UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Douglas Lohmann

An Automated Non-Intrusive Fault Injection Technique Integrated into the Universal Verification Methodology

Florianópolis
2019

Douglas Lohmann

An Automated Non-Intrusive Fault Injection Technique Integrated into the Universal Verification Methodology

Dissertação submetida ao Programa de Pós-Graduação em Ciência da Computação para a obtenção do título de Mestre em Ciência da Computação.
Orientador: Prof. Dr. José Luís Almada Güntzel
Coorientador: Prof. Dr. Djones Vinicius Lettnin

Florianópolis
2019

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Lohmann, Douglas
An Automated Non-Intrusive Fault Injection Technique Integrated into the Universal Verification Methodology / Douglas Lohmann ; orientador, José Luís Almada Güntzel, coorientador, Djones Vinicius Lettnin, 2019.
88 p.

Dissertação (mestrado) - Universidade Federal de Santa Catarina, Centro Tecnológico, Programa de Pós-Graduação em Ciência da Computação, Florianópolis, 2019.

Inclui referências.

1. Ciência da Computação. 2. Injeção de falhas. 3. Metodologia Universal de Verificação. 4. SystemC. 5. Linguagem Específica para um Domínio. I. Almada Güntzel, José Luís . II. Vinicius Lettnin, Djones. III. Universidade Federal de Santa Catarina. Programa de Pós-Graduação em Ciência da Computação. IV. Título.

Douglas Lohmann

An Automated Non-Intrusive Fault Injection Technique Integrated into the Universal Verification Methodology

O presente trabalho em nível de mestrado foi avaliado e aprovado por banca examinadora composta pelos seguintes membros:

Prof^{a}. Cristina Meinhardt, Dr^{a}.
Universidade Federal de Santa Catarina

Prof^{a}. Edna Natividade da Silva Barros, Dr^{a}.
Universidade Federal de Pernambuco

Prof. Raimes Moraes, Dr.
Universidade Federal de Santa Catarina

Certificamos que esta é a **versão original e final** do trabalho de concluão que foi julgado adequado para obtenção do título de Mestre em Ciência da Computação.

Prof. José Luís Almada Güntzel, Dr.
Coordenador do Programa

Prof. José Luís Almada Güntzel, Dr.
Orientador

Florianópolis, 10 de Outubro de 2019.

Este trabalho é dedicado aos meus pais e ao meu irmão.

AGRADECIMENTOS

Gostaria de agradecer aos meus pais, Terezinha Maria Lohmann e Edison Lohmann, que fizeram e fazem o impossível para que eu alcance meus objetivos. Os esforços dos meus pais não se limitam somente a este trabalho. Não conseguiria citar aqui todas as vezes que fui acolhido, amado e que recebi conselhos que levo para minha vida. Aprendi com a simplicidade e a alegria deles o que jamais poderia aprender por livros e sei que sou muito privilegiado por isso. Não poderia deixar de agradecer ao meu irmão, Daniel Lohmann, que sempre foi exemplo e me guiou na vida acadêmica. Sem ele certamente eu não teria chegado aqui. Toda minha gratidão a essas três pessoas.

Aos professores José Luís Almada Güntzel e Djones Vinicius Lettnin, que são a base sólida desse trabalho e de todo conhecimento aqui adquirido. Ao longo deste trabalho foram inúmeros aprendizados e lições que me fizeram crescer. Em particular, agradeço ao professor Lettnin, que além de guiar esse trabalho também abriu a porta do laboratório e me deu a incrível oportunidade de estudar em uma universidade renomada. Também agradeço ao professor Güntzel que me auxiliou e dedicou seu precioso tempo para compartilhar seus conhecimentos.

Aos membros da banca que aceitaram prontamente meu convite e dedicaram tempo para contribuir com este trabalho. Todas as críticas e sugestões foram muito importantes para a versão final desta dissertação.

Não poderia deixar de agradecer aos meus amigos que me deram todo apoio e força para que eu conseguisse concluir mais esse desafio. Ao meu amigo Alexis Huf, que dedicou tempo e teve toda paciência para me ensinar. Com toda certeza, esse trabalho não teria alcançado êxito sem a sua ajuda. Também agradeço a Helena Stein, que sempre esteve ao meu lado e sempre me deu forças para terminar: você é a inspiração e o aconchego necessário nas horas difíceis.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Código de Financiamento 001.

To my mind, voyaging through wildernesses, be they full of woods or waves,
is essential to the growth and maturity of the human spirit. It is in the
wilderness that you really learn who you are.
(CALLAHAN, 2002)

RESUMO

As tecnologias de fabricação de chips contemporâneas tornam possível a fabricação de sistemas embarcados extremamente complexos compostos de hardware e software que são muito difíceis de serem verificados e testados. Como a evolução tecnológica tende a seguir, tal dificuldade deverá crescer nos próximos anos. Apesar dos notáveis avanços dos algoritmos e técnicas de Electronic Design Automation (EDA) nos últimos anos, as ferramentas de projeto e verificação não evoluíram no mesmo ritmo que a capacidade de fabricação de chips, o que deu origem aos chamados gap de projeto e gap de verificação. Neste contexto, novas técnicas que tornem possível a verificação de sistemas embarcados complexos são necessárias, de forma a reduzir o gap de verificação, aumentando assim a produtividade e reduzindo os custos. Buscando contribuir para mitigar tal situação, a indústria vem liderando uma iniciativa para padronizar os processos de verificação a qual resultou na Metodologia de Verificação Universal (UVM). A UVM define regras e diretrizes para melhorar o desenvolvimento de testbenches e a execução da simulação através do reúso de código e outras características relevantes. Ela também viabiliza a reutilização de componentes. Na última década, a linguagem SystemC tornou-se amplamente utilizada tanto para o projeto quanto para verificação de sistemas no alto nível de abstração, uma vez que tal nível permite lidar com a complexidade dos sistemas embarcados contemporâneos. Entretanto, as ferramentas de síntese de alto nível não estão suficientemente maduras para produzir descrições no nível de transferência entre registradores (RTL) com boa qualidade e por isso, a indústria ainda lança mão de conversões semiautomáticas para gerar descrições RTL em VHDL ou verilog, ao invés de usar somente SystemC ao longo de todo o fluxo. Tal procedimento causa um problema ao fluxo de verificação, uma vez que os modelos de injeção de falhas desenvolvidos para o alto nível raramente são reutilizados no RTL. Além disso, apesar do grande número de técnicas de injeção de falhas que existem, a maioria delas são difíceis de serem integradas aos testbenches e ao projeto. Dadas as limitações e dificuldades mencionadas anteriormente, este trabalho propõe uma nova técnica não-intrusiva de injeção de falhas batizada de UVM-FI, integrada à UVM. UVM-FI consiste em uma biblioteca de código livre, escrita em C++/SystemC que contém tipos básicos de falhas, gatilhos e localizações de falhas. A integração da UVM-FI com a UVM baseia-se no mecanismo de compartilhamento de dados da UVM e viabiliza o reúso. Como segunda contribuição importante, este trabalho também propõe uma Linguagem de Domínio Específico (em inglês, Domain-Specific Language - DSL) que permite a criação automática de modelos de falhas em SystemC. A adoção de SystemC permite a descrição de todos os componentes de verificação em uma mesma linguagem, incluso o próprio projeto. A capacidade do injetor de falhas proposto e sua facilidade de integração no ambiente UVM foram avaliadas e comparadas para diversos cenários distintos. Desta forma, este trabalho contribui para o avanço da área de verificação por meio da criação de ambientes de verificação reutilizáveis, capazes de levar a cabo a injeção de falhas.

Palavras-chave: Injeção de falhas. Metodologia Universal de Verificação. SystemC. Linguagem Específica para um Domínio.

RESUMO ESTENDIDO

Introdução

A simulação de falhas consiste em injetar falhas no sistema e, em seguida, exercitá-lo para observar o impacto gerado pelas falhas. A técnica de injeção de falhas pode ser aplicada ao *hardware* e ao *software* para medir o desempenho do sistema e validar mecanismos de tolerância a falhas.
A complexidade dos sistemas tem aumentado constantemente e nesse contexto, metodologias para criar testes e técnicas para melhorar a confiabilidade dos sistemas foram propostas para otimizar ou até tornar viável o processo de verificação. A Metodologia de Verificação Universal (UVM) surgiu nesse contexto, como um esforço de padronização conduzido pelo consórcio Accellera [1].
Entretanto, integrar modelos de falhas à UVM não é uma tarefa fácil. Como a injeção de falhas em si não é padronizada, sua integração deve ser feita manualmente, o que resulta em custos extras. Portanto, um novo método para automatizar os modelos de injeção de falhas, facilitando sua integração ao UVM, poderia melhorar todo o processo de verificação.

Objetivos

O principal objetivo deste trabalho é apresentar, validar e avaliar uma nova técnica de injeção de falhas que permita a fácil integração de modelos de injeção de falhas na UVM e que também facilite a reutilização de componentes em um ambiente de testes. Como objetivo secundário, este trabalho procura aprimorar a descrição do modelo de falhas em termos de facilidade de uso e expressividade.

Contribuições

Os objetivos foram alcançados com o desenvolvimento de uma extensão da UVM com novos componentes que permitam a injeção de falhas de forma não intrusiva, sendo essa nova técnica batizada de UVM-FI. Além de estender a UVM disponibilizada pela Accellera, a UVM-FI foi implementada em SystemC e ao contrário da maioria dos trabalhos correlatos, seu código é aberto e pode ser baixado em https://gitlab.com/lohmann/uvm-fi.
As principais contribuições deste trabalho são:

- Uma nova técnica, escrita em SystemC, não intrusiva de injeção de falhas que permitem a reutilização de componentes para criar testes de injeção de falhas com a UVM ;
- Uma biblioteca de modelos de falha que podem ser integrados a um ambiente UVM usando o princípio plug-and-play;
- Uma Linguagem de Domínio Específico (em inglês, Domain-Specific Language - DSL) metaprogramável implementada em SystemC combinada com a UVM para descrever modelos formais de falhas.

Metodologia

1. Condução de uma Revisão Sistemática da Literatura com escopo em técnicas de injeção de falhas;
2. Análise das principais técnicas de injeção de falha em SystemC e C++;

[1] http://accellera.org/

3. Análise das ferramentas e técnicas disponíveis para permitir a introspecção em código na liguagem SystemC;

4. Implementação da biblioteca UVM-FI;

5. desenvolvimento da DSL;

6. Desenvolvimento de experimentos de injeção de falhas utilizando a UVM-FI;

7. Avaliação e validação da UVM-FI;

8. Avaliação da capacidade de expressividade da descrição dos modelos de falhas utilizando a DSL;

9. Desenvolvimento de estudos de caso para aplicação da UVM-FI em sistemas mais complexos e mais semelhantes aos sistemas reais;

10. Avaliação dos resultados obtidos através de estudos de caso.

Proposta

A técnica proposta, chamada de UVM-FI, agrega capacidades de injeção de falhas à UVM. Tal integração evita a reimplementação desnecessária de componentes que são equivalentes entre as duas técnicas (como monitores, geradores de sequencias e analisadores de dados) e permite a análise de confiabilidade mais cedo na fase de verificação.

