2023 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE 10 YEAR ANNIVERSARY

# Efficient Debugging on Virtual Prototype using Reverse Engineering Method

Sandeep Puttappa, Senior Staff Engineer, Infineon Technologies, Bangalore, India
(*sandeep.puttappa@infineon.com*)

Dineshkumar Selvaraj, Lead Principal Engineer, Infineon Technologies, Bangalore, India
(*dineshkumar.selvaraj@infineon.com*)

Ankit Kumar, Associate Engineer, Infineon Technologies, Bangalore, India
(*ankit.kumar@infineon.com*)

*Abstract*— **In automotive industry, the development of software (SW) involves collaborative efforts among Semiconductor suppliers, Third party SW partners, Tier1 suppliers, and OEMs. On other hand, Virtual Prototype, a software model of the hardware implemented using SystemC, has become a preferred choice for SW validation platform due to their early availability, ease of use, prolonged maintenance lifespan, and superior debugging capabilities. With multiple stakeholders participating in SW development, the complexity of the software itself increases. Consequently, there arises a necessity to exchange SW-related information among these stakeholders when an issue is detected during its validation over ECU Virtual Prototype platform. However, given the confidential nature of the software, sharing such information presents a new challenge. This challenge, in turn, complicates the task of reproducing the identified issue on a Virtual Prototype. In this paper, we will discuss on 'Reverse Engineering' method, where events are recorded on log format during ECU level simulation and then same recorded data is used to replicate the original behavior in Standalone reproducer environment consisting of only the IP SystemC model and the reproducer model.**

***Keywords*— *Reverse Engineering, Virtual Prototype, SystemC model, Trace and Debug, Standalone debugging***

## I. INTRODUCTION

In Automotive Industry, the complexity of the software is constantly increasing over time. With Software Defined Vehicle (SDV) becoming popular, the modern cars are having multiple number of advanced System on Chip (SoC)/Microcontroller Unit (MCU) components and a big part of them is operated and differentiated using the software to improve the safety of the vehicle and its performance. On other hand, there exists a deep value chain consisting of Semiconductor suppliers, Third party SW partners, Tier1 suppliers and OEMs. In addition to increasing complexity of software, the Time to Market is becoming a critical requirement due to changes in market dynamics resulted by new entrants into the market. Hence the industry is looking for new methodologies to speed up the development of these complex software components. One of these methodologies is simulation based virtual validation prior to HW availability. In this context, the MCU simulation platform also called as Virtual Prototype (VP) is becoming popular as it mimics the functionality and Hardware - Software interface precisely making it best fit to frontload the validation of SW. Moreover, the usage of MCU VP is continued in post silicon phase by integrating the MCU VP into software Continuous Integration (CI) and Continuous Delivery (CD) framework as VP helps to reduce the cost of HW based test benches. This means, the MCU VP is expected to be maintained for long time (~15 years) very similar to the production software. Lastly the VP being a software model, it provides an enhanced tracing and debugging capabilities enabling faster debugging of the complex software failures. As the MCU VP supports scalability, the MCU VP Prototype is extendable by integrating the models of external components making is as VP at Electronic Control Unit (ECU) level which increases the complexity of VP platform further.

Given that the usage of MCU VP is gaining a significant traction in the industry and its complexity is increasing, it is important to focus on the improvements of quality and debuggability of the VP. One of today's challenges related to debugging MCU VP issues is unavailability of access to the application use cases and software components and ECU level architecture and configuration that result in failure of the MCU VP. Since SW

components are considered a key confidential IP by respective suppliers, it is not easy to share with VP suppliers. As we already discussed, there is deep value chain in software eco-system, it involves increased time and effort to debug complex failure though joint debugging involving all relevant parties. As MCU VP is a complex system on its own, complete reproducer information is necessary to debug and fix the issue in quick time. This brings the challenge to MCU VP developer, on how to reproduce the customer issue in the absence of SW binary or reproducer.

