sequence_of_all_faults);

  enable_fault          = false;
}
```

Figure 7: example of a fault injection stimulus module, which creates different scenarios

F. Activation during a test case:

The simplest way to integrate fault injection in a special test case is to instantiate the fault injection stimulus module, which was described in the last section. The fault injection can be activated by calling the activation

```
                                   : :
//////////////////////////////////////////////////////////////////
void ikeba_toplevel_2_stim_bmic_bh_error::tb_measure_inclusion()
{

  ////////////////////////////////////////////////////////////////
  // instantiation of a fault injection stimulus module
  ///////////////////////////////////////////////////////////////
  fault_stim = new stimuli_fault_injection_ikeba("fault_stim");
                                   : :
```

```
                                   : :
//////////////////////////////////////////////////////////////////////
void ikeba_toplevel_2_stim_bmic_bh_error::stimulus_sequence()
{
  //dynamic control sequence
  wait(1.0, SC_US);
  dut->resn.write(false);
  wait(2.0, SC_US);
  dut->resn.write(true);
  wait(100.0, SC_US);

  wait(0.1, SC_SEC);
  fault_stim->activate_fault_injection(true); //activiation of fault injection
                                   : :
```

Figure 8: instantiation and activation of a stimulus fault model inside a generic test case file

function of the fault injection stimulus model depending on either the simulation time or a trigger event. The COSIDE environment enables the configuration of test cases by using a generic approach and provides the automated generation of template files, so-called generic test case files. Typically, the activation/ deactivation of the fault injection is described in the stimulus_sequence part of such a generic test case file.

G. *Inclusion of statistical aspects:*

The occurrence of faults in real systems can depend on diverse conditions, which are very often not predictable. In order to reproduce similar behavior, it is state of the art to use statistical methods.

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
/// Create 1. Scenario: Crosstalk from a constant source; occurrence time and active failure location are statistical
/// varied and activated/ deactivated in a sequence loop
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
scenario1 = new fault_scenario_template<bool >(std::make_pair(port, port_b), fault_injection_base::FAULT_CROSSTALK, fault_injection_base::REPLACE);
scenario1->stat_failure_time_function           = exponential;
scenario1->location_function                    = fault_location;
scenario1->set_mean_fault_occurrence(           sc_core::sc_time(30.0, SC_US) );
scenario1->set_fault_duration(                  sc_core::sc_time(5.0, SC_US) ));
stat_mode_val            = STAT_STATISTICAL_SUB;
```

Figure 9: Example of fault injection by using statistical functions from the statistical libraries

Statistical functionality can be integrated in the actual release of the fault injection library by using a statistical library, which was introduced in [2]. For a certain test case it is possible to dice both, the time when a failure occurs and, if more than one fault model is included as well as the failure location.

H. *Specifics of TLM-transactions:*

The approach which was described in the previous sections is not applicable for TLM-transactions, due to missing signals and ports in a literal sense. Instead, TLM uses so-called sockets in order to transmit the information, which can be understood as using access functions. Therefore, fault injection into TLM transactions has to be realized in a completely different way.

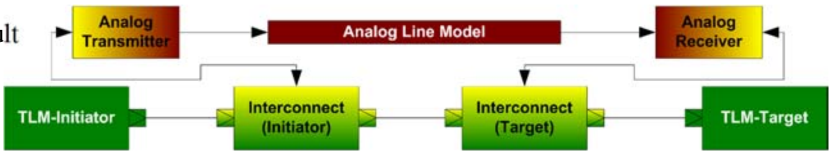


Figure 10: adaptor insertion example, source: [3]

A practicable approach for fault injection into TLM transactions would be the usage of so-called adaptors. Such adaptors represent interconnects which are integrated automatically between TLM initiators and/ or TLM targets. The TLM adaptor approach was already described in [3]. The advantage of these adaptors is their high flexibility, which can be used for fault injections, too. Special manipulation function have to be provided depending on the used transaction, because the transaction content is specific to the actual transaction as well as its manipulation. On the other hand, there is an effort to write and automatically integrate the adaptors into specific TLM models. However, the usage of the tlm_generic_libraries [4] for custom TLM applications extremely simplifies this task. Using adaptors for fault injection into TLM transactions is a reliable approach. An investigation of other TLM approaches was not done and is therefore out of focus for this work.