O ambiente de injeção de falha se baseia na UVM, adicionando um mecanismo ECA (Event-Condition-Action) e uma DSL (Domain-Specific Language) para descrição do modelo de falha. Além disso, a UVM-FI fornece uma interface para conectar-se ao DUT via recursos já disponibilizados pela UVM.

A entrada para a UVM-FI é uma descrição do modelo de falha. Com base na especificação de falha desejada, o mecanismo (ECA) gerenciará as ações e os tipos de falha registrados para executar a injeção de falha no momento pretendido.

Resultados e Discussão

A proposta foi avaliada por meio de experimentos controlados para validar a injeção de falhas e analisar o modelo de descrição de falhas utilizando a DSL. Além disso, foram desenvolvidos cenários envolvendo sistemas mais complexos e reais. Primeiramente, é apresentado um exemplo de injeção de falha em um contador de 8 bits, o qual foi comparado com trabalhos correlatos. Uma vez validado o exemplo e avaliado seu comportamento, constatou-se que o injetor foi capaz de injetar falhas em modelos SystemC conforme o esperado. Em seguida, foi realizada uma avaliação de injeção de falha em termos de expressividade da descrição do modelo de falha para três cenários diferentes. Por fim, a UVM-FI foi aplicada para injetar falhas em dois cenários; um multiplicador de matriz com mecanismo tolerante a falhas e a uma plataforma virtual com simulador de conjunto de instruções (ISS) do microcontrolador MSP430 com modelo de memória transacional (em inglês Transaction-Model Modeling - TLM). Nos dois cenários, foi possível aplicar a técnica proposta com sucesso e validar a versatilidade e capacidade da UVM-FI.

A UVM-FI não requer nenhuma alteração no DUT original. Além disso, não é necessário sobrescrever ou estender o código-fonte SystemC, pois a UVM-FI é uma técnica completamente não intrusiva. O modelo de falha é descrito apenas no ambiente de teste e muitos locais, eventos e tipos de falhas podem ser desenvolvidos e executados com eficiência no DUT.

Considerações Finais

Este trabalho de mestrado, propôs, validou e avaliou uma nova técnica de injeção de falha não intrusiva e totalmente integrada à UVM, essa técnica foi batizada de UVM-FI.

UVM-FI consiste em uma biblioteca de código-fonte aberto, escrita em C++/SystemC, contendo tipos, eventos e locais de falhas que podem ser integrados a um ambiente UVM usando o princípio plug-and-play. Os experimentos e estudos de caso validaram esse novo injetor e demonstraram sua capacidade, permitindo uma comparação com outras abordagens.

A descrição dos modelos de falhas na UVM-FI é feita utilizando SystemC DSL. Os estudos de caso desenvolvidos neste trabalho demonstraram que a DSL é menos verbosa e mais legível do que as outras abordagens baseadas em XML.

Palavras-chave: Injeção de falhas. Metodologia Universal de Verificação. SystemC. Linguagem Específica para um Domínio.

ABSTRACT

Current chip fabrication technologies render possible the implementation of extremely complex embedded systems composed of hardware and software that are very difficult to verify and test. As long as technology evolution tends to continue, such difficulty will continue to grow in the next years. Despite the remarkable advances in Electronic Design Automation (EDA) algorithms and techniques in the past years, design and verification tools have not evolved in the same pace as chip fabrication capability, giving rise to the so-called design and verification gaps. In this context, new techniques to make possible the verification of complex embedded systems are demanded, so as to reduce the verification gap and thus increase productivity and reduce costs. Looking to mitigate this situation, industry has been leading an initiative to standardize the verification procedure that resulted in the Universal Verification Methodology (UVM) which sets rules and guidelines for enhancing testbench development and simulation execution through code reusability and other relevant features, and allows for component reuse as well. In the last decade, SystemC language has become widely used for design and verification at the high levels of abstraction, since such level allows to handle the complexity of current embedded systems. However, high-level synthesis tools are not mature enough to produce good quality Register-Transfer Level (RTL) descriptions and thus industry still relies on semi-automated conversion to generate RTL descriptions in VHDL or Verilog instead of using SystemC all along the design flow. Such procedure poses a problem in the verification flow, since fault injection models developed for the high level are seldom reused in the RTL. Moreover, despite the great number of existing fault injection techniques, most of them are challenging to integrate with the testbench and with the design. Given the previously mentioned limitations and difficulties, this work proposes a novel non-intrusive fault injection technique named UVM-FI that is fully integrated into UVM. UVM-FI consists of an open-source library, written in C++/SystemC, containing basic fault types, triggers and fault locations. The integration of UVM-FI into UVM relies on the UVM data share mechanism and enables reusability. As a second important contribution, this work also proposes a metaprogrammed Domain-Specific Language (DSL) to allow for automatic creation of fault models in SystemC. The adoption of SystemC allows for describing all verification components in the same language including the design itself. The capacity of the proposed fault injector and its ease of integration into the UVM environments were evaluated and compared in several distinct scenarios. Therefore, this work contributes to the advance in the verification area by creating reusable verification environments capable of performing fault injection.

Keywords: Fault injection. Universal Verification Methodology. SystemC. Domain Specific Language.

LIST OF FIGURES

Figure 1 – Design and verification gap comparison. 29
Figure 2 – Fault Injection Environment Architecture. 36
Figure 3 – UVM Testbench architecture. 37
Figure 4 – SystemC language architecture. 38
Figure 5 – Fault injection structure with centralized controller for FIMs. 41
Figure 6 – Fault injection structure with FIM. 42
Figure 7 – Proposed layered architecture for UVM-FI. 48
Figure 8 – SystemC project with UVM and UVM-FI. 49
Figure 9 – UVM-FI fault injection components. 49
Figure 10 – Sequence diagram for an example of fault injection flow. 50
Figure 11 – Example of DSL expression and corresponding AST. 51
Figure 12 – UVM-FI Testbench architecture. 53
Figure 13 – 8-bit counter DUT. 57
Figure 14 – 8-Bit counter waveform diagram results with UVM simulation. 61
Figure 15 – 8-Bit counter wave diagram results with UVM-FI simulation. 62
Figure 16 – Matrix multiplier DUT. 66
Figure 17 – TMR matrix multiplier DUT. 67
Figure 18 – MSP430 memory map. 70

LIST OF LISTINGS

1 UVM fault environment . . . 54
2 Store object in UVM data share . . . 54
3 Restore objects from UVM data share . . . 55
4 8-Bit counter SystemC implementation. . . . 58
5 8-Bit counter with mutants. . . . 59
6 8-Bit counter with simulator command. . . . 60
7 8-Bit counter with simulator command. . . . 61
8 Registering a fault condition for 8-bit counter. . . . 61
9 Scenario A: XML fault model. . . . 63
10 Scenario A: fault model without DSL . . . 63
11 Scenario A: fault model with the proposed DSL . . . 63
12 Scenario B (probabilistic): DSL description . . . 64
13 Scenario C (Moving average): DSL description . . . 64
14 Registering fault model resources. . . . 66
15 Registering a fault condition . . . 66
16 Registering a fault condition . . . 68
17 Registering a fault condition for bit-flip scenario . . . 72
18 Registering a fault condition for coupling faults . . . 72

LIST OF TABLES

Table 1 – Existing fault injection techniques and our proposal. 46
Table 2 – Character count per Technique per Scenario 65
Table 3 – Fault injection results for a simple matrix multiplier. 67
Table 4 – Fault model specification. 68
Table 5 – Fault injection results 68
Table 6 – MSP430F249 Memory Organization. 71
Table 7 – AES algorithm results. 72
Table 8 – CRC algorithm results. 73
Table 9 – UVM and UVM-FI simulation time for coupling fault execution. 73
Table 10 – UVM-FI SystemC DSL operators. 87
Table 11 – UVM-FI helper functions provided by DSL. 87

CONTENTS

1 INTRODUCTION 29
1.1 MOTIVATION 31
1.2 GOALS 32
1.3 CONTRIBUTIONS 32
1.4 ORGANIZATION 32

2 BACKGROUND 33
2.1 FAILURE, ERROR, AND FAULT 33
2.2 FAULT INJECTION 34
2.3 UNIVERSAL VERIFICATION METHODOLOGY 36
2.4 SYSTEMC 38
2.5 DOMAIN-SPECIFIC LANGUAGES 39

3 RELATED WORK 41
3.1 FAULT INJECTION STRATEGIES 41
3.2 SUMMARY OF RELATED WORK 44

4 THE PROPOSED FAULT INJECTION TECHNIQUE AND ITS INTEGRATION INTO THE UVM 47
4.1 UVM-FI 47
4.2 A DOMAIN-SPECIFIC LANGUAGE TO AUTOMATE FAULT INJECTION IN SYSTEMC MODELS 51
4.3 INTEGRATION INTO THE UNIVERSAL VERIFICATION METHODOLOGY 52
4.4 UVM-FI CONCLUDING REMARKS 55

5 EVALUATION OF THE PROPOSED FAULT-INJECTION TECHNIQUE 57
5.1 UVM-FI VALIDATION 57
5.2 EVALUATION OF THE EXPRESSIVENESS AND EASE OF USE: THE IMPACT OF THE PROPOSED DSL 62
5.3 FAULT INJECTION SCENARIOS 65
5.3.1 Matrix Multiplication 65
5.3.2 Virtual Platform Target Application 69

6 CONCLUSION 75
6.1 SUMMARY OF THE RESEARCH ACHIEVEMENTS 75
6.2 FUTURE WORK 76

BIBLIOGRAPHY **79**

APPENDIX **83**

APPENDIX A – PUBLICATIONS **85**

A.1 PUBLISHED PAPERS DIRECTLY RELATED TO THE DISSERTATION SUBJECT 85

A.1.1 ***2018 IEEE 9th Latin American Symposium on Circuits & Systems (LAS-CAS)*** **85**

A.1.2 ***2018 25th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*** **85**