## II. RELATED WORK

Although debugging is a key challenge, there exists very few publications in public domain that focus on problems around effective debugging of SystemC models. Ref [1] briefly mentions about how the trace generated at MCU level can be used to reproduce failing scenario using MCU VP setup. This requires tool specific utilities and does not work in Accellera SystemC environment. Ref [2] describes reusing RTL traces to verify SystemC model at unit level. Hence it focuses on improving the quality of the model and does not address the challenge related to debugging field issues reported by customers. Ref [3] describes the Record and Replay of SW issues in embedded systems which motivated us to apply a similar concept for SystemC model debugging. Ref [4] presents development of a non-intrusive SystemC tracing tool. However, this requires shipping an additional tool to end users. Hence, we propose a method that enables recording end user issue at ECU level and replaying at unit level based on purely tracing capabilities of SystemC models.

## III. PROPOSED SOLUTION

In MCU VP, each functional block (IP) model is commonly implemented using SystemC and TLM 2.0 modelling libraries. These libraries support the design of the functional block models at higher abstraction by using event-based design principles. This design principle is based on optimizing the number of invocations of the SystemC processes and events in the simulation. This helps to meet the required simulation performance and to ensure 100% functional correctness. In such design, the model behaviors are invoked in response to one of the following triggers.

*A. Model Configuration Changes*

*B. Register Transactions*

*C. Interface Modifications*

In SystemC based IP model, the Model Configuration Changes occur only in SystemC Elaboration phase and whereas the Register Accesses and Interface Modifications occur in the subsequent SystemC Simulation phase.

In SystemC model, it is possible to identify the event(s) that initiated the process(es) during the simulation. In addition, the SystemC model also enables monitoring of model behavior by utilizing debug messages. By using these two capabilities, we can effectively capture the various triggers influencing the model's behavior. If these triggers information is captured alongside simulation time, then this data can subsequently be used within a standalone environment to replicate the original behavior.

In this paper, we propose a method:

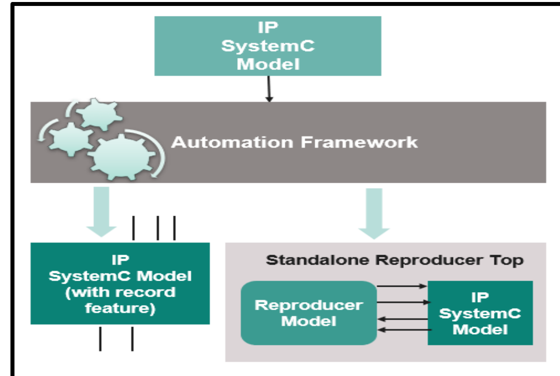
- To implement logging feature in the IP SystemC model for recording all the triggers that occur at model boundaries
- To replicate the model’s behavior accurately in an independent environment using recorded data

## IV. IMPLEMENTATION

The implementation of the 'Reverse Engineering' method requires the fulfilment of the following two prerequisites.