IV. FAULT INJECTION DURING LAB VALIDATION

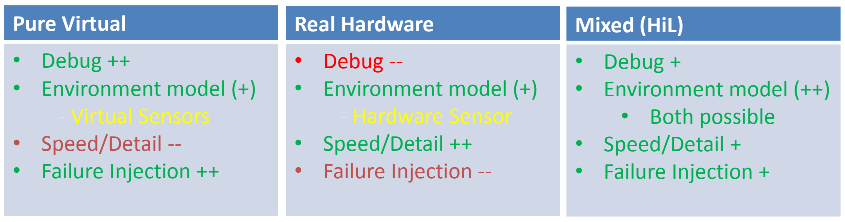
The above introduced fault injection mechanism is not limited to a virtual prototype and can be applied during lab validation or even re-used from the virtual prototype. By lab validation we understand the test of early available parts or prototypes of the system, are insufficient for a complete system test. Therefore, we distinguish between the following three major groups of prototypes:

- Pure Virtual – a virtual prototype completely modeled and executed by a simulator
- Real Hardware – a setup where every part of the prototype is represented by an existing piece of hardware
- Mixed (HiL) – a mix of both, thereby the virtual part is executed on special hardware (HiL Tester)

The requirements for such a mixed environment aim to obtain a HiL-Tester capable of running SystemC AMS for the model part as well as hardware/ software interfaces, which guarantee value and timing accurate connections.

This is discussed in detail in [7]. To compare the different types of prototypes they are evaluated regarding different aspects listed below:

- Debug – Accessibility and traceability of signals for error root cause analysis
- Speed/Detail – Tradeoff between speed vs detail level of the model
- Fault Injection – Possibility to manipulate expected behavior to evaluate system robustness



Pure Virtual	Real Hardware	Mixed (HiL)
• Debug ++	• Debug --	• Debug +
• Environment model (+)	• Environment model (+)	• Environment model (++)
- Virtual Sensors	- Hardware Sensor	• Both possible
• Speed/Detail --	• Speed/Detail ++	• Speed/Detail +
• Failure Injection ++	• Failure Injection --	• Failure Injection +

Figure 11 Comparison of different tpyes of prototypes

The result of the comparison is shown in Figure 11. The table therefore shows that virtual prototypes are most easy to debug and always have a tradeoff between modelling detail and the resulting simulation speed. Especially when it comes to fault injection the virtual prototypes are superior and can use the fault injection library discussed in this paper. The hardware prototypes on the other hand maintain by nature every piece of detail, while maintaining full speed, but have a very limited accessibility to signals and internals for debugging. Their fault injection is limited to locations applicable, the resulting consequences and the deterministic repeatability. Locations are for example are accessible terminals which can be cut or short circuited. Consequences could be permanently broken hardware or hazard to the environment. The HiL prototype contains real hardware along with the model running on the HiL-Tester. Thereby the discussed advantages for software and hardware parts remain but the partitioning can be varied as needed up to a full hardware model of the DUT. At this point only the test bench would remain in SystemC AMS on the HiL-Tester.

To highlight the benefits of the mixed prototype, the partitioning of the BMS of the case study is discussed. One scenario for the prototype would be the firmware development. Therefore the control unit with its application processor is used in real hardware to realistically run the firmware, while the environment remains in software to easily inject faults, for which the firmware should be resistant. Especially the battery cells should remain in software as they could cause a fire or get damaged if short circuited, which would lead to replacements during firmware design or regression runs. This also benefits the test time because batteries have to be charged and discharged to obtain a certain state of charge, which is not even deterministically reproducible because complex physicochemical reactions are involved. The HiL setup also provides a realistic and safe environment for early versions of the firmware, since faults can also occur unintended during the firmware development. Therefore, the introduced fault injection library also aids the design flow during lab validation and significantly improves test coverage at this stage.

V. Case Study Battery management

An important key point while developing, especially automotive components and systems is to guarantee their functional safety. An applicability of the fault injection according to this topic for real systems was investigated in the government funded project IKEBA. The basis for the work was a system model of a battery management system which was created in the project. This complex system model includes beside digital and analog hardware components also non-electrical effects and real software applications of the target system. The battery model was created by using characterization results of true battery cells made by the KIT in Karlsruhe.

One aim of the project was to investigate the ability of the management software application of the project partner Hella to recognize failures fast enough and, if necessary, to react on them, e.g. after single cell damages or overheating. During the project it was decided that a dynamical fault injection provides the best approach to fulfill this task.