APPENDIX B – DSL **87**

LIST OF ABBREVIATIONS AND ACRONYMS

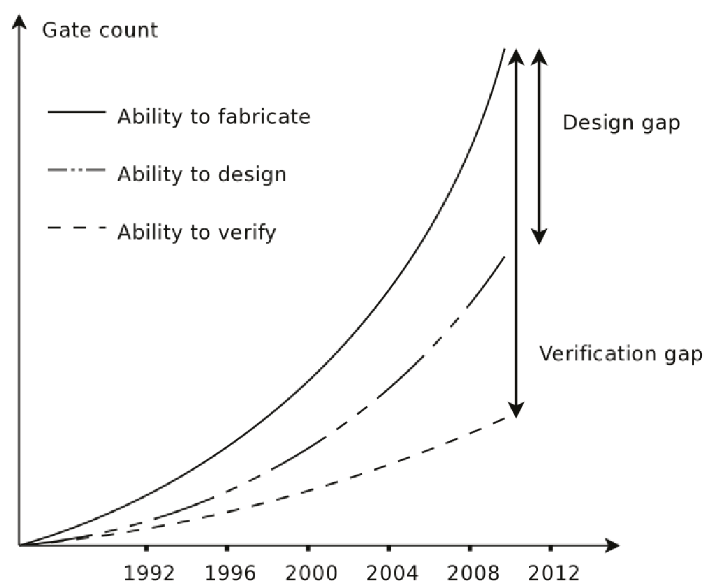
AST	Abstract Syntax Tree
DSL	Domain-Specific Language
DUT	Design Under Test
DUV	Design Under Verification
ECA	Event-Condition-Action
EDA	Electronic Design Automation
ESL	Electronic System Level
FI	Fault Injection
FICU	Fault Injection Control Unit
FIM	Fault Injection Module
HDL	Hardware Description Language
ISS	Instruction Set Simulator
RTL	Register-Transfer Level
STL	Standard Template Library
SCV	SystemC Verification library
TLM	Transaction-level Modeling
UVM-FI	Universal Verification Methodology with Fault Injection technique
UVM	Universal Verification Methodology
UVC	UVM Verification Component
VERIFY	VHDL-based Evaluation of Reliability by Injection Faults Efficiently

1 INTRODUCTION

Current chip fabrication technologies allow the implementation of extremely complex embedded systems composed of hardware and software that are very difficult to verify and test. As long as technology does not cease to evolve, the complexity of embedded systems verification and testing will continue to increase not only due to the increase in the number of components but also because the likelihood of failures will eventually grow. Verification is the task of checking the correct behavior of the system. It deserves much attention since a failure that is not detected during design may cause a product recall resulting in financial losses (GAJSKI et al., 2009). In addition, verification is crucial in the case of critical systems such as health-care equipment and autonomous vehicles where a single failure can be life-threatening. Therefore, the need for reliability and the increasing complexity of embedded systems bring new challenges to both design and verification flows.

Although current fabrication technologies may provide chips with billions of transistors, Electronic Design Automation (EDA) tools are not able to tackle designs that fully exploit such fabrication capability (GROSSE; DRECHSLER, 2010). However, this difficulty is not new. Historically, design tools have failed to ensure the design of systems with the maximum number of transistors that fabrication technologies could allow, a flaw that has been referred to as the **design gap**. In addition, for any given technology node the verification effort is always greater than the design effort and consequently, verification tools are still more limited with regards to the complexity of systems what gives rise to the so-called **verification gap**. Figure 1 depicts the evolution of both design and verification gaps along the past years. In that one can observe that the ability to verify systems is evolving much more slowly than the ability to fabricate them.

Figure 1 – Design and verification gap comparison.



Source: Adapted from (GROSSE; DRECHSLER, 2010).

In order to deal with the increasing complexity of electronic systems, engineers and researchers from industry and academia have devised new levels of abstraction for design representation and behavior modeling, being the Electronic System Level (ESL) (RIGO; AZEVEDO; SANTOS, 2011) (MARTIN; BAILEY; PIZIALI, 2007) the highest one. At the ESL it is possible to model complex systems using a high level language, such as C++ and Java. However, to better integrate the ESL with the Register-Transfer Level (RTL), paving the way for high-level synthesis, specific languages to model both hardware and software in a more appropriate manner have been developed. The most relevant ones are SystemVerilog and SystemC.

Created and standardized by the IEEE (ASSOCIATION et al., 2012), SystemC is a system design and modeling language that extends the C++ language with a set of macros and class libraries, making possible hardware and software modeling with the same language. With SystemC, engineers can develop and verify complex systems composed of both hardware and software possibly at different levels of abstraction. However, it has been used mainly at the high levels of abstraction because high-level synthesis tools are not mature enough to provide RTL descriptions with a quality equivalent to that an experienced designer can produce. Due to that, so far the industry still relies on a semi-automated conversion to generate RTL descriptions in VHDL or Verilog from SystemC or SystemVerilog, which increases the risk of introducing new errors. In addition, the gap between high-level and RTL renders verification extremely difficult. Thus, despite the potential offered by the language, currently SystemC-based verification is not performed all along with the design steps and generally, the coverage and test are checked manually (GROSSE; DRECHSLER, 2010).

The number of faults and the difficulty in verifying a system correlate with its complexity. Indeed, complex systems tend to have more faults and less observability and controllability for verification. Therefore, design and verification methodologies that cover all design steps are needed to increase the quality of the system. Particularly, to increase reliability the system behavior must be exercised in the presence of faults, and not only in expected cases. This is accomplished by fault simulation.

Fault simulation consists in injecting faults in the system and then exercising it to observe the impact of faults on its behavior. Faults are described in a fault model, containing the fault type, fault trigger, and the fault location. The fault injection technique can be applied to both hardware and software to measure the system performance and to validate fault-tolerant mechanisms. It can be performed by simulating a system model, by exercising a system prototype or even by executing directly in the real system (BENSO; PRINETTO, 2003). The environment to perform system verification and fault simulation is named Testbench. A Testbench is composed of a stimulus generator, a monitor, a scoreboard, and the Design Under Test (DUT) (BERGERON, 2003).

While the complexity of systems has steadly increased, methodologies to create Testbenches and techniques to improve the dependability of systems have been proposed to optimize or even to make the verification process viable. The Universal Verification Methodology (UVM) has emerged in this context as a standardization effort conducted by the consortium Ac-

cellera [1]. UVM sets rules and guidelines for enhancing testbench development and simulation execution through code reusability, modular Testbenches, stimuli generation and transaction-level communication between the verification environment and the DUT, among other features (ROSENBERG; KATHLEEN, 2013). Lately, UVM is being widely adopted by industry.

1.1 MOTIVATION

In the last years, several fault injection techniques have been proposed, each one devoted to different types of faults at different levels of abstraction and making use of different languages (SHAFIK; ROSINGER; AL-HASHIMI, 2008), (BELTRAME et al., 2008), (LU; RADETZKI, 2013). Although those techniques are already used and validated in many projects, they are not integrated with UVM or with any other standard methodologies. In addition, the majority of them have complex frameworks that hamper their integration into other verification methodologies to build efficient Testbenches.

As already mentioned, SystemC offers excellent potential for both design and verification of complex systems and for this reason it is widely used. Nonetheless, the way fault models are described in existing SystemC-based fault injection tools raises some issues. Many fault injection tools are based on the Extensible Markup Language (XML) or use a run-time console. The use of manual interactions through consoles precludes tests from running in batch, making it difficult to replicate experiments. The use of XML makes it difficult to setup system verification under fault injection because XML description of fault models is verbose. Moreover, the XML format is quite distant from SystemC. In addition, by using XML each tool requires a specific structure to describe fault models which must be learned by the designer. Application Programming Interface (API)-based approaches could avoid the limitations of XML but still yield complex setup code that is difficult to maintain.

Before the efforts to standardize verification methodologies promoted by Accellera, different methods for verification were created to perform similar tasks, which resulted in extra costs to retrain designers and convert codes. The adoption of UVM can reduce such problem, allowing verification based on reusable Testbenches and components. However, the integration of fault injection models into UVM is not an easy task. As fault injection itself is not standardized, its integration must be done manually, which results in extra costs. **Therefore, a new method to automate fault injection models making their integration into UVM easier could improve the whole verification process and thus is highly desirable.** It is also desirable that all components of such method are built with the same language used to model the system under verification, which in the context of UVM may be either SystemC or SystemVerilog. This way, designers will not have to deal with different languages and different syntaxes.

[1] http://accellera.org/

1.2 GOALS

Given the limitations of existing approaches described in the previous section, the main goal of this work is to propose, validate, and evaluate **a new fault injection technique that allows the easy integration of fault injection models into UVM and that is able to provide reusability**. As secondary goal, this work looks for enhancing fault model description in terms of ease of usage and expressivity.

1.3 CONTRIBUTIONS

The two goals proposed for this master degree work were met. In particular, the main goal was met by extending the Accellera UVM standard with a new non-intrusive fault injection technique for the high levels of abstraction, giving rise to a novel technique named **UVM-FI**. Besides relying on Accellera UVM, UVM-FI was implemented in SystemC and unlike most of the state-of-the-art correlate works, its code is open and can be downloaded from https://gitlab.com/lohmann/uvm-fi.

The main contributions of this work are:

- A new non-intrusive fault injection technique written in SystemC that allows reusing of components to create fault injection campaigns in a similar fashion as UVM allows for reusing verification components to build Testbenches.
- An easily extensible library of fault models that can be integrated into an UVM environment using the plug-and-play principle. The library includes basic fault types, triggers and fault locations.
- A metaprogrammed Domain Specific Language (DSL) in SystemC to be combined with the UVM to describe formal fault models without the need for specific compilers or preprocessed code. Unlike current fault injection techniques, there is no need to create fault injection environment manually or to describe the system in the XML format. Moreover, the use of a DSL enhances the capability of describing more sophisticated fault injection tests by providing a better way to describe the fault model.

1.4 ORGANIZATION

The remainder of this text is organized as follows. Chapter 2 presents the main concepts about libraries and techniques used in this dissertation. Chapter 3 discusses the related work, the main fault injection techniques already proposed and their capabilities. Chapter 4 details the contributions of this work which are a new fault injection technique, the DSL improvements and the facilities to integrate fault injection with UVM. Chapter 5 discusses the results. Finally, Chapter 6 draws the conclusions and proposes some future works.

2 BACKGROUND

This chapter describes the basic concepts and tools related to this research. Section 2.1 presents the definition of failure, error, and fault for system verification and introduces the concept of dependability as well. Section 2.2 describes the basics of fault injection, its concepts and techniques. Section 2.3 shows the fundamentals of Universal Verification Methodology (UVM). Section 2.4 briefly describes the main concepts of SystemC, the language that encompasses all this work. Section 2.5 presents a brief discussion of Domain-Specific Languages (DSLs).

2.1 FAILURE, ERROR, AND FAULT

A system *failure* occurs when the system does not perform as specified. A failure is caused by an error, where an *error* is a fault manifestation that causes the system to enter an incorrect state. The causes of an error may be an imperfection or a defect within some hardware or software component, which is called a *fault* (AVIZIENIS et al., 2004). The presence of a fault in a system may not generate a failure. This happens if the system execution does not activate the fault. The fault, error, and failure are called the factors or the threats to dependability.

Faults in a system can occur either by hardware/physical defects or software incorrect design. Hardware or physical faults can be grouped by their duration as permanent, transient, or intermittent faults. Permanent faults are those caused by irreversible component damages. Transient faults are caused by the environmental conditions such as the electromagnetic interference or radiation effects. Intermittent faults are caused by the unstable hardware or varying hardware states (BENSO; PRINETTO, 2003). Software faults are a consequence of a bad design and can be introduced in all phases of the software development process. According to Benso & Prinetto (2003), software faults can be categorized as follow:

- *Function faults:* Require a design change to correct the fault due to an incorrect or missing implementation;
- *Algorithm faults:* Incorrect or missing implementation that can be fixed without design modifications;
- *Timing/serialization faults:* Absence or incorrect serialization of shared resources. Can be solved by implementing a mechanism to protect the data, like mutexes or semaphores;
- *Checking faults:* Incorrect data validation and incorrect conditional statements;
- *Assignment faults:* Values assigned incorrectly or not assigned.

