In high-reliability and safety-critical applications, RT and gate-level fault-injection simulations are often performed in order to ensure a certain level of fault detection coverage which is necessary to ensure compliance with standards such as ISO 26262. There are many techniques available for accelerating the simulations including emulation platforms, however, in most cases, classifying the failing scenarios remains a manual task and is often the limiting factor in the number of fault injections that can be performed.

In this article, we show how the components of a UVM functional verification environment can easily be extended to record additional information about the types of errors that have occurred. This additional information can be used to classify failing tests based on their system level impact (e.g. Silent Data Corruption, Detected Uncorrected Error, etc.). We present an architecture that can be implemented on Mentor’s Questa® Verification Platform for designs with UVM DVE.

**INTRODUCTION**

The integrated circuits used in high-reliability applications must demonstrate low failure rates and high-levels of fault detection coverage. Safety Integrity Level (SIL) metrics indicated by the general IEC 61508 standard and the derived Automotive Safety Integrity Level (ASIL) specified by the ISO 26262 standard specify specific failure (FIT) rates and fault coverage metrics (e.g. SPFM and LFM) that must be met. To demonstrate that an integrated circuit meets these requirements requires a combination of expert design analysis combined with fault injection (FI) simulations. During FI simulations, specific hardware faults (e.g. transients, stuck-at) are injected in specific nodes of the circuits (e.g. flip flops or logic gates).

Designing an effective fault-injection platform is challenging, especially designing a platform that can be re-used effectively across designs. In this article, we outline the architecture for a complete FI platform. We show how this architecture can be easily integrated into a general purpose design verification environment (DVE) that is implemented using UVM.

**REQUIREMENTS OF A FI ENVIRONMENT**

The purpose of a FI environment is to measure the effect of circuit level faults on a high level application such as an automotive electronic control unit (ECU) that controls a braking system. The DVE that is used for functional design validation provides the key to understanding how a low-level fault affects the behavior of the full integrated circuit. When a fault causes an error at the chip level, by interpreting the error messages that are produced (e.g. interrupts, mis matching output data, …), design and system engineers can map the chip-level behavior to a relevant system level effect. To meet safety goals, the requisite fraction of faults must produce safe effects. The table below enumerates the key requirements and features of a FI environment:

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Scalable</strong></td>
<td>Typically, it is necessary to inject tens of thousands or hundreds of thousands of faults. This creates a large volume of data which must be managed. This data includes the list of faults which must be injected, the list of faults that have already been injected and the effect that they produced. Managing this data in the form of ad-hoc text files is not a scalable approach, thus a well-organized, relational data-base is a requirement.</td>
</tr>
<tr>
<td><strong>Re-Usable</strong></td>
<td>The investment in the FI Environment must be re-usable across multiple designs. The amount of code that must be customized for each design must be kept to a minimum. Those parts of the system that are not directly tied to the simulator can be implemented externally. Those parts that interact directly with the DVE must be coded using a standard verification methodology such as UVM, so that they can be easily integrated.</td>
</tr>
</tbody>
</table>
In Figure 1 we show the proposed architecture for a FI platform. The original DVE is shown in blue and the elements of the FI platform in red. The core element in this architecture is the FI database (FIDB) which holds all data related to the faults scheduled to be simulated and those that have been simulated, including their effects. This is implemented using an industrial relational database platform.

The web interface provides all users with a view of the fault campaigns and the results, including the computation of metrics (SPFM). The web interface also allows users to set parameters for and schedule new FI campaigns.

With this architecture, the simulation jobs are launched to the compute farm without a preconceived notion of which fault they will simulate. Instead, when the generic FI simulations are launched, the simulator queries which fault to inject. This is done through a combination of the UVM FI Extension and the FI VPI interacting with the FIDB. As the simulation executes and then completes, data is sent back to the FI DB about the impact of the fault. An in-depth description of all aspects of the platform is beyond the scope of this article. Instead, the focus of this article is on how the use of UVM facilitates the integration of the FI platform into a DVE. We discuss two aspects of this problem. First, we show how the timing of the fault injection can be controlled using UVM. We also show how the results reporting can be implemented efficiently using UVM.

