# Fault-injection-Enhanced Virtual Prototypes Enable Early SW Development for Automotive Applications

Mohammad Badawi, Javier Castillo, Andreas Mauderer, Jan-Hendrik Oetjens, Robert Bosch GmbH, Reutlingen, GERMANY

***Abstract*—Using virtual prototypes to enable early software development has become widely accepted approach in the in automotive industry. However, evaluating the dependability of programmable System-on-Chips must involve both hardware and software combined and therefore, the virtual prototype must also provide the needed capabilities for software development to implement and qualify software safety mechanisms. In this paper, we present a fault injection framework to raise the level of abstraction in fault modeling and to ease performing and tracing user-defined fault combinations. Using real industrial virtual prototypes, we demonstrated the utilization of the presented fault injection framework and we found that the performance overhead fault is below 1% for real-world applications.**

***Keywords— Virtual Prototyping; Fault Injection; System dependability; ISO 26262; SystemC; TLM.***

## I. Introduction

The use of programmable System-on-Chips (SoCs) in modern automotive applications has been continuously increasing. As the automotive applications get more sophisticated, the utilized SoCs get more complex. Additionally, strict safety standards, e.g., ISO26262, must be met and the SoCs must go through an extensive evaluation process to ensure their reliability and safety in the presence of failures. This process constitutes evaluating the hardware (HW) as well as the software (SW) [1], in addition to the SW-based mechanisms that react to HW faults. Once all these components are available, it is possible to conduct dependability evaluation processes such as fault simulation.

Discovering inconsistences between the design and specification earlier in the design process can reduce project cost and time to market. One method to accomplish this is to prepare SW before HW is available, i.e., early SW development. Additionally, SW-based safety mechanisms such as error detection mechanisms (EDM), error recovery mechanisms (ERM), and fault management [2] need to be ready to conduct verification and evaluation of dependability and safety. The increased complexity of programmable SoCs used in automotive applications, has made it infeasible to use design representations at gate level (GL) or register transfer level (RTL). Their slow simulation makes it impossible to develop SW and execute real-life scenarios, and their late availability risks delaying the evaluation of system-level safety. Furthermore, enhancing SW with EDM and ERM imposes complexity and performance overhead as mentioned in [3], forcing SW development to start even earlier to cope with additional iterations and qualification tasks that are required. Therefore, virtual prototypes (VP) at high abstraction levels, such as transactional level models (TLM), became a widely accepted solution in the industry for enabling early SW development [1][4].

In this paper, we focus on the industrial application of VPs to enable early development of SW-based safety mechanisms that are needed to tolerate and react to HW failures. In our approach, we use a fault injection interface, and we instrument the VP with fault injecting callback functions to model transient faults (single-event upset (SEU) and multiple-event upset (MEU)) and permanent faults in registers, communication, and computation. Furthermore, our approach reports comprehensive details regarding the time and types of faults injected in the test cases. With the correct post-processing, it would be possible to detect fault patterns and dependencies between faults within the simulation.

## II. Related Work

The work in [5] proposed an approach to improve the confidence level of fault injection into VPs, which assumes that the GL model of the design is available, but without safety features. The authors used an SoC with processing elements and memory as case study. The TLM sockets in the VP of the SoC were instrumented by callback functions to inject SEU faults. Simulations were performed for GL and VP together to eliminate failures that were masked at VP boundary. Consequently, the authors could identify realistic and unrealistic failures. Our work uses similar callback functions, but we use more types of callbacks in different locations, and we use lookup tables that provide mapping between module boundaries in HW and VP.

The solutions presented in [6,7] also studied the correspondence between faults in RTL and TLM. Our approach is different from the solutions [5-7] since we target applying our method in a new project to enable early development of SW-based safety mechanisms, where RTL and GL reuse is not possible.

Kooli et al. in [1] presented a mutation analysis method for SW testing and for HW reliability to study how faults are propagated between HW and SW layers. In our work, we use callback functions to manipulate the result of computation functions to enable the analysis of accumulated faults.

Oetjens et al. in [4] elaborated the advantages of employing virtual prototypes to evaluate the functionality of the system and highlighted the consistency between VP and HW as the most important challenge in this approach. Building on this outcome, our approach aims to tighten the feedback loop between SW, RTL, and safety.

## III. Fault Injection Framework

The fault injection framework presented in this paper models transient faults (SEU and MEU) as well as permanent faults in registers, communication, and computation. As shown in Figure 1, the fault injection framework consists of the fault injection interface, the fault injection callback functions, and the reporting and analysis.