Fault models are descriptions of the fault attributes. A fault model contains the fault location, the time that the fault should be injected (also called trigger) and the type of the injected error. The location of a fault is where the fault should be introduced (e.g. memory space or interconnections). Timing describes when the fault should be activated in the system,

this attribute represents the duration and persistence of a fault. Type of perturbation is the type of fault. Some examples of physical faults are open transistors, shorting lines that are physically close to each other, and broken wires (BUSHNELL; AGRAWAL, 2004). Usually, software faults are corrupted instructions or wrong operations in the code. The fault model description depends on the set of faults supported by the fault injection tool and the means used to implement them.

Dependability is defined as the ability to deliver service that can justifiably be trusted (LAPRIE, 1992). However, it is necessary to define a criterion for deciding if the service is dependable. In this context, another definition of dependability is the ability to be tolerant for the most frequent and critical failures for the acceptable system behavior (AVIZIENIS et al., 2004).

Specifically, dependability of a system can be defined as a set of attributes that assess the system safety. According to Avizienis et al. (2004) dependability is defined as the set of following attributes:

- *Availability:* Readiness for correct service.
- *Reliability:* Continuity of correct service.
- *Safety:* Absence of catastrophic consequences on the user(s) and the environment.
- *Integrity:* Absence of improper system alterations.
- *Maintainability:* Ability to undergo modifications and repairs.
- *Confidentiality:* Absence of unauthorized disclosure of information.

Many techniques have been developed over the last years to increase system dependability. The main means for achieving dependability are Fault Prevention, Fault Tolerance, Fault Removal, and Fault Forecasting. Fault Prevention aims to avoid the presence of a fault into the system or prevent that fault to occur. Fault Tolerance mechanisms prevent a system failure: even if a fault does occur, the system will continue behaving correctly. Fault removal consists of methods to reduce the presence of faults, at least the most usual faults in a system. Fault forecasting provides the means of tracking the number and the consequences of faults (AVIZIENIS et al., 2004).

2.2 FAULT INJECTION

To increase the system's dependability, validation of fault injection techniques can be used. The purpose of fault injection is to introduce artificial faults into systems and observe their behavior under those faults. This technique can be used for both hardware and software to measure the system tolerance to failures particularly, when fault tolerant mechanisms are used in the design. It can be performed on either simulations models, prototypes, or directly at the real system. (BENSO; PRINETTO, 2003).

Fault injection techniques can be categorized by how much they modify the original system to introduce a fault. Non-invasive approaches are transparent to the system under test; invasive techniques require modifications of the original system, by introducing more code or changing some components. The non-invasive approaches are desired because changes in the original system usually require a lot of effort, and manually work can modify the system behavior.

Fault injection can also be implemented and categorized by whether they are implemented (BENSO; PRINETTO, 2003), as follows:

- *Hardware-implemented fault-injection:* A special hardware to allow the injection of faults is created and connected to the model under test;
- *Software-implemented fault-injection:* Additional code/software is created to manipulate objects of the original model;
- *Simulation-implemented fault-injection:* Faults are injected to the model by a simulator or debugger;
- *Emulation-fault injection:* Performed by a simulation based fault injection system in FPGAs to emulate the circuit behaviour at hardware speed;
- *Hybrid fault injection:* Combine software-implemented fault injection and hardware monitoring.

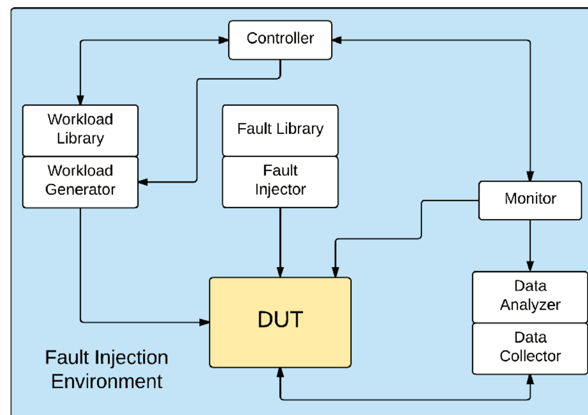
The simulation-based fault injection do not operate at the physical hardware; the system is simulated using a simulation language like VHDL or SystemC. The faults are deliberately injected to the model and its behavior is observed. For instance, a virtual platform developed to simulate an autonomous vehicle test what happens when a real memory fault occurs.

Usually, the main strategies for implementing simulated fault injection are based on mutants, saboteurs or simulator command. The mutant technique replaces a component by another one to enable faults. Such component works like the original one; when activated, the mutant introduces faults. The saboteurs technique introduces a new component in the model. The component can be inactive like the mutant. When activated, the saboteur changes the data or time properties of a signal. The simulator command technique issues commands to control fault injection time, replacing the variables and signals values. The drawbacks of saboteurs and mutants are that both require modifications of the design and introduce overhead in simulations. The main drawback of simulator command technique is the need of simulator that supports such commands. Fault injection environments, as defined in (HSUEH; TSAI; IYER, 1997), are those where a fault injector can be used to introduce faults according to multiple fault models into a design that is under evaluation in a controlled testbench. Figure 2 illustrates the basic components of such environment. It usually consists of the following components:

- *Fault injector:* Inject fault into the target system and execute commands from the workload generator;

- *Fault library:* Library containing different fault types, fault location, fault time, and appropriate hardware semantics or software structure;
- *Workload generator:* Generate the inputs for the target system;
- *Workload library:* Store the different types of inputs (workloads) for the target system;
- *Controller:* Control the experiment execution;
- *Monitor:* Supervise the execution of commands and initiate data collection whenever necessary;
- *Data Collector:* Perform the online data collection;
- *Data analyzer:* Process and analyze the data.

Figure 2 – Fault Injection Environment Architecture.



Source: The author, adapted from (HSUEH; TSAI; IYER, 1997)

2.3 UNIVERSAL VERIFICATION METHODOLOGY

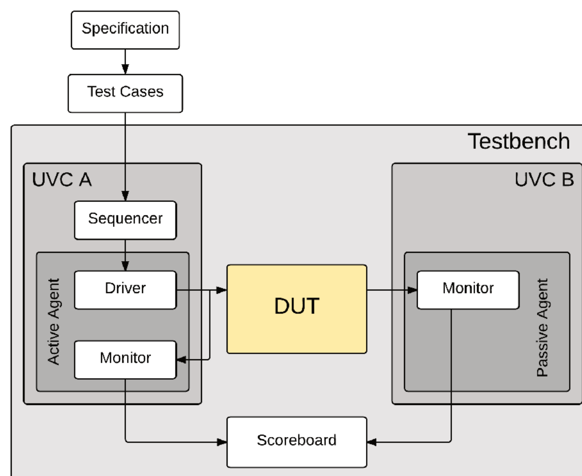
UVM is a verification methodology which sets rules and guidelines for enhancing testbench development and simulation execution through code reusability, modular testbenches, stimulus generation and transaction-level communication between the test environment and the Design Under Test (DUT), among other qualities (ROSENBERG; KATHLEEN, 2013). It was originally implemented for the e Verification Language developed by Verisity Design (now Cadence Design Systems) (ROSENBERG; KATHLEEN, 2013). Later, the UVM was ported to SystemVerilog HDL, and recently released for SystemC as a public review release.

UVM's reusability is based on its modular testbench architecture, in which its basic modules, such as drivers and monitors, are integrated in a larger structure called UVM Verification Component (UVC). These UVCs follow a plug-and-play approach, where the same interface in two different DUTs can use the exact same UVC, reducing time spent on testbench development. Sequences hierarchically create input data for the DUT, based on constraints.

These *sequences* can be reused between different models without any need for configuration or code modification.

The basic architecture of an UVM testbench is shown in Figure 3. Based on the verification documentation and specification, Test cases to be simulated are fed to the testbench. The UVC *sequencer* creates *sequences*, and passes them to a *driver*, which organizes the data and sends them to the DUT. Meanwhile, a *monitor* collects the data sent by the *driver* and forwards to the *scoreboard* for checking. Another UVC is responsible for collecting the output data through a *monitor*, and feeding the same *scoreboard*, which will also collect the functional coverage.

Figure 3 – UVM Testbench architecture.



Source: The author, adapted from (ROSENBERG; KATHLEEN, 2013)

Another important issue is the definition of Design Under Test (DUT) and Design Under Verification (DUV). Both are used in the UVM context depending on the design phase. If we are performing the UVM in order to check if the product meets the specification, we are performing the verification phase and the correct terminology is DUV. When the design is checked for defects to make sure the device is working, the correct terminology is DUT. Although they can be both applied to this work, we will always use DUT in our illustrations assuming that we want to perform the tests in the design.

Although the concept of UVM is independent of the programming language, there are some standards to develop a structured verification environments following the UVM for specific languages such as VHDL and SystemVerilog. Accellera Systems Initiative [1] developed a standard UVM environment fully compatible with the Accellera UVM standard in SystemC, as well as SystemVerilog that provide the main UVM components and structure to facilitate the development and verification with UVM.

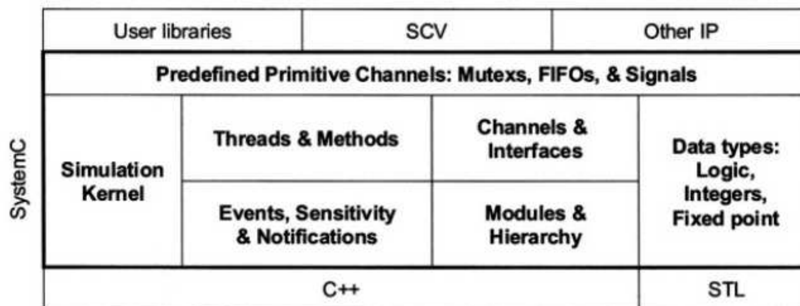
[1] http://accellera.org/

2.4 SYSTEMC

SystemC is a system design and modeling language built up from a set of macros and class libraries extended from the C++ language to enable both software and hardware modeling. Since SystemC is based on C++ language and the Standard Template Library (STL) framework the software modeling is an intrinsic part and the notability of SystemC is the addition of hardware concepts, such as time model, hierarchy and structure, and hardware data types.

Figure 4 illustrates the main SystemC components. All the language execution is based on an event-driven simulation kernel that performs the processes simulation. An process is implemented as Methods or Threads instances. Moreover, the framework offers mechanisms to control the simulation execution, by events and notifications, communication between processes, hierarchy structure organization, and data types interfaces and extensions.

Figure 4 – SystemC language architecture.