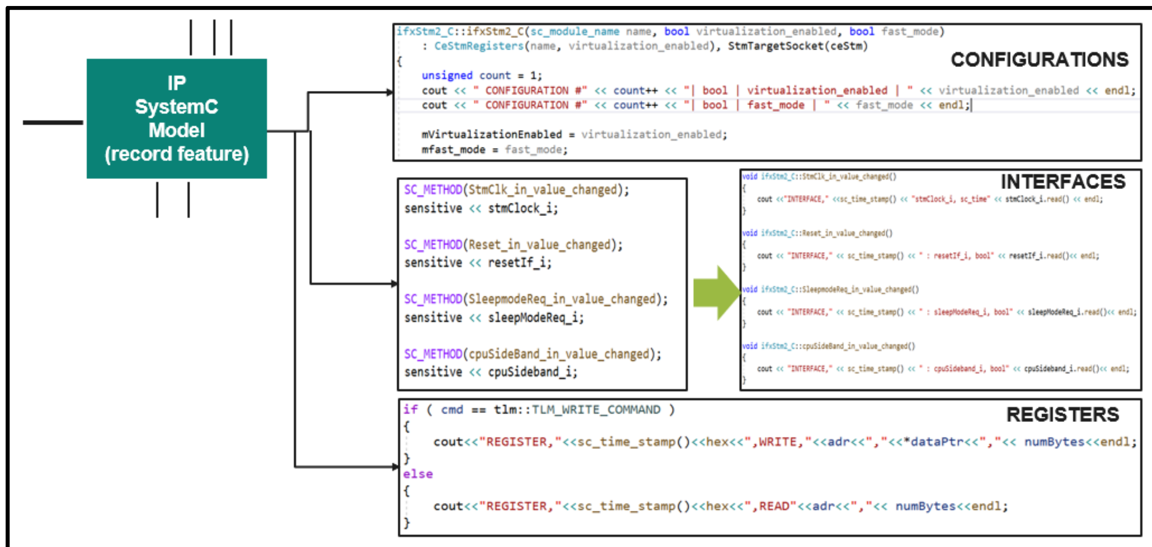
- Instrumentation of logging into the existing IP SystemC model
- Creation of a standalone reproducer environment.

Both of these prerequisites are achieved through an automation framework developed in-house.




### A. Instrumentation of logging into the existing SystemC model

This step involves enhancing the existing IP SystemC model by adding a recording feature. The instrumentation involves inserting code one time within the model's codebase that can capture and log various events and states during the model's execution. In order to enable the recording of events occurring at the model boundary, it's necessary to introduce extra callback functions and implement logging within these callback functions.




The following section outlines three categories of triggers that can impact the model's behavior along with an explanation of how to capture and record these events.

- Configurations

  Typically, models are designed with wide range of features and configurability helps in making the design more scalable and re-usable across many MCU VP platforms. Based on these static configurations, the model hierarchy is created during elaboration phase. These configurations remain unchanged throughout the simulation phase. These configurations are logged on simulation output during the model's construction in the following format.

  *Model Constructor (Model Name, Data Type1 Parameter Name1, Data Type2 Parameter Name2, …)*
  *{*
  *CONFIGURATION, 1, Data Type1, Parameter Name1, Value1*
  *CONFIGURATION, 2, Data Type2, Parameter Name2, Value2*
  *……*
  *}*

- Register Transactions

  During the simulation, the interactions between software and the model involve bus transactions to access registers and memory implemented in the IP model. These transactions are triggered by various bus masters like CPUs, DMA, etc. When the model receives these transactions, the model bahaviours might be triggered. By instrumenting the functions responsible for handing the bus transactions in the model, it becomes possible to capture all transaction details and then present them in the following format within the simulation log output.

```
Register Access Function (Transaction Object)
{
        REGISTER, Simulation Time, Address, Read/Write, Data, Data Length
}
```

- Interface Modifications

  In the simulation phase, external models might influence or modify values on the interfaces. The SystemC library provides capabilities for registering processes with the SystemC kernel, which can be invoked when specific types of events take place. To capture changes in values on interfaces, SystemC processes can be employed by registering interface modifications as events. As a result, whenever a value alteration occurs on the interfaces, the SystemC kernel triggers the corresponding processes. The recorded values are then extracted and presented in the simulation log output. In order to record interface modifications, below updates are required in the SystemC model. For the illustration purpose, logging method for reset interface is described in the below section.