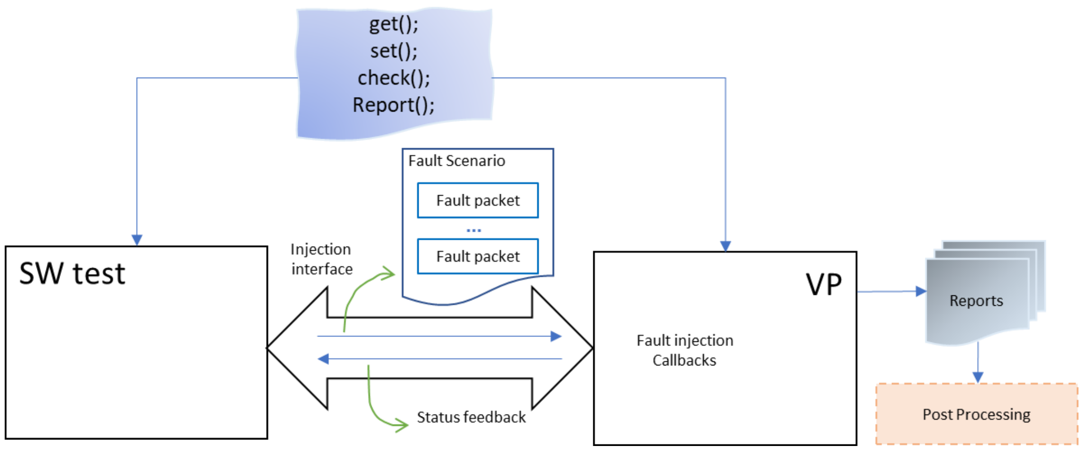
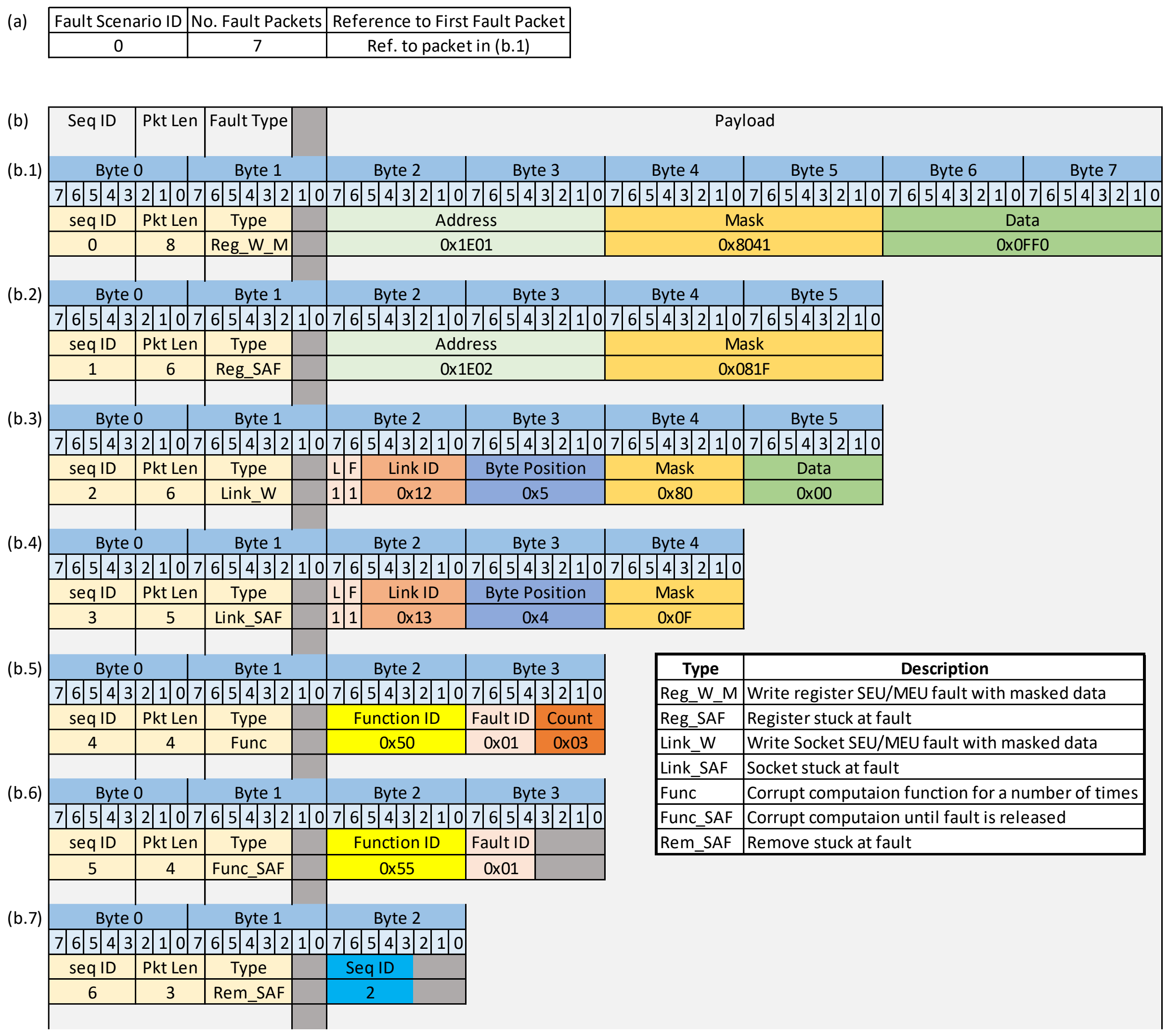