Source: The author, adapted from (BLACK et al., 2009)

Channels and Interfaces are provided for communication between concurrent processes using events. For example, the class *sc_signal* is a primitive channel to model the behaviour of single wires carrying a digital electronic signal between modules.

Usually, a hardware design is composed by hierarchical blocks, like the *entity/architecture* components in VHDL. In SystemC, blocks are components encapsulated as modules, and they are represented by the *sc_module* base class. A module may contain processes, communication channels, and other modules.

An important concept of SystemC that surrounds this work is the implementation of concurrency by the simulator. SystemC is an event-driven simulator that uses simulation process to model concurrency; as consequence, the concurrency is simulated and not real concurrent executed. Since SystemC is an event-driven simulator, the events are important to control the kernel simulation execution. The *sc_event* class is the base representation of an event in SystemC. By definition, an SystemC *event* is the occurrence of an *sc_event* notification that happens at single instant in time.

When the SystemC is coupled with verification libraries (such as the SystemC Verification library (SCV)), it provides mighty features for systems verification. Besides the verification, many other innovations are in development to increase the powerful of SystemC for

systems design and verification, like the SystemC Universal Verification Methodology (UVM). The SystemC, SCV and the SystemC-UVM are developed and maintained by Accellera Systems Initiative (Accellera) [2]

2.5 DOMAIN-SPECIFIC LANGUAGES

Domain-Specific Languages (DSL) are programming languages designed for usage in a specific domain (HUDAK, 1998) such as parsing, document and media authoring or hardware description. Having a specific and well-defined domain allows such languages to incorporate powerful and versatile abstractions that ultimately increase programmer productivity and decrease the effort required for debugging and maintenance. A particular class of DSLs are Domain-Specific Embedded Languages which reuse capabilities of an existing general-purpose programming language in order to create an abstraction layer with its own syntax and semantics, which can be seamlessly integrated with code written in the host programming language.

C++, the language that SystemC extends, contains two mechanisms that can be used for meta-programming (i.e., code that is able to generate code): the C preprocessor and templates. Unlike the C preprocessor, template-based meta-programming is Turing-completeness (VELDHUIZEN, 2003), a feature that opens several possibilities of application, including the design of embedded DSLs by overriding C++ operators (ABRAHAMS; GURTOVOY, 2004). Examples are the Boost.Range[3] adaptors. With operator overloading and template meta-programming, an expression such as `copy(vec | filtered(is_person(_1) | reversed)` is correctly processed by the C++ compiler and has the semantics of iterating `vec` discarding any object for which the function `is_person` returns `false`, in reverse order. In contrast, the plain C++ code using standard library containers would require a loop with an inner `if`, amounting to at least three lines of code.

[2] www.accellera.org
[3] http://www.boost.org/doc/libs/1_66_0/libs/range/doc/html/index.html

3 RELATED WORK

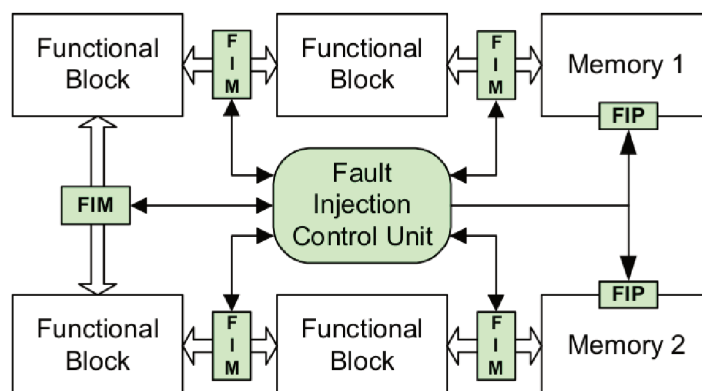
Chapter 2 presented the background on fault injection and other fundamentals related to this proposal. This Chapter discusses the related work for injecting faults, mainly in SystemC simulated models, and then, summarizes and compares the related work with this dissertation proposal.

3.1 FAULT INJECTION STRATEGIES

Fault injection is a popular technique for systems reliability at any design level. Usually, existing fault injection (FI) tools do not focus on HDL languages, and most of them are limited to RTL level (GRACIA et al., 2001). Fault injection techniques are being ported to SystemC models as developers are adopting more levels of abstraction and using SystemC for that.

Rothbart et al. (2004) presented a fault injection module (FIM) to simulate attacks in Smart Cards. The FIMs are components with fault injection capabilities that are connected between modules into the SystemC design. Figure 5 shows the (ROTHBART et al., 2004) approach, in which a fault injection control unit (FICU) is used to control the FIMs. The fault injection control unit manages each FIM that is responsible to inject a specific fault in a functional block or in memory.

Figure 5 – Fault injection structure with centralized controller for FIMs.

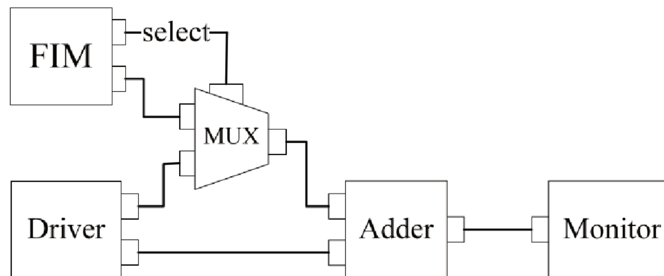


Source: Adapted from (ROTHBART et al., 2004)

Chang & Chen (2007) showed a framework for fault injection at the bus-cycle accurate level, untimed and timed functional transaction-level modeling. Their approach is based on the work presented in (ROTHBART et al., 2004). The authors in (CHANG; CHEN, 2007) show a distributed control units instead of centralized controller to manage the FIMs. The insertion method of fault is the same as (ROTHBART et al., 2004), into the interconnections of the functional blocks (FIM). Figure 6 shows a block diagram of an adder with the FIM to decide when activating a fault injection. The FIM activates the fault injection by setting the desired

fault in the "MUX" connected to the input port of the adder. A design can have one or more FIMs depending on the desired fault model. A central unit module is responsible for controlling the FIMs.

Figure 6 – Fault injection structure with FIM.



Source: Adapted from (CHANG; CHEN, 2007)

Both approaches (ROTHBART et al., 2004; CHANG; CHEN, 2007), are highly intrusive because it is necessary to connect the framework for fault injection inside the SystemC design. Besides that, the techniques are limited for block interconnection and require manual effort in re-design to insert the desired fault for each component. As the system and the fault model complexity increase, the effort to reconnect the ports and implement the faults also increases. The fault model description is embedded into the unit controls and FIMs logic. To enable fault model description, it is necessary to create a mechanism to describe fault models and then, generate the FIMs automatically. Another approach for that would be to describe a generic FIM that has all possible faults and then, create an interface to the fault model description. Such approach is hard to implement because the FIM needs to be generic and also, it depends on the module that will be tested.

Misera, Vierhaus & Sieber (2007) showed that VHDL techniques can be applied to SystemC models. The authors developed conventional fault injection techniques, such as *saboteurs* and *mutants* for SystemC models and presented the first results for fault injection in SystemC. Although these techniques are widely used in VHDL and Verilog, they require modifications in the source code. It is desirable to keep the Design Under Test (DUT) without changes or introduces a few modifications as possible in order to avoid the fault injector changing the system behavior itself. The authors applied saboteurs and mutants changing the DUT source code for that; modifications on DUT is a characteristic of those techniques. Besides, a less intrusive approach was presented by the authors, the simulated command for SystemC. In simulated command fault injection, the authors overwritten some classes in the SystemC library instead of changing the DUT source code. Although the simulated command is less intrusive than saboteurs and mutants, it usually requires source code changing due to the limitations of fault location types. It is more complicated than others as the modifications are made extending the SystemC library.

Shafik, Rosinger & Al-Hashimi (2008) proposed a less intrusive fault injection simulation technique for SystemC models. The approach consists in replacing original C++/SystemC types with versions that have faulty behavior. For instance, the *sc_logic* type can be replaced for other fault injection enabler type *LogicReg*; this type has faults enabled by the manager mechanism. The advantage of this approach is less intrusive than the other techniques, but it is not entirely non-intrusive because the original types need to be modified by the fault types. One of the main advantages of (SHAFIK; ROSINGER; AL-HASHIMI, 2008) is that it does not require the *sc_start* method replacement to avoid undesired initialization as was replaced by *sc_initialize* and *sc_cycle* in (MISERA; VIERHAUS; SIEBER, 2007).

Lu & Radetzki (2011) showed an efficient and comparative fault simulation method in SystemC, based on datatype extension on SystemC kernel. The difference between this work and (SHAFIK; ROSINGER; AL-HASHIMI, 2008) is the injection method. Lu & Radetzki (2011) implemented a kernel extension instead of source code modifications, and the structure of simulation is made as a concurrent and comparative simulation(CSS) to speed up the simulation. Although no new representation format and no new simulator have to be created, this method needs source code transformations in the SystemC kernel to extend the data type. The approach presented in (LU; RADETZKI, 2011) is extended with more examples and explanations in (LU; RADETZKI, 2013). However, the focus of the articles was not the description of the fault model, but the fault injector mechanism and its optimization in terms of execution time.

Beltrame et al. (2008) introduced a new technique for fault injection based on reflective properties of programming languages. Beltrame et al. (2008) created a non-intrusive reflective simulation platform (ReSP) that enables SystemC and Python interoperability through automatic Python wrapper generation. ReSP makes it possible to query, examine and modify the internal status of the SystemC models. Bolchini, Miele & Sciuto (2008) developed a platform with ReSP to present a fault injection environment and a fault model to explore a multi-processor based platform. The paper showed the versatility of ReSP platform developing for injecting transient and permanent faults. The drawback of this approach is the complexity to extend the fault injector, and it is necessary the Python language integration.

In the ReSP environment, faults can be described by the console, which is useful during debugging. To conduct fault injection campaigns or to perform many runs on the same architecture, it is necessary to describe the fault models in an XML file, in the same way as other fault injectors. Neither (BELTRAME et al., 2008) nor (BOLCHINI; MIELE; SCIUTO, 2008) presented the XML syntax for fault model description or example descriptions. The XML description introduces repeatability of fault injection testing. However, as seen in (MICHAEL; GROSSE; DRECHSLER, 2011), fault model descriptions in XML are more verbose than fault model description contained in SystemC itself.

Another example of work using XML for fault model description is (YAN et al., 2017). In their approach, the authors generate SystemC faulty components from XML descriptions. Thus, each component can be configured to have a faulty behavior in the simulation. The fault

injector is characterized as a saboteur technique as it creates faulty components to perform the fault injection. The difference from this approach to the other saboteurs presented is the automatic generation of components using the XML descriptions.

Albertini, Rigo & Araujo (2012) implemented a white-box introspection mechanism. Since SystemC does not have reflexive capabilities, the authors implemented a white-box introspection mechanism by porting Reflex-SEAL (ROISER; MATO, 2005) library to SystemC. Although the authors did not use the technique to test fault injection and did not perform the test for bus developing, this work can be extended to perform fault injection tests and also supports bus design and verification.