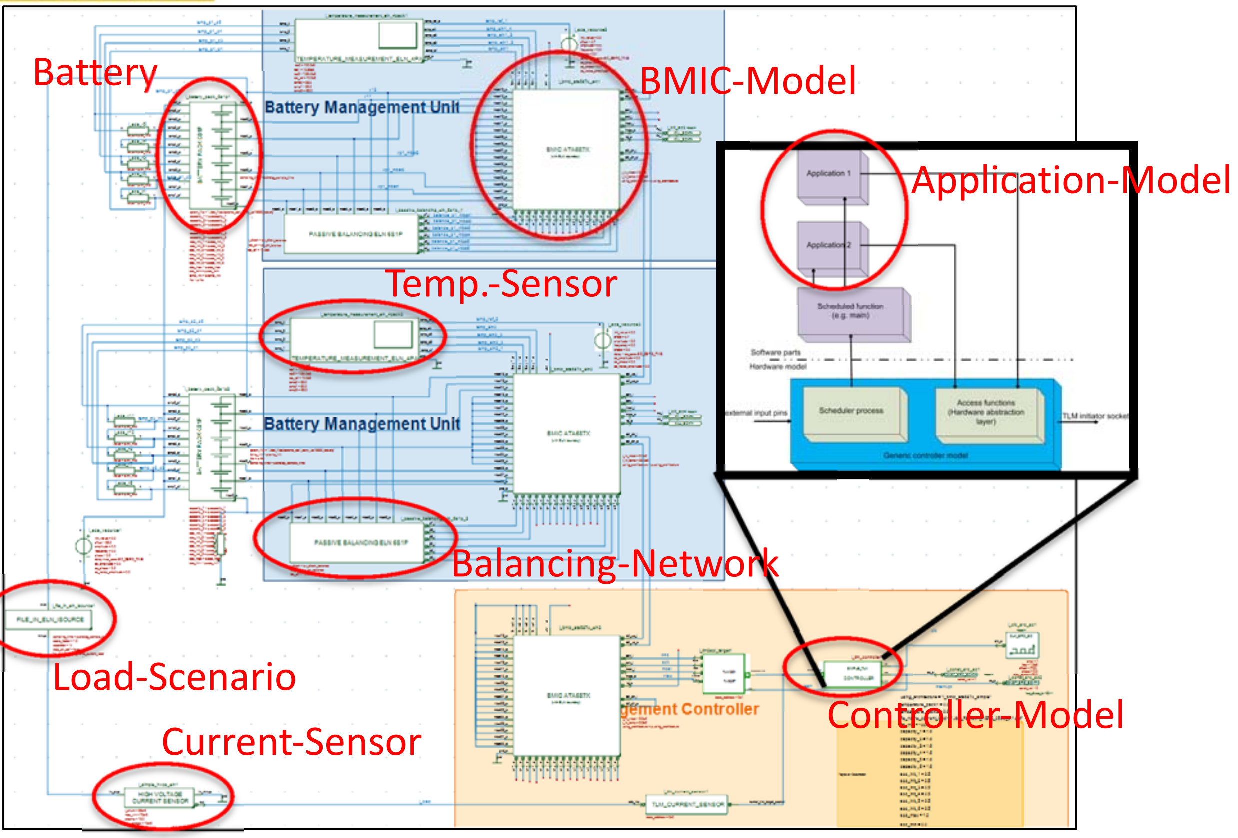


Figure 12: Toplevel of the battery management system model

Therefore, the fault injection approach was exemplary used for the following tasks:

- Investigation of the influence while battery cells run out of specified voltage or temperature ranges
- Investigations of communication failures in the SPI interface
- Provoke dynamical load balancing
- Generate special stimuli, e.g. for the internal ADC of the BMIC

Exemplary, the following figure shows the application of the library to inject faults into battery pack models. This is done by inserting switchable resistances in series to the terminal connections of a battery cell. Afterwards, the related cell can be deactivated by setting the resistance value to high impedance. An additional voltage source in parallel to the battery cell is activated in this case. Finally, the voltage on the cell terminals can be defined by the test case description. This description can include several scenarios like over/ under voltage, short spikes, ramps/ sweeps or many other scenarios. Figure 14 shows a comparison of simulation results for the nominal case and a faulty cell voltage. The test case which includes fault injection is used in order to check the ability and reaction time of the application for detecting battery failures. Therefore, this simple example shows the high flexibility of the approach.

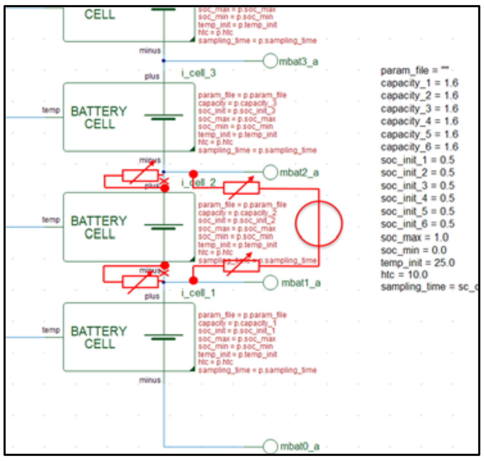


Figure 13: fault injection into battery packs

Overall, the system model assisted in the implementation and test phase of the BMS-software. The dynamic fault injection in particular allowed for accelerated implementation of the fault reaction routines by provinding a virtual testbench for cell boundary conditions (like described above). Although the functional safety features of the IKEBA-software application are far from extensive, the experience gained during the project provides confidence that full functional safety applications would benefit from the dynamic fault injection approach. The virtual system model was faster to set up than an equivalent HiL-Tester while being of lower cost and although it cannot replace HiL-tests completely, it can certainly help to reduce HiL workload.