Figure 1: The Fault Injection Framework

### A. *Fault injection interface*

The fault injection interface enables data exchange between SW (or tester) and virtual prototype. It allows SW to inject fault scenarios into the VP model and provides feedback information to the SW. Faults are injected into the VP in form of fault packets that are encapsulated within a fault scenario. A fault scenario is a set of faults that have been randomly grouped together or carefully tailored for complex fault combination. The fault scenario has a unique ID within the test case to ease traceability and analysis. It includes a reference to the first fault packet in the scenario (as shown in Figure 2(a), and it specifies the number of fault packets. This way the specified number of fault packets will be injected consecutively starting from the first packet. The data structures for the scenarios and the fault packets are parameterized and can be included as a header file at SW and VP sides. This way the data structures in the header can be specialized and the bit width of the fields can be selected depending on the requirements of the use case.

The fault packet has a fixed header length and a variable payload length. The header has a common structure, which includes the sequence ID, the packet length (to be used for integrity checks), and the packet type. Using the packet type, the payload is parsed and translated into injectable faults that are mapped to specific fault injecting callback functions.



(a)

| Fault Scenario ID | No. Fault Packets | Reference to First Fault Packet |
|---|---|---|
| 0 | 7 | Ref. to packet in (b.1) |

(b)

| Seq ID | Pkt Len | Fault Type | | Payload |
|---|---|---|---|---|

(b.1)

| Byte 0 | Byte 1 | Byte 2 | Byte 3 | Byte 4 | Byte 5 | Byte 6 | Byte 7 |
|---|---|---|---|---|---|---|---|
| 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 |

| seq ID | Pkt Len | Type | | Address | Mask | Data |
|---|---|---|---|---|---|---|
| 0 | 8 | Reg_W_M | | 0x1E01 | 0x8041 | 0x0FF0 |

(b.2)

| Byte 0 | Byte 1 | Byte 2 | Byte 3 | Byte 4 | Byte 5 |
|---|---|---|---|---|---|
| 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 |

| seq ID | Pkt Len | Type | | Address | Mask |
|---|---|---|---|---|---|
| 1 | 6 | Reg_SAF | | 0x1E02 | 0x081F |

(b.3)

| Byte 0 | Byte 1 | Byte 2 | Byte 3 | Byte 4 | Byte 5 |
|---|---|---|---|---|---|
| 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 |

| seq ID | Pkt Len | Type | | L | F | Link ID | Byte Position | Mask | Data |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 6 | Link_W | | 1 | 1 | 0x12 | 0x5 | 0x80 | 0x00 |

(b.4)

| Byte 0 | Byte 1 | Byte 2 | Byte 3 | Byte 4 |
|---|---|---|---|---|
| 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 |

| seq ID | Pkt Len | Type | | L | F | Link ID | Byte Position | Mask |
|---|---|---|---|---|---|---|---|---|
| 3 | 5 | Link_SAF | | 1 | 1 | 0x13 | 0x4 | 0x0F |

(b.5)

| Byte 0 | Byte 1 | Byte 2 | Byte 3 |
|---|---|---|---|
| 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 |

| seq ID | Pkt Len | Type | | Function ID | Fault ID | Count |
|---|---|---|---|---|---|---|
| 4 | 4 | Func | | 0x50 | 0x01 | 0x03 |