**TEMPORAL CONTROL OF FAULT INJECTION**

When performing a fault-injection simulation, there are fundamentally three main tasks that need to be performed. First, the simulator must determine which fault to inject. Second, at the appropriate time, the fault must be injected and finally, during the remainder of the simulation, the result of the fault must be assessed. Prior to the adoption of standardized verification methodologies, the timing of these tasks had to be done in an ad hoc or customer specific fashion. However, using the UVM run-time schedule, these tasks can be easily coordinated.

Referring to the proposed architecture in Figure 1, during the **uvm_pre_reset_phase**, the UVM FI extensions can query the FI DB to determine which fault to inject. Since the database resides outside the simulator, a VPI library routine ($fi_setup) is required. This VPI routine issues the necessary queries to first register that the simulation has

**Figure 1: High-Level Architecture of an FI Platform**

![Figure 1: High-Level Architecture of an FI Platform](image)
started running. Then it obtains the fault to be simulated during the current run. Typically, the fault consists of a 3-tuple consisting of: the time when the fault should be injected, the target node where the fault should be injected and the type of fault to inject (e.g. transient, stuck at,...).

In a naïve implementation, the simulation would execute normally until it is time to inject the fault. However, significant simulation time can be saved by using $save/$restore to quickly advance the simulator state to a time that is close to the injection time.

Rather than referencing the time for the fault relative to time zero, it makes more sense to take the start of the uvm_main_phase as the reference. This way, one can avoid injecting faults during the configuration phase and the time offset for fault injection remains valid, even if the time required for configuration is variable, for example due to randomization of the configuration.

From these simple examples, we see that the UVM run-time phases make it possible to synchronize fault injections in a generic fashion.

RESULTS REPORTING
Prior to the adoption of standardized verification methodologies such as UVM, error logging was typically performed using the $display statement. In such legacy DVEs, the results of the simulation log had to be analyzed using scripts to determine if any errors had occurred. When adding fault injection capabilities to a DVE, it was necessary to know the patterns to search for in the log file in order to identify errors. Then, as an additional step, these error messages had to be mapped to system effects.

With UVM, systematic message reporting is implemented with messaging methods and macros.

Using the uvm_report_catcher a call back can be systematically added to all message reporting calls. Using this capability, all messages with a severity of uvm_error can be detected. A copy of the message is then stored in the FI DB, via the FI VPI. The time, id and message string are stored in the FI DB.

The fact that the time of the messages can be stored in the database is significant. The standards require that the fault reaction time be shown to be lower than the fault tolerant time interval. By recording the time of the fault injection and the time of the UVM messages which indicate that the fault has been detected in the FI DB, the fault reaction time can be recorded. Using the web interface, the users can issue queries to extract metrics such as the fault reaction time.

The proposed approach of systematically intercepting all the error reporting messages in a DVE using the uvm_report_catcher requires only a minimum amount of code and it is compatible with all UVM based DVEs. Using VPI routines, this data can be quickly exported to a relational database, where the results can be analyzed off-line.

CONCLUSIONS
Due to standards such as ISO 26262, there is a growing need to perform complex fault injection campaigns on complex ASICs using advanced DVEs. The core of any fault injection platform is a robust database which can manage the large volume of data and such a database must be external to the simulator.

In the past, the architecture of DVEs varied widely making it difficult to provide a generic fault injection platform. However, through a careful partitioning of the FI platform (as shown in Figure 1) and through judicious use of UVM to interface with the DVE, a generic FI platform is possible. This reduces the development costs associated with FI analysis and makes it faster to show compliance with the quantitative reliability and fault detection metrics specified in the standards.
VERIFICATION ACADEMY
The Most Comprehensive Resource for Verification Training

27 Video Courses Available Covering
• SystemVerilog OOP
• Formal Verification
• Intelligent Testbench Automation
• Metrics in SoC Verification
• Verification Planning
• Introductory, Basic, and Advanced UVM
• Assertion-Based Verification
• FPGA Verification
• Testbench Acceleration
• PowerAware Verification
• Analog Mixed-Signal Verification

UVM and Coverage Online Methodology Cookbooks
Discussion Forum with more than 6800 topics
Verification Patterns Library

www.verificationacademy.com