In addition to the techniques presented, several tools were developed for injecting faults into simulations, but the majority of tools are designed for VHDL language and rarely for SystemC models, such as VERIFY (VHDL-based Evaluation of Reliability by Injection Faults Efficiently) (SIEH; TSCHACHE; BALBACH, 1997) and MEFISTO (ARLAT et al., 2003). There are a few tools to automate the fault injection into SystemC-based models, one of them is the SCEMIT. Proposed by (LISHERNESS; CHENG, 2010), this tool aims the automated injection of errors into C/C++/SystemC models using the mutants approach. Although this tool improves the fault injection, turning the process more automatic than other approaches, the mechanism to perform fault injection is the same mutant approach presented first in (MISERA; VIERHAUS; SIEBER, 2007).

3.2 SUMMARY OF RELATED WORK

Several techniques and tools for fault injection in SystemC models have been proposed, most of them adapted from approaches that were initially designed for VHDL at a low level of abstraction. With the increasing complexity of systems, the need for high levels of design abstraction and new techniques for system verification has arisen. Although there are many approaches for fault injection in SystemC, each one has advantages and drawbacks. The previous presented state-of-art fault injection techniques are complicated to use when the engineer desires to integrate them with another verification methodology because they require considerable manual effort to make a sample fault injection experiment and to implement the testbench. Table 1 summarizes the main techniques with their characteristics and compare with our proposed technique.

Although the fault injection techniques presented in literature were successfully applied for system dependability validation, it is challenging to attach and synchronize them with other verification methodologies such as UVM. Our approach suppresses this complexity of integration with UVM, and consequently takes all the advantages of the UVM's structure. Our technique is based on simple data sharing mechanism implemented by UVM. This property turns fault models feasible and straightforward implementation to cover all fault model attributes and the majority of fault perturbations presented in Section 2.1. Besides, most of the works do not focus on user-friendly formal fault model description, an essential requirement

for designing fault injection and the possibility of easy fault description and expressiveness for testers.

Table 1 – Existing fault injection techniques and our proposal.

Related work	Method	Injection methods	Fault location	Fault model description	Intrusiveness level
(ROTHBART et al., 2004) (CHANG; CHEN, 2007)	Fault injectons modules (sabouter)	Source code modifications	Interconnections	Inside fault modules	DUT
(MISERA; VIERHAUS; SIEBER, 2007)	Sabouter	Source code modifications	Signal stuck-at	Inside fault modules	DUT
	Mutant	Source code modifications	Any	Inside fault modules	DUT
	Simulator commands	Suspend, continue, stop get/set	Public variables	External Commands	Overload DUT objects/types
(SHAFIK; ROSINGER; AL-HASHIMI, 2008)	Data type extension	Source code modifications	Extended variables	External Commands	Overload DUT objects/types
(BELTRAME et al., 2008) (BOLCHINI; MIELE; SCIUTO, 2008)	Reflective wrapper	Suspend, continue, stop get/set	All SystemC elements	XML and Console	Non Intrusive
(MICHAEL; GROSSE; DRECHSLER, 2011)	TLM mutants	Extend SystemC TLM-2.0	TLM	XML	SystemC TLM library
(LU; RADETZKI, 2011) (LU; RADETZKI, 2013)	Data type extension	Kernel extension	Extended type variables and signal	Not given	SystemC Kernel
(ALBERTINI; RIGO; ARAUJO, 2012)	Reflective wrapper	white-box introspection mechanism	Public variables and signals	Console	Non Intrusive
(YAN et al., 2017)	Sabouter	Source code modifications	Any	XML	DUT
Proposed approach	**UVM data sharing mechanism**	**UVM resources/configuration databases**	**Public variables and signals**	**DSL**	**Non Intrusive**

Source: The author.

4 THE PROPOSED FAULT INJECTION TECHNIQUE AND ITS INTEGRATION INTO THE UVM

This chapter presents our Fault Injection (FI) technique that is fully integrated into the Universal Verification Methodology (UVM). Section 4.1 introduces the UVM-FI, which allows for performing fault injection upon a UVM testbench. Section 4.2 presents the details of the proposed Domain-Specific Language (DSL), designed to describe formal fault models. Section 4.3 shows details of the UVM and UVM-FI integration, in order to facilitate further improvements and extensions. Section 4.4 presents the conclusion remarks that surround the proposed UVM-FI.

4.1 UVM-FI

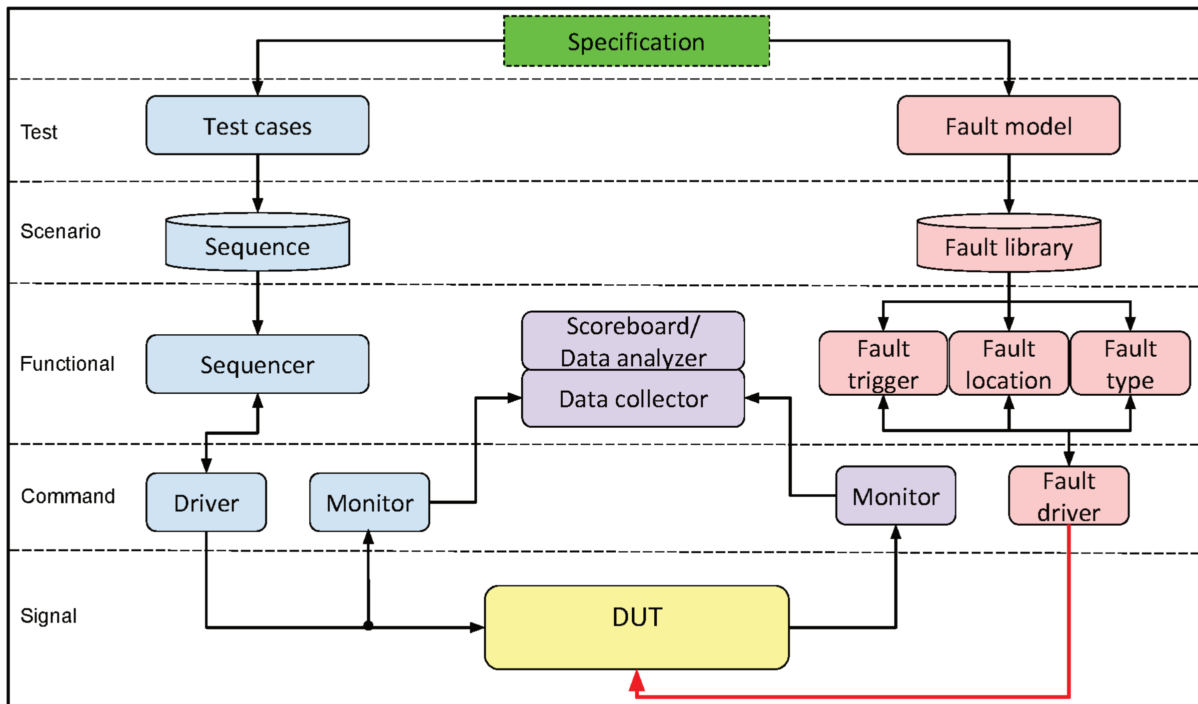
In last years, designers are relying more and more on the Universal Verification Methodology (UVM) to verify very complex embedded systems. Meanwhile, fault injection is recognized as a key technique in the verification process. However, the integration of fault injection and UVM in the same design flow is very complex. Although there are many tools to enable fault injection, they do not reuse previously created UVM components. Figure 2 and Figure 3 (Chapter 2) show components, such as monitors, data analyzers, and stimuli generators which, despite existing independently in UVM and in fault injection environments, assume similar roles. The envisage integration between UVM and fault injection avoids unnecessary re-implementation of those components and enables dependability analysis earlier in the verification phase.

Figure 7 illustrates the layered architecture of UVM-FI, from the specification to the DUT signal interfaces. The left-hand side of the figure shows the components belonging to the UVM (light blue) whereas the right-hand side shows the fault injection components created for the UVM-FI (salmon). Besides modeling the design itself, the specification also provides the requirements to create the test cases that are used to design the sequences for a UVM testbench scenario. The functional layer contains the data analyzer, which is the the lowest abstraction for sequences and monitors. The monitors and drivers provide the interface between the layer above and the DUT through the signal layer. On the right-hand side of Figure 7, the FI components can be organized following the same layered architecture. Fault models can be created from the specification, based on the desired dependability evaluation. Fault models consist of high abstractions for fault scenarios. A fault library is a set of fault triggers, fault locations, and fault types. Many scenarios can be expressed using the fault library. In a fault injector, the driver component is responsible for interfacing the fault model representation from the functional layer to the DUT interfaces.

The Accellera UVM-SystemC [1] class library uses the standard UVM patterns constructors for test and sequence creation, verification components and testbench configuration

[1] http://www.accellera.org/downloads/drafts-review

Figure 7 – Proposed layered architecture for UVM-FI.



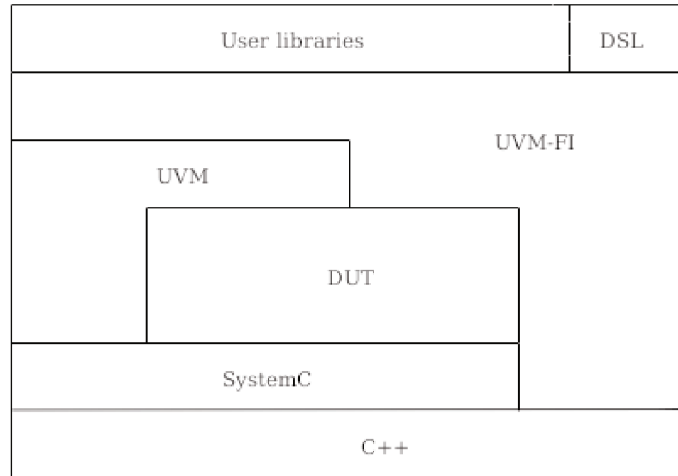
Source: the author.

and execution. UVM provides a monitor base class, a scoreboard for data analyzer, and a stimuli generation interface. Based on the UVM environment and on the components shared between the UVM and a fault injection environment (gray), this work extends the UVM by adding the missing components for fault injection, which are: fault location, fault type, fault trigger, and the fault driver.

Figure 8 illustrates the major components in a project developed in SystemC with UVM and UVM-FI integration. The fault injection environment builds upon the UVM, by adding an Event-Condition-Action (ECA) engine and a Domain-Specific Language (DSL) for fault model description. The lowermost layer is the C++ language. It means that every program built in SystemC will be compiled with a C++ compiler to generate the executable. The UVM-FI is developed in C++ and is included into the UVM. Also, the UVM-FI provides an interface to connect with the DUT via UVM standard resources.