(b.6)

| Byte 0 | Byte 1 | Byte 2 | Byte 3 |
|---|---|---|---|
| 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 |

| seq ID | Pkt Len | Type | | Function ID | Fault ID | |
|---|---|---|---|---|---|---|
| 5 | 4 | Func_SAF | | 0x55 | 0x01 | |

(b.7)

| Byte 0 | Byte 1 | Byte 2 |
|---|---|---|
| 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 |

| seq ID | Pkt Len | Type | | Seq ID | |
|---|---|---|---|---|---|
| 6 | 3 | Rem_SAF | | 2 | |

| Type | Description |
|---|---|
| Reg_W_M | Write register SEU/MEU fault with masked data |
| Reg_SAF | Register stuck at fault |
| Link_W | Write Socket SEU/MEU fault with masked data |
| Link_SAF | Socket stuck at fault |
| Func | Corrupt computaion function for a number of times |
| Func_SAF | Corrupt computaion until fault is released |
| Rem_SAF | Remove stuck at fault |

Figure 2 Fault Scenario and Fault Packets with Example Values

## B. *Fault Injection Callback functions*

In our approach, we instrument the VP with fault injecting callback functions that can be triggered before or after the execution of the function that is subject to fault. We use three types of callbacks depending on the modeled fault as follows:

1. TLM socket callback: Used to model a communication fault in an interconnect, interface or register port that is directly accessible by the socket. We extended the callback concept described in [5] to support multiple faults in addition to single faults. Nevertheless, using this callback allows modeling a fault only when reading or writing to a register occurs via the TLM socket. In other words, the fault occurs only when a TLM transaction is triggered. To inject faults to registers without a TLM transaction, a special register access callback is required.
2. Register access callback: Used to model a register fault at any point in time, without the need of a TLM transaction. This way, it is possible to the trigger the fault and proceed with the execution to examine the effect of the presented fault. For example, forcing a timeout before the time window has ended without explicitly triggering a socket read or write transaction.

3. Function corruption callback: Used to hook a customized function (like “saboteur” concept in [8]) to a targeted data processing function in the VP model to corrupt its outcome, e.g., suppress or manipulate the resulting data. This type of callback enables raising the level of abstraction when representing faulty computation and makes it easier for SW to model and examine accumulated faults.

### *C. Integration*

To elaborate in more detail, we describe next how each type of fault packet and fault callback functions are integrated together to make fault injection possible.

*1)* To inject a transient fault to an interconnect or an interface, the packet type Link_W shown in Figure 2(b.3) is used. It specifies the link that is subject to the fault with an ID. The IDs of the links are provided to SW use case in form of a lookup table that maps the implemented socket in the VP to the actual HW that is subject to fault. The packet states the faulty data byte to inject, as well as which byte of the payload to inject the fault to. If multiple bytes need to be corrupted, multiple Link_W packets are used and all of them will have the same sequence ID to ease tracing. The packets can then be distinguished by the value of **L**ast and **F**irst flags during execution, since the first packet has (L=0, F=1), the last packet has (L=1, F=0) and all middle packets have (L=F=0). This way the fault can be set up. However, the fault will be triggered when the socket fault callback is executed the first time the socket is accessed after this setup. In case of permanent faults, the packet type Link_SAF shown in Figure 2(b.4) can be used and it will trigger the fault in the socket whenever it is accessed until the fault is released.

*2)* To inject a transient fault to a register (SEU or MEU), the fault packet type Reg_W_M shown in Figure 2(b.1) is used. This fault type contains the register address and the faulty data used to replace the content in the register. The mask can be used to enable modifying only a subset of bits in the register. Similarly, the fault packet Reg_SAF shown in Figure 2(b.2) can be used to inject a permanent fault forcing the value of the register to be changed according to the provided mask. In this case the faulty behavior remains until it is released. Both packet Reg_W_M and Reg_SAF result in activating the register callback function of the targeted register to change its content.