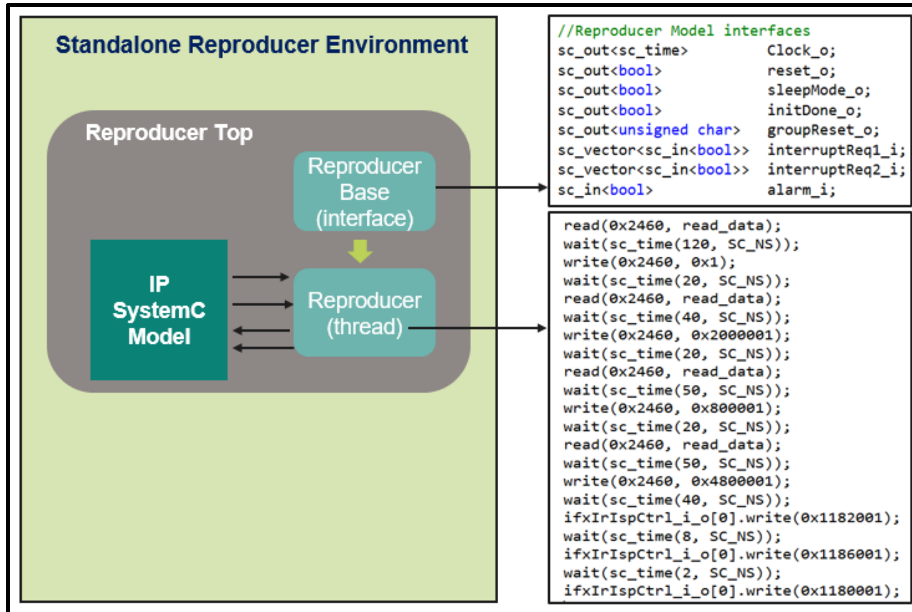
```
sc_in <bool> reset_in;          //interface declaration
void reset_in_value_changed(); // SystemC process triggered on reset_in interface value change

SC_METHOD(reset_in_value_changed)
Sensitive << reset_in;

void reset_in _value_changed ()
{
        INTERFACE, Simulation Time, INTERFACE_NAME, TYPE, CURRENT_VALUE
}
```

*B. Creation of Standalone Reproducer environment*

This step involves automated generation of independent reproducer environment to replicate the original behavior of the IP SystemC model. In this environment both Reproducer Model and the original IP SystemC model are integrated within Reproducer Top as shown in the figure below.



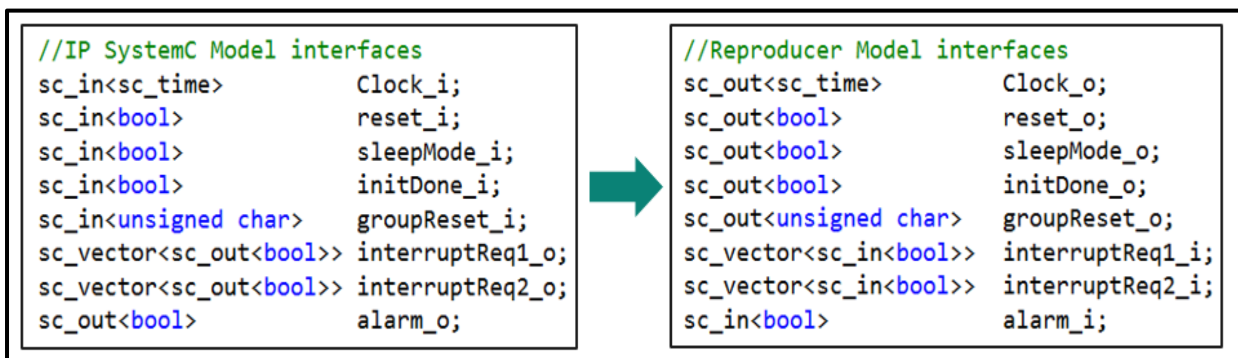

Reproducer model aims to mimic the end user simulation environment closely at the boundary of IP SystemC model. The Reproducer model is also designed to take the recorded data from the instrumentation step and simulate the same series of events at IP SystemC model boundary. This ensures that behavior of the IP SystemC model remains consistent, responding to the same events as it did in the original end user simulation environment.

The key elements of the Standalone Reproducer environment include

- Reproducer Base

  The Reproducer model has been designed to provide the interfaces that work together with the interfaces of the IP SystemC model in a complementary manner. To illustrate, if a model implements the input interface as 'sc_in<bool> clock_in', then the reproducer model implements the corresponding output interface as 'sc_out<bool> clock_out'. These two interfaces are meant to interact during simulation.