Figure 9 shows the UVM-FI fault injection architecture, highlighting the engine. A fault model description is the input to the UVM-FI. Based on the desired fault specification, the engine will manage the registered actions and fault types to perform the fault injection at the intended moment. The engine receives fault models from the test engineer and performs fault injection. A fault model consists of a fault trigger (i.e., a condition that triggers the fault injection), a fault location (i.e., wherein the DUT the fault will be injected) and a fault type (i.e, what will be the interference). The engine splits the fault trigger into a set of event sources and a boolean condition yielding, respectively, the event and condition elements of ECA. To

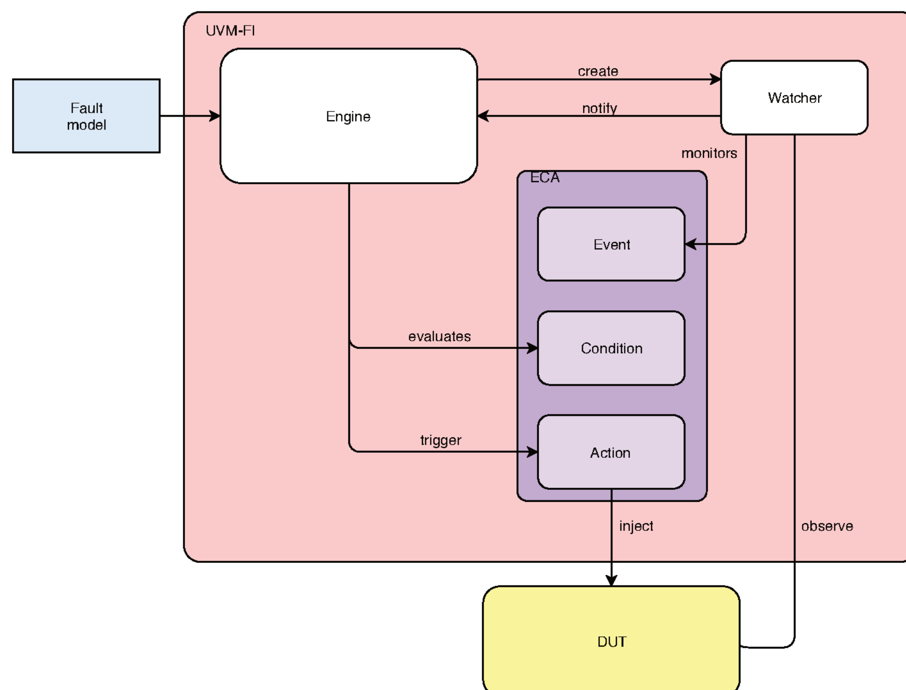
Figure 8 – SystemC project with UVM and UVM-FI.



Source: The author.

observe specific variables and signals from the DUT the engine creates watcher components. The fault location and fault type are given by a single fault injection component, that the engine stores as the action element. If the monitored element changes, the watcher notifies the engine. This notification causes the engine to evaluate the relevant fault condition. If the condition is valid the engine triggers the action. The action component consists in injecting the fault into the DUT. The description of fault conditions may be expressions in the proposed DSL which will be presented in Section 4.2.

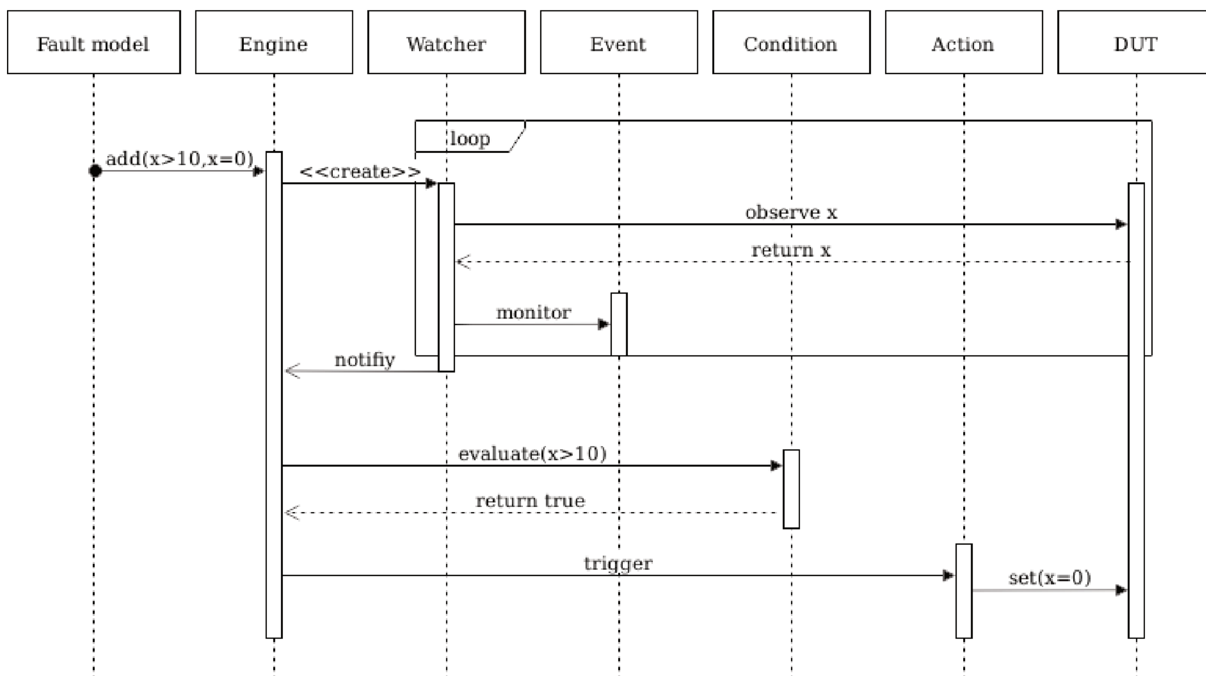
Figure 9 – UVM-FI fault injection components.



Source: The author.

Figure 10 presents a sequence diagram for an example of fault injection flow. Suppose that the desired fault model is to set the DUT variable “x” to zero when “x” is greater than 10. The fault model is compacted in an input method with two arguments, “x>10” (trigger) and “x=0” (fault location and fault type). The engine creates a watcher for the variable “x”, which monitors the variable continuously. When the variable changes, the watcher notifies the engine immediately, causing it to evaluate the condition “x>10”. If the condition is true, the engine triggers the fault by executing the registered action. The action injects the fault in the DUT, so the “x” variable value becomes zero. The created watcher continues monitoring the DUT, and it can notify the engine if another change is observed.

Figure 10 – Sequence diagram for an example of fault injection flow.



Source: The author.

The advantage of UVM-FI is that the design verification with fault injection can be built using only the UVM components, without requiring another framework, tool, or language to perform the fault injection. In addition, fault models can be reused as well as the UVM Testbench environment itself. Many advantages of UVM already implemented like the UVM DUT interfaces are useful to the fault injection environment. Besides, UVM provides interfaces to connect the Testbench to the DUT; this is not a simple task, if a non-intrusive approach is desired. Although the proposed fault injection technique has many advantages, it has some drawbacks. For instance, it has a limited number of fault types, just like other tools. Another drawback concerns the overhead of monitoring many variables in the DUT. The use of a real parallel fault trigger would overcome those limitations, but SystemC does not officially support parallel kernel simulation. Alternative parallel SystemC kernel implementations are available,

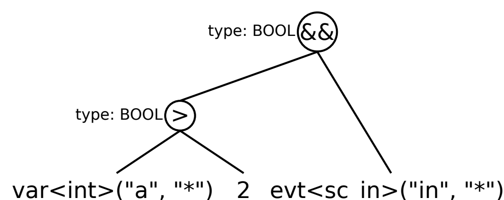
but they are not maintained by Accellera group, such as (VENTROUX; SASSOLAS, 2016) and (CHANDRAN et al., 2009).

4.2 A DOMAIN-SPECIFIC LANGUAGE TO AUTOMATE FAULT INJECTION IN SYSTEMC MODELS

A number of tools are capable of injecting faults, but there is lack of improvements on the fault models description languages. Consequently, many of those fault injection tools require a considerable manual effort to describe fault models. In this work, we improve the fault models description by developing a Domain-Specific Language (DSL) that helps designers to create fault models in SystemC. The main goal of a DSL in the context of the present work is to describe expressions mixing any C++ value and variables with C/C++/SystemC variables and objects (e.g., signals). When the compiler processes such expressions, the resulting code does not execute them but instead, outputs an Abstract Syntax Tree (AST). This AST can be later inspected and executed several times, at run-time, both for determining if a fault injection condition has been satisfied and as part of the fault injection.

The output produced by compiling the DSL expressions is always an AST; therefore, the DSL itself is comprised of nodes that can be combined with all C++ logical and arithmetic operators. There are three types of terminal nodes on the AST: constants, captures, and variables. Constants have their value assigned during the parsing of the expression (at run-time, when the AST is constructed); captures are initialized at the same moment, from C/C++ variables and store a reference to their value, being subject to change; finally, variables represent objects accessed at evaluation time through UVM introspection mechanisms. Every C++ logical and arithmetical operator is also a node type in the DSL, containing one or two children (e.g., ! is unary and + is binary).

Figure 11 – Example of DSL expression and corresponding AST.



Source: The author.

Figure 11 shows an example of DSL expression and the corresponding AST, where `"*"` is the UVM variable scope in the UVM database and `"a"` and `"in"` are variable names. The leftmost and rightmost leaves are variable nodes and refer through introspection to objects inside the DUT. The middle leaf is simply a constant. This expression evaluates to `true` whenever the default event of `in` occurs and, at that moment, `a > 2` holds. For efficient detection of when the expression becomes true, the leaf nodes of the AST are visited in order to create watcher

objects. These objects exploit UVM introspection mechanisms in order to promptly notify any interested party when the expression must be re-evaluated.

The atomic elements of the DSL, whose presence allows the C++ compiler handling the whole expression as a DSL expression, are `var`, and `evt`. A `var` object stores the name of a DUT object registered in the UVM configuration database. An `evt` object is similar to a `var` object, but refers to the default event of SystemC ports and signals. For the built-in C++ operators, `var` behaves like the referred object and `evt` behaves as a Boolean (true after the event occurs).

In addition to `var`, `evt`, and the built-in C++ operators, the DSL includes other helper constructs. The `cap(x)` function allows the DSL expressions to bind to a C++ variable `x` by reference, instead of copying its value. The `unif(a, b)` function represents a random value uniformly distributed between `a` and `b`. The `call(f, a1, ...)` function represents the return of function `f` if called with arguments `a1, ...`, which may themselves be a DSL expression. This provides an escape hatch for any situation where complex or stateful calculations are required as part of the DSL (e.g., coordinate transformations or finite impulse filters).

The UVM-FI fault model conditions are expressed using the DSL. The expression is parsed to generate the AST that will be an input to the engine. For example, if it is desired to inject a fault every time that some event (event_x) occurs with probability of 50%, such scenario can be expressed using the DSL as `"event_x && (50 > unif(0, 100)"`, where a logical AND (&&) condition is used to express the relation between the event occurrence and its probability of 50%. Another functionality provided by the DSL is the helper function "unif" which describes a uniform probability. Appendix B describes the complete list of DSL C++ overloaded operators and functions implemented in this dissertation. Other functions can be implemented to improve the UVM_FI capabilities.