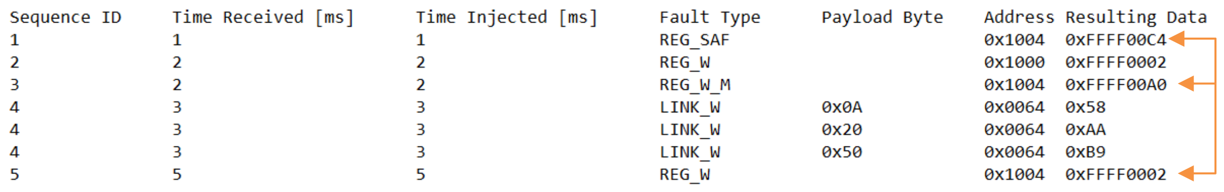
*3)* To inject a computation fault, the packet type Func shown in Figure 2 Fault Scenario and Fault Packets with Example Valuescan be used. As a result, the targeted function (specified by Function ID) will be hooked to the pre-defined corrupting callback function (specified by fault ID). At this point, the fault is configured, but it will only be triggered when the targeted function is executed. The faults will be triggered for a limited number of calls (defined by Count in the packet). In case of permanent faults, the packet type Func_SAF shown in Figure 2(b.6) can be used and the fault will persist until it is released. Since the computation functions in VP represents processing units in HW perspective, a mapping table between the HW implementation and function implementation in the VP is provided to SW to abstract the details of functionality.

*4)* To release a permanent fault, the packet type Rem_SAF shown in Figure 2 Fault Scenario and Fault Packets with Example Values(b.7) can be used. The sequence ID in the release fault packet identifies the fault to be removed.

### *D. Reporting and analysis*

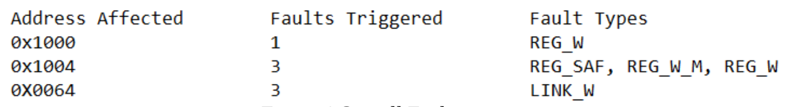
Our framework collects multiple types of information about the fault injection at run time and generate detailed logs. We report which faults were injected, the time at which the fault packet was received, the time at which the fault was triggered, the address of the register it is affecting, and the data and mask of the fault. This report is generated in different formats, i.e., CSV and XML files to simplify post-processing for further analysis.
By analyzing the files, we can determine which faults are overlapping and affecting the same register. We can determine if there is a dependency between any of the faults injected in the test case. The reports provide detailed information that can be post-processed as required by the use case.
An example of the fault injection reporting is shown in Figure 3 and Figure 4. The example is shown for a plain text log to ease readability. Figure 3 Example log of a Fault Injection Scenarioshows that when a fault packet is received by the VP and when a fault is triggered in the internal modules. A permanent fault will be triggered in the VP at multiple points in time until the fault is released, whereas a transient fault will be triggered only once. As shown in the figure, REG_SAF is applied to register 0x1004 and any subsequent write/read to this register will be affected by this fault, i.e., this occurs at time 2ms and 5ms. This kind dependency between faults is important to

take into consideration and more advanced analyses are required to detect such patterns automatically. The log also shows a summary for the number of faults executed and the addresses affected, as shown in Figure 4.



```
Sequence ID      Time Received [ms]      Time Injected [ms]      Fault Type      Payload Byte      Address Resulting Data
1                1                       1                       REG_SAF                           0x1004  0xFFFF00C4
2                2                       2                       REG_W                             0x1000  0xFFFF0002
3                2                       2                       REG_W_M                           0x1004  0xFFFF00A0
4                3                       3                       LINK_W          0x0A              0x0064  0x58
4                3                       3                       LINK_W          0x20              0x0064  0xAA
4                3                       3                       LINK_W          0x50              0x0064  0xB9
5                5                       5                       REG_W                             0x1004  0xFFFF0002
```

Figure 3 Example log of a Fault Injection Scenario



```
Address Affected          Faults Triggered          Fault Types
0x1000                    1                         REG_W
0x1004                    3                         REG_SAF, REG_W_M, REG_W
0X0064                    3                         LINK_W
```

Figure 4 Overall Fault statistics

## IV. CASE STUDIES

In our solution, we developed a VP of an SoC under design for the automotive industry. We conducted experiments to measure the performance and overhead associated with parsing the fault packets and triggering the corresponding fault injection callbacks. We demonstrate the flexibility of the framework by considering the following two industrial VP integration methods for our experiments.

### A. *Simulation based Functional Mock-up Units (FMU)*

This use case is a single simulation process that includes the VP model and a mock-up unit representing the CPU. Here, the simulation environment is responsible for synchronization and communication between the VP and the CPU mock-up unit. Communication is handled with the use of global variables that are mapped to the ports and signals of the VP. These global variables are updated periodically before each time step in the simulation. Fault injection is possible with the use of extra global variables. The VP reads these global variables and forwards the faults to the internal modules. Two main tests were executed: the first consisted of determining the overhead of each fault type and the second consisted of measuring the overhead of a specific fault scenario and its effect on various VP models.