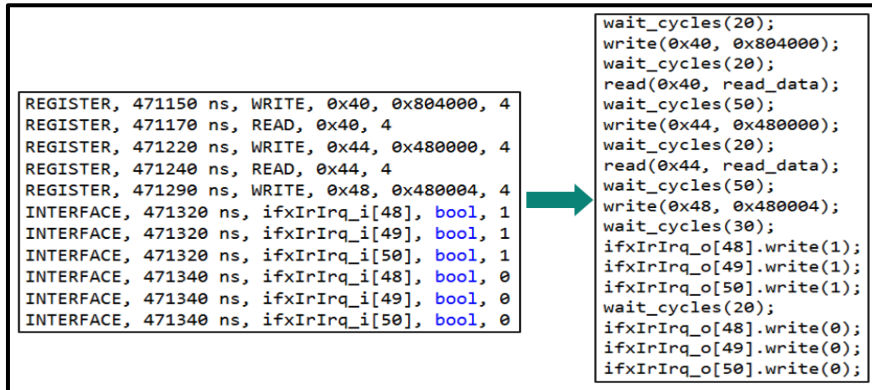
  Below figure illustrates IP SystemC model interfaces and Reproducer model complementary interfaces.




- Reproducer thread

  Within the Reproducer model, another significant component is a SystemC thread. In this thread, values logged at instrumentation step are used to drive on the complementary interfaces of Reproducer model. An In-house Automation Framework is used to translate the recorded data at instrumentation step into sequence of events that are to be executed within SystemC thread of Reproducer model.

Translation of recorded data into sequence of SystemC boundary trigger events is shown in below figure.

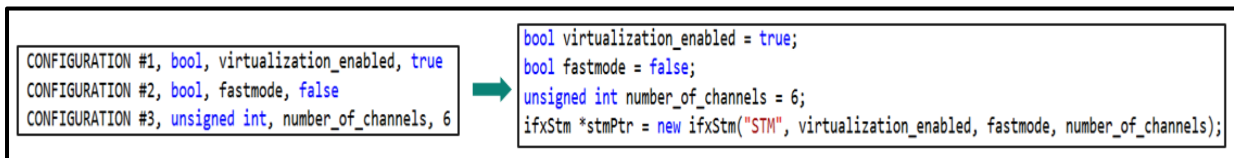


```
REGISTER, 471150 ns, WRITE, 0x40, 0x804000, 4
REGISTER, 471170 ns, READ, 0x40, 4
REGISTER, 471220 ns, WRITE, 0x44, 0x480000, 4
REGISTER, 471240 ns, READ, 0x44, 4
REGISTER, 471290 ns, WRITE, 0x48, 0x480004, 4
INTERFACE, 471320 ns, ifxIrIrq_i[48], bool, 1
INTERFACE, 471320 ns, ifxIrIrq_i[49], bool, 1
INTERFACE, 471320 ns, ifxIrIrq_i[50], bool, 1
INTERFACE, 471340 ns, ifxIrIrq_i[48], bool, 0
INTERFACE, 471340 ns, ifxIrIrq_i[49], bool, 0
INTERFACE, 471340 ns, ifxIrIrq_i[50], bool, 0
```

```
wait_cycles(20);
write(0x40, 0x804000);
wait_cycles(20);
read(0x40, read_data);
wait_cycles(50);
write(0x44, 0x480000);
wait_cycles(20);
read(0x44, read_data);
wait_cycles(50);
write(0x48, 0x480004);
wait_cycles(30);
ifxIrIrq_o[48].write(1);
ifxIrIrq_o[49].write(1);
ifxIrIrq_o[50].write(1);
wait_cycles(20);
ifxIrIrq_o[48].write(0);
ifxIrIrq_o[49].write(0);
ifxIrIrq_o[50].write(0);
```

- IP SystemC model

The configuration details recorded during the instrumentation step are used while instantiating the IP SystemC model within the Standalone Reproducer environment. This ensures that the model architecture remains consistent before starting the simulation. Below figure describes the recorded configuration values at instrumentation step and reusing the same configurations to instantiate IP SystemC model.