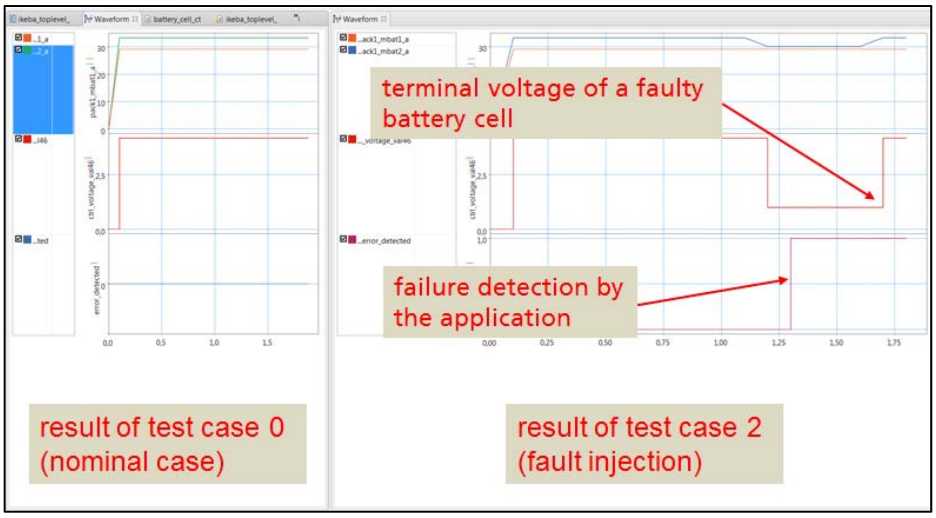

terminal voltage of a faulty battery cell
failure detection by the application
result of test case 0 (nominal case)
result of test case 2 (fault injection)

Figure 14: results for nominal case and fault injection

VI. CONCLUSION

The presented library allows the user to non-intrusively inject faults in ports of different domains in SystemC AMS, which includes normal SystemC ones. Therefore the normal test bench includes an additional SystemC model which dynamically rewires the DUT to include the failure, while also providing control handles for it. The library has been successfully used to inject faults into different parts of a battery management system (BMS) developed during the IKEBA research project.

VII. FUTURE WORK

Although the library has proven successfully to work for different SystemC AMS projects the authors wish to continue the work to improve its applicability. The library should be easily to use and integrate, for example into an existing UVM-SystemC environment [8]. Therefore it can provide additional or extend test sequences, leading to more comprehensive test runs improving the verification and validation coverages. Another focus will be the detection of the injected faults and the traceability of its propagation throughout the DUT.

ACKNOWLEDGMENT

We thank our former Fraunhofer-EAS colleagues Thomas Uhle and Karsten Einwich for prior activities.

REFERENCES

[1] IEEE Computer Society, 1666-2005 IEEE Standard SystemC Language Reference Manual

[2] T.Markwirth, J.Haase, K.Einwich; Statistical Modeling with SystemC-AMS for Automotive Systems; FDL'08; 2008

[3] S.Schulz, J.Becker, T.Uhle, K.Einwich, S.Sonntag; Transmitting TLM transactions over analogue wire models; Design Automation and Test in Europe (Date'10) conference; March 2010

[4] T.Markwirth; Entwicklung einer SystemC Modellbibliothek zur Transaktions-Level-Modellierung (TLM) für den Konzeptionsentwurf von Systemen der Automobilelektronik; Diplomarbeit TU-Dresden; 2011

[5] Accellera Systems Initiative, SystemC AMS 2.0 Standard, http://www.accellera.org/downloads/standards/systemc.

[6] P. Ehrlich, T. Nguyen, T. Vörtler, "UVM-SystemC based hardware in the loop simulations for accelerated Co-Verification", Design and Verification Conference & Exhibition (DVCon Europe), October 2015.

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

[8] M. Barnasconi, F. Pêcheux and T. Vörtler, "Advancing system-level verification using UVM in SystemC", Design and Verification Conference & Exhibition (DVCon), March 2014