For the first test, we used the minimal VP model shown in Figure 5. This VP has a single slave module and a single SPI master; certain faults were injected and parsed at the top level and then forwarded to the slave module. This minimalistic VP model facilitates the visualization of the overhead induced by the fault injection mechanisms. We measured the total overhead caused by all the faults injected in a single run and then calculated the overhead per fault. This process was repeated 10 times for each fault type to obtain robust measurements.

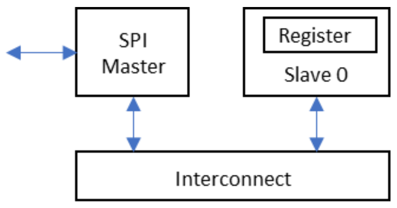


Figure 5 Example VP for fault overhead measurements

In this test, we injected sets of 50, 500, 1000 and 2000 register faults into the VP and measured the total time required to inject these sets of faults. In other words, we measured the total overhead caused by injecting 50 consecutive register faults in the VP, then we repeated the test with 500 faults, 1000 faults and finally with 2000 faults. This process was repeated 10 times for each register fault type and the average overhead for each set of faults was calculated (column Overhead [ms] in Table 1). Then we calculated the overhead per fault by dividing the total overhead by the number of faults injected in that run (column Overhead per Fault [ms]). The register faults applied were direct writes to registers, i.e., overwrite current register data with new data (Reg_W) and masked writes to register (Reg_W_M). The process and measurements were repeated by injecting 10 link faults (Link_W and Link_SAF) into the model. Table 1 indicates the results obtained.

Table 1 Measured Overhead for faults

| Fault Type | Num Faults | Overhead [ms] | Overhead Per Fault [ms] |
|---|---|---|---|
| Reg_W | 50 | 0.518 | 0.010 |
| Reg_W | 500 | 4.103 | 0.008 |
| Reg_W | 1000 | 8.370 | 0.008 |
| Reg_W | 2000 | 17.776 | 0.009 |
| Reg_W_M | 50 | 0.699 | 0.014 |
| Reg_W_M | 500 | 7.288 | 0.015 |
| Reg_W_M | 1000 | 14.761 | 0.015 |
| Reg_W_M | 2000 | 27.705 | 0.014 |
| Link_W | 10 | 0.003 | 0.0003 |
| Link_SAF | 10 | 0.006 | 0.0006 |

The second test case consisted of injecting a fault scenario into real-life applications; we used the example VP and two other VP models of ASICs used in the automotive industry (Project A and Project B). The internal structure of these VPs is shown in Figure 6; however, the configuration and number of SystemC processes is different between the two projects.

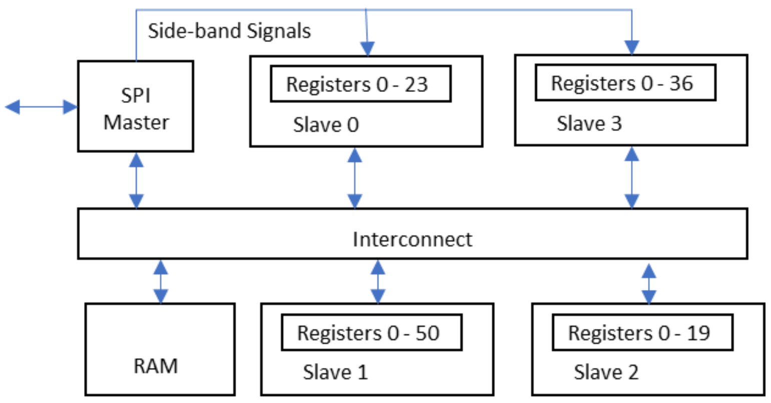


Figure 6 Simplified block diagram of VPs used for fault injection

For this test, we created a fault scenario where multiple faults of different types are injected into the models, and we measured the overhead caused by this. The scenario we examined consisted of the following faults: 3 Sock_W, 3 Sock_Saf, 50 Reg_W and 50 Reg_W_M. The average overhead was measured and compared with the total execution time of the user-defined applications executed on the three models. The resulting execution times with and without fault injection are presented in Figure 7.