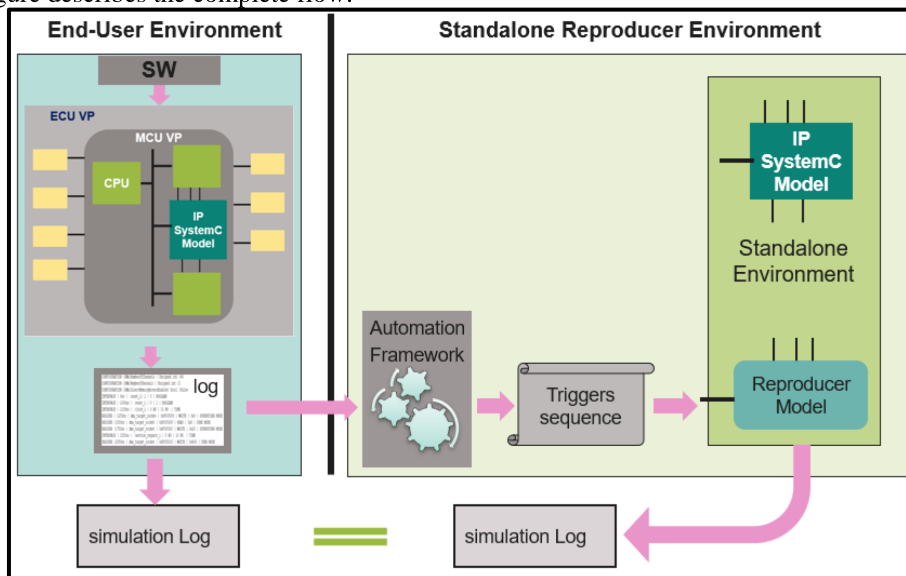
```
CONFIGURATION #1, bool, virtualization_enabled, true
CONFIGURATION #2, bool, fastmode, false
CONFIGURATION #3, unsigned int, number_of_channels, 6
```

```
bool virtualization_enabled = true;
bool fastmode = false;
unsigned int number_of_channels = 6;
ifxStm *stmPtr = new ifxStm("STM", virtualization_enabled, fastmode, number_of_channels);
```

## V. EXPERIMENTAL RESULT

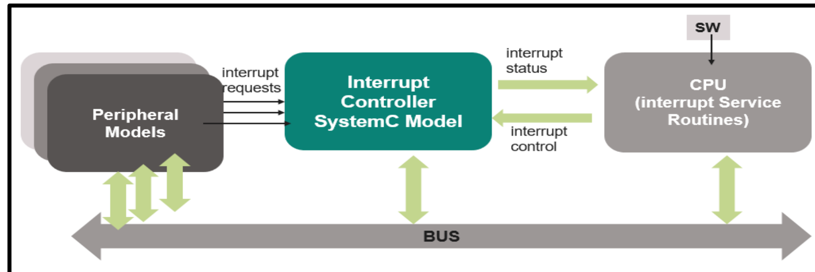
This method has been deployed internally for experimentation. In this section, experimental outcome will be presented utilizing 'Interrupt Controller' SystemC model. At first, a recording feature was instrumented into the Interrupt Controller SystemC model and a standalone reproducer environment was generated using in-house automation framework as detailed in section IV. To validate the method, simulation is carried out, and a simulation log is generated on two distinct environments: end-user environment and standalone reproducer environment. Finally, the simulation logs are compared. If identical messages within SystemC model are found in both logs, it serves as a confirmation of a successful reproduction.

Following figure describes the complete flow.