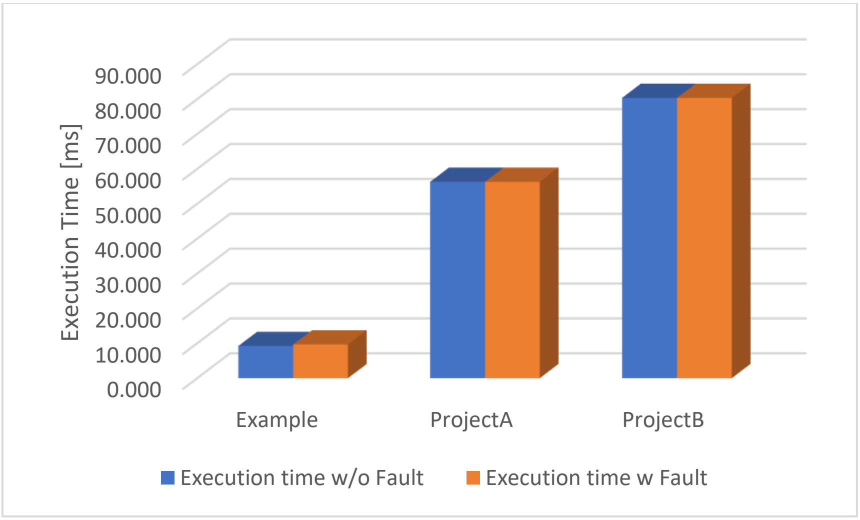


Figure 7 Execution of different applications with and without fault injection.

As can be seen in Figure 7, almost no difference is observable between the execution time of the applications with and without fault injection. The average overhead of the fault injections for this scenario is about 0.441 ms. When comparing this with the execution time of the example application, it represents 4.78% of its total execution time. However, this is a very simple example that does not contain a realistic application. Therefore, we repeated this calculation for Project A and Project B and determined that the fault injection represents 0.78% and 0.55% of the execution time of each project, respectively.

### B. *Multi-process simulation*

This scenario consists of two processes: the fault injection and external stimulus process (client) and the VP running a SystemC kernel (server). The communication between the processes is based on inter-process communication (IPC) using Berkeley sockets [9]. An advantage of this approach is it robustness compared to using global variables and its portability since support for sockets is common in industrial simulators.

This approach uses a single IPC interface for providing normal external stimulus/data and faults to the VP, where an additional header is used to distinguish between fault and data. Nevertheless, the flexibility of this approach allows us to support separate interfaces for data and faults.

In the current solution, we transmit the time information together with the fault packet and the server decodes the message and forwards the faults to the internal modules. The simulation time is used to synchronize both processes. The payload allows for further functionality to be developed, i.e., new faults can be easily added and decoded.

In this simulation environment we used the VP for Project A with a user-defined test case. We then injected the same fault scenario mentioned in Simulation based Functional Mock-up Units (FMU), however now the faults were transmitted using IPC instead of global variables. The expected execution time of this simulation is much greater than before due to the added overhead of the IPC mechanism. Nevertheless, the parsing of the fault packets should be similar to the first simulation environment. The obtained execution time with and without fault injection is shown in Figure 8. The overhead of the fault injection represents 0.00003% of the total execution time.

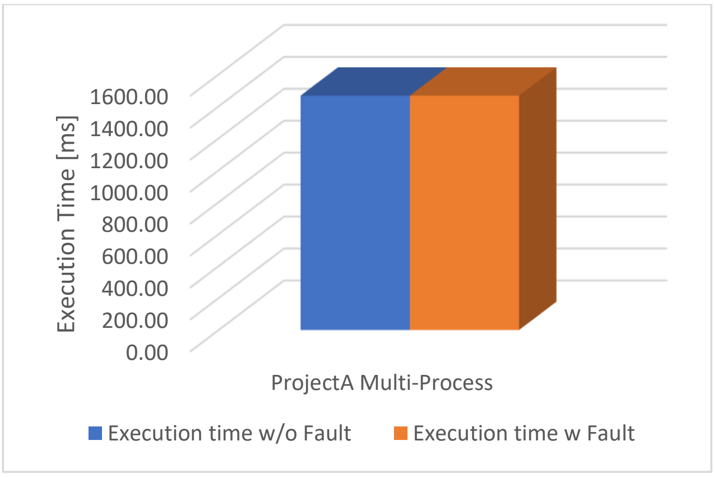


Figure 8 Execution time for Multi-process Simulation for Project A with and without fault injection