Following use-case is demonstrated on end-user environment and standalone reproducer environment

1) SW running on CPU configures the registers in Interrupt Controller and other peripheral models
2) Multiple Peripheral models triggers interrupt at Interrupt controller's interrupt request lines
3) Arbitration Unit in Interrupt controller arbitrates among all pending interrupt requests
4) Interrupt Controller drives the winning interrupt status on CPU Status input line
5) CPU executes the Interrupt Service Routine (ISR) associated with the requested interrupt
6) After ISR execution, CPU drives an acknowledgement details in Interrupt Controller input line
7) The interrupt controller iterates through the arbitration process again, and steps 3 to 7 are reiterated




The process of generating simulation logs in two separate simulation environments is detailed as follows:

- End-User environment

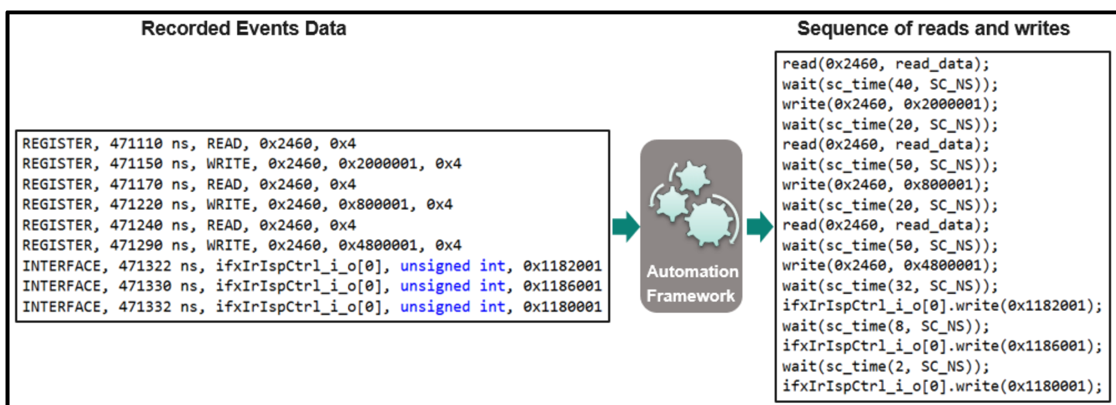
  End-user Environemnt comprises of a MCU/ECU virtual prototype where Interrupt Controller SystemC model with instrumented recording feature is integrated in it. In this environment, end user runs the orignal application (SW). Upon running a simulation, all boundary triggers (configuratin, register access, interface changes) at the Interrupt Controller model are logged and recorded as shown in below figure.

```
REGISTER, 471110 ns, READ, 0x2460, 0x4
REGISTER, 471150 ns, WRITE, 0x2460, 0x2000001, 0x4
REGISTER, 471170 ns, READ, 0x2460, 0x4
REGISTER, 471220 ns, WRITE, 0x2460, 0x800001, 0x4
REGISTER, 471240 ns, READ, 0x2460, 0x4
REGISTER, 471290 ns, WRITE, 0x2460, 0x4800001, 0x4
INTERFACE, 471322 ns, ifxIrIspCtrl_i_o[0], unsigned int, 0x1182001
INTERFACE, 471330 ns, ifxIrIspCtrl_i_o[0], unsigned int, 0x1186001
INTERFACE, 471332 ns, ifxIrIspCtrl_i_o[0], unsigned int, 0x1180001
```

  End-User shares the log with model developer.