## V. SUMMARY

This paper emphasizes the advantages of using abstract virtual prototypes to enable early development of SW-based safety mechanisms at design phase, where RTL and GL are either not yet available or not possible to use for simulating real-life use cases due to their limited simulation speed. Our framework benefits from state-of-the art solutions and provides additional techniques to increase integration flexibility and raise the level of abstraction in fault modeling to ease the examination of user defined faulty behavior. We proposed using fault injection scenarios as fault envelopes and introduced sequence identifiers to ease traceability. To demonstrate the flexibility in integrating our framework, we conducted two case studies for real-life automotive applications. The first use case is based on single process mock-up functional unit. The second use case is based on multi processes with IPC, where the same interface is shared between fault packets and traffic data. We measured the average per-fault overhead that is associated to using our framework and we found it to be negligible. We composed different fault scenarios and found the average induced overhead to be less than 1% of the total execution time of the test case. We also provided comprehensive reporting which can enable further understanding of failure behavior and the relation between failures.

## VI. Future Work

Following the work presented in this paper, we plan to enhance fault triggering based on time and user-defined event patterns. We will finalize the features for tracing fault propagation paths to better understand the masked faults and construct the dependencies between different faults. To improve tool integration and operability, we aim at utilizing available industrial standards, like IPXACT, to automatically generate fault injection interface based on use case requirements and to implement standard-compliant reporting to enable post processing with different tools.

## VII. ACKNOWLEDGMENT

This work has been partially supported by the German Federal Ministry of Education and Research (BMBF) in the project MANNHEIM-FlexKI under grant 01IS22086A.

## References

[1] Maha Kooli, Firas Kaddachi, Giorgio Di Natale, Alberto Bosio, Pascal Benoit, Lionel Torres, Computing reliability: On the differences between software testing and software fault injection techniques, Microprocessors and Microsystems, Volume 50, 2017, Pages 102-112.

[2] C. Vitucci, D. Sundmark, M. Jägemar, J. Danielsson, A. Larsson and T. Nolte, "A Reliability-oriented Faults Taxonomy and a Recovery-oriented Methodological Approach for Systems Resilience," 2022 IEEE 46th Annual Computers, Software, and Applications Conference (COMPSAC), Los Alamitos, CA, USA, 2022, pp. 48-55, doi: 10.1109/COMPSAC54236.2022.00016.

[3] H. Schirmeier, M. Hoffmann, C. Dietrich, M. Lenz, D. Lohmann and O. Spinczyk, "FAIL*: An Open and Versatile Fault-Injection Framework for the Assessment of Software-Implemented Hardware Fault Tolerance," 2015 11th European Dependable Computing Conference (EDCC), Paris, France, 2015, pp. 245-255, doi: 10.1109/EDCC.2015.28.

[4] .J. . -H. Oetjens et al., "Safety evaluation of automotive electronics using Virtual Prototypes: State of the art and research challenges," 2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC), San Francisco, CA, USA, 2014, pp. 1-6.

[5] B. -A. Tabacaru, M. Chaari, W. Ecker, T. Kruse and C. Novello, "Fault-effect analysis on system-level hardware modeling using virtual prototypes," 2016 Forum on Specification and Design Languages (FDL), Bremen, Germany, 2016, pp. 1-7.

[6] V. Herdt, H. M. Le, D. Große and R. Drechsler, "On the application of formal fault localization to automated RTL-to-TLM fault correspondence analysis for fast and accurate VP-based error effect simulation - a case study," 2016 Forum on Specification and Design Languages (FDL), Bremen, Germany, 2016, pp. 1-8.

[7] J. Perez, M. Azkarate-askasua and A. Perez, "Codesign and Simulated Fault Injection of Safety-Critical Embedded Systems Using SystemC," 2010 European Dependable Computing Conference, Valencia, Spain, 2010, pp. 221-229.

[8] L.A.B. Naviner, J.-F. Naviner, G.G. dos Santos, E.C. Marques, N.M. Paiva, FIFA: A fault-injection–fault-analysis-based tool for reliability assessment at RTL level, Microelectronics Reliability, Volume 51, Issues 9–11, 2011, Pages 1459-1463, ISSN 0026-2714.

[9] KALITA, Limi. Socket programming. International Journal of Computer Science and Information Technologies, 2014, 5. Jg., Nr. 3, S. 4802-4807.