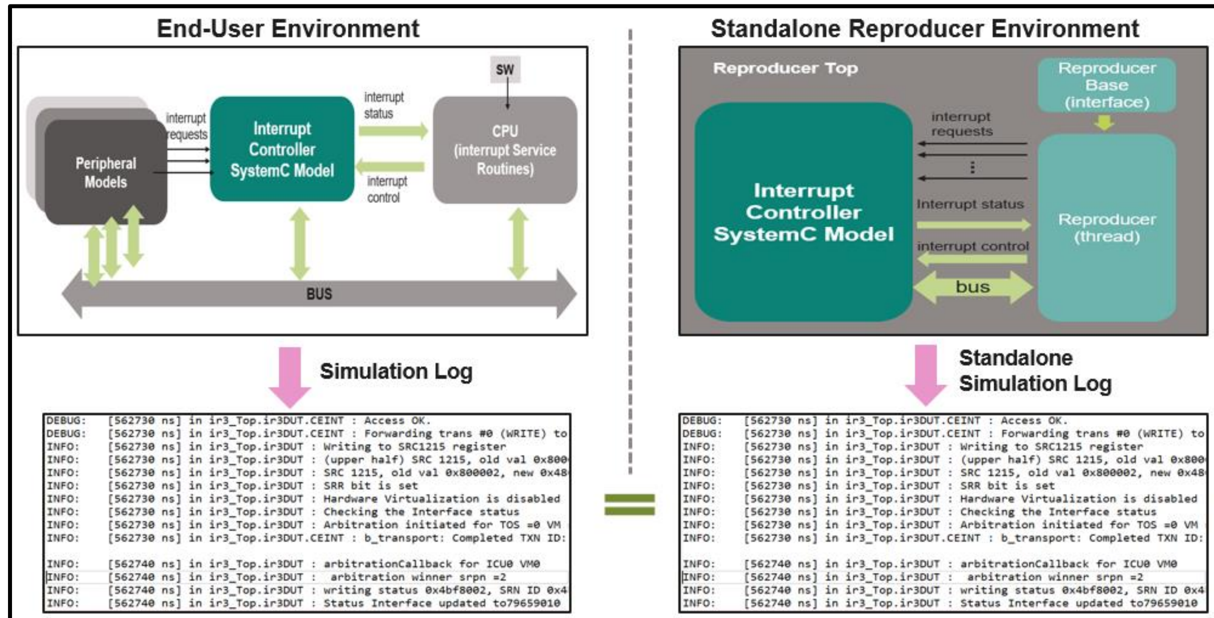
- Standalone Reproducer Environment

  Standalone Reproducer Environment consists of automation framework to translate the recoded events log to sequence of interfaces read and write events. This sequence is then used within the SystemC thread of the reproducer model. The figure provided below illustrates the process of transforming recorded events logs into a sequence of interface read and write events.



Within the Standalone Reproducer model, this event sequence is executed by the SystemC thread during simulation and simulation log is generated.

A comparison was performed between the simulation logs generated for the Interrupt Controller SystemC model in the end-user environment and the reproducer environment. Identical debug messages were consistently displayed in both simulation logs throughout the entire simulation process.




## VI. CONCLUSION AND FUTURE WORKS

This methodology is implemented for an existing SystemC model of an IP which is currently under development. The same would be extended for all SystemC models that are part of Infineon's next generation automotive microcontrollers. The future work also includes adapting this methodology for serial communication interfaces and bus master interfaces. Lastly, incorporation of dynamic simulation snapshots along with this methodology would be planned to leverage the debugging capabilities.

## VII. REFERENCES

[1] Sam Tennent, "Developing & Testing Automotive Software on Multi-SoC ECU Architectures using Virtual Prototyping," DVCon Europe, 2018.

[2] Amit Nene and Swaminathan Ramachandran, "Trace Based Approach for Unit Level Debug and Verification of C/C++ IP Models," Design & Reuse, 2008

[3] Nima Honarmand and Josep Torrellas, "Replay Debugging: Leveraging Record and Replay for Program Debugging," ISCA, 2014

[4] Nils Bosbach, Jan Moritz Joseph, Rainer Leupers, Lukas Jünge, "NISTT: A Non-Intrusive SystemC-TLM 2.0 Tracing Tool," IEEE Patras, Greece, 2022

[5] Harry Broeders and René van Leuken, "Extracting behavior and dynamically generated hierarchy from SystemC models", DAC,2011

[6] Stefan Kraemer; Rainer Leupers; Dietmar Petras; Thomas Philipp, "A checkpoint/restore framework for systemc-based virtual platforms", IEEE Tampere, Finland, 2009