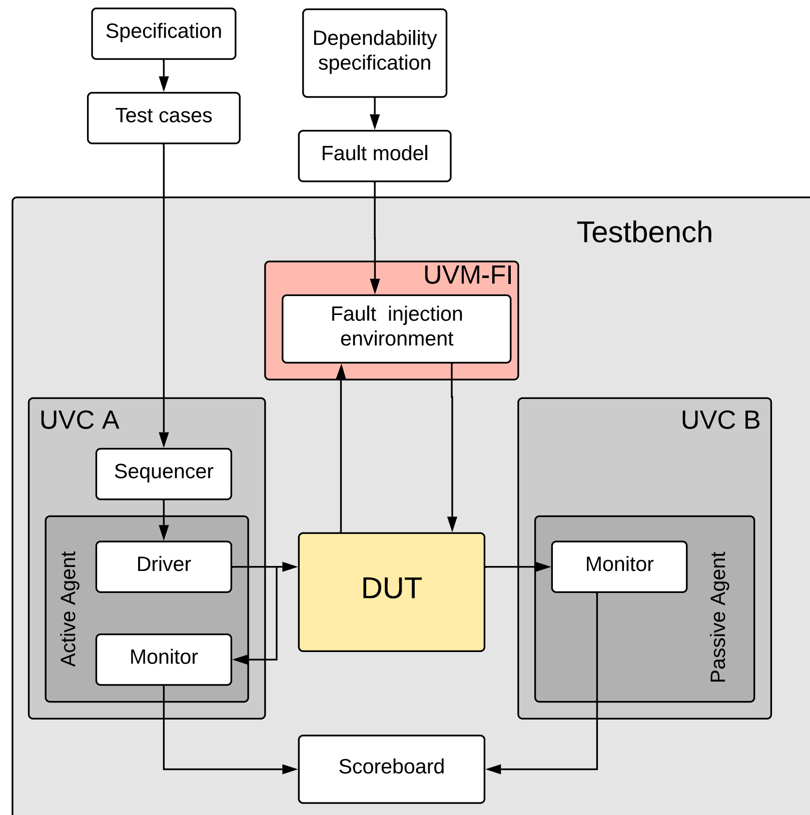
4.3 INTEGRATION INTO THE UNIVERSAL VERIFICATION METHODOLOGY

The capacity of integration with UVM is one of the main advantages of the UVM-FI. The dependability analysis can be developed in early phases of the project, complementing the verification step instead of being performed only in the final tests. For the UVM-FI be completely compatible with UVM, it needs an interface component that extends from the UVM components and can be included in the testbench. Besides, the library has to provide an efficient fault model description in terms of capability of expressiveness and effort necessary to describe a fault that will be injected in the system.

Figure 12 depicts the fault injection coupled with the UVM environment where the UVM-FI is represented as a new component of the UVM, such as a new type of UVM Verification Component (UVC). By this representation, the UVM-FI is in the same testbench level of the other UVCs. The SystemC-UVM provides a base UVM environment class (*uvm_env*) to create hierarchical containers of other components that together complete the entire environment. The "Testbench" component in Figure 12 represents the UVM environment component.

The fault injector should be create in the “Testbench” at same level of other verification blocks. Although in Figure 12 the environment represents the entire testbench, later it can be reused as a sub-environment in the system integration phase.

Figure 12 – UVM-FI Testbench architecture.



Source: The author.

The fault injection component is an instance of an object that contains a fault injection environment. In the fault environment, all properties to perform the fault injection are defined, such as the fault trigger, location, and type. It is worth mentioning that the user may create types of faults that are not implemented in the UVM-FI. For example, the designer can implement a method to flip bits in a memory and invoke it with the method `call()` previously described in Section 4.2.

Listing 1 shows the testbench fault injector environment component. The component extends from the *uvm_base_fault_env* the base UVM-FI external interface class. The fault injector is integrated with the UVM-FI, but the fault models need to be registered in the engine. For that, it is necessary to define a condition and a fault function. The *uvm_base_fault_env* has a public engine object that is accessible to create interfaces at the UVM testbench level. The engine allows the registration of fault model calling the method *register_fault_condition(condition, fault)*.

The *register_fault_condition* ECA *engine* method is used to register a fault model at the Testbench. To make the DSL more compact, the $<<$ operator is overloaded to be used in

```
class fault_env : public uvm_base_fault_env {
    ...
  void build_phase(uvm::uvm_phase& phase) {
    ...
    int value = 0;
    evt<sc_out<bool>*> condition("evt_trigger","*");
    *engine << fm(condition,
            var<int*>("location", "*").set(cap(value)));
  }
  UVM_COMPONENT_UTILS(fault_env);
};
```

Source: The author.

Listing 1: UVM fault environment.

place of the aforementioned method. When this operator is used with C++ output streams, fault models can be repeatedly streamed into the ECA engine. A helper function, `fm` (from "fault model") is defined to group a fault trigger (as the first argument) and a fault injector (as the second argument), both described using the DSL.

The UVM provides a mechanism that allows storing DUT public objects, such as variables, signals, and ports. This mechanism is the base of the proposed fault injector technique. With the store and restore capability provided by UVM, DUT objects can be stored in the testbench and then they can be restored in a UVM-FI environment.

The UVM configuration database and the resources database are two important features to allow fault injection without extra tools. The UVM configuration database is a data sharing mechanism where one can also include the DUT configuration and share it with fault injector components. The *uvm_resource_db* can also be used for that, but the difference is that the *uvm_config_db* is built on top of the *uvm_resource_db* and provides a model hierarchy support (COOPER; MARRIOTT, 2014).

Listing 2 shows a fragment of code to add an object (*dut_if_in*) to the UVM database. To do that, the database provides a method *set()*. The first argument is the context, the local to begin the search for the object in the database. The second argument is the scope, in the Listing 2 the parameter "*" meaning the global scope. The third argument is a unique name of the variable in the database. The last argument is the object instance. Any object can be stored in the database, including pointers to internal DUT elements.

```
vip_if* dut_if_in = new vip_if();
uvm::uvm_config_db<vip_if*>::set(0, "*", "vif", dut_if_in);
```

Source: The author.

Listing 2: Store variable in UVM data share.

At any moment of the simulation, the stored variables can be restored from the database

using the method *get()*. Listing 3 retrieves the object added to the database by the code in Listing 2. The method has the same structure for the arguments of *set()*, but the last argument is an object that the retrieved value is assigned to.

```
vip_if* vif_in;
uvm::uvm_config_db<vip_if*>::get(0, "*", "vif", vif_in)
```

Source: The author.

Listing 3: Restore objects from the UVM data share.

The UVM-FI interfaces the DUT variables stored in the data share mechanism through *uvm_var* (*var* in DSL abstraction). For example, in Listing 1 the fault location is *var<int*>("location", "*")*. The variable is an *int* stored in the data share mechanism with the name "location" and at the global scope ("*"). The *uvm_var* will handle the variable internally in UVM-FI library and whenever an action for that location is triggered the UVM-FI modifies the variable using the UVM database to access it.

The fault injector parameter from DSL *fm* represents both the fault type (which fault is to be injected) and the fault location (where the fault is to be injected). The UVM-FI provides a `set()` method as fault type. This method assigns the argument value to the location. Line 8 in Listing 1 shows an example with method `set()`. A local variable "value" is assigned to the stored variable "location". Although the method `set()` can be used to express most of fault types, the UVM-FI can be easy extended to other types of faults, such bit-flip.

The ECA engine creates a watcher to monitor the DUT elements (Figure 9). The UVM-FI needs to check the current condition for each watcher created. For that, the UVM-FI implements a component that frequently requests, from the engine, an evaluation for each available watcher. Such component is called *uvm_poller*. The component starts to run at the beginning of the simulation and notifies the engine every 1 nanoseconds. The *uvm_poller* notification time can be configurable, according to the precision required to evaluates the watchers, some fault events can not be triggered if the notification time interval is too long.

4.4 UVM-FI CONCLUDING REMARKS

Unlike most of the state-of-the-art correlate works, its code is open and can be downloaded from https://gitlab.com/lohmann/uvm-fi. The new components inserted in UVM are extended from the *uvm_component* class. This class provides interfaces for hierarchy, phasing, factory, process control reporting and recording. Although the fault injection is not part of the UVM flow, with this extension one can easily control the fault injection and synchronize the tests with the UVM environment.

The integrated UVM-SystemC fault injection allows access the DUT components in a non-intrusive approach using UVM database resources. Also, the simulation is controlled by

UVM resources, such as UVM phasing, and events. Providing an easy fault injection integration with Testbench both the fault injection complexity and the effort to test the design are reduced.

5 EVALUATION OF THE PROPOSED FAULT-INJECTION TECHNIQUE

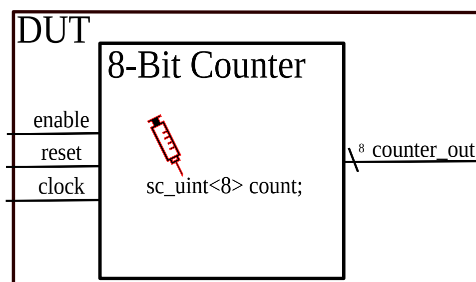
This chapter describes the case studies to validate and evaluate the proposed fault injection technique integrated with UVM, the UVM-FI. Firstly, Section 5.1 presents a fault injection example performed in an 8-bit counter. Then, Section 5.2 shows a fault injection evaluation in terms of fault model description for three different scenarios. Lastly, Section 5.3 demonstrates the fault injector capabilities applied to a matrix multiplier with fault-tolerant mechanism and a virtual platform with the MSP430 microcontroller Instruction Set Simulator (ISS) with Transaction-level Modeling (TLM) memory model.

5.1 UVM-FI VALIDATION

We performed fault injection in an 8-bit counter designed by (SHAFIK; ROSINGER; AL-HASHIMI, 2008). We compared the proposed technique with those presented in (MISERA; VIERHAUS; SIEBER, 2007) and (SHAFIK; ROSINGER; AL-HASHIMI, 2008), in terms of intrusiveness and simplicity. Besides, results for a fault injection execution using the UVM-FI are shown to validate the proposed fault injection technique.

Figure 13 depicts the block diagram with interfaces of the 8-bit synchronous counter adopted as a DUT. It has an enable, a reset, and a clock as inputs; an 8-bit port as output. The enable input is used to activate it whereas the reset forces its output to zero. The counter starts at zero and is incremented at every positive clock event, being 256 its maximum value. There are many ways to implement an 8-bit counter in SystemC, using flip-flops or with fault-tolerant mechanism; however, this experiment uses the original code from (SHAFIK; ROSINGER; AL-HASHIMI, 2008).

Figure 13 – 8-bit counter DUT.



Source: The author.

Listing 4 presents the 8-bit counter from Shafik, Rosinger & Al-Hashimi (2008) with no fault injection modifications. The module “counter8bit” runs a thread *incr_count()* method that increments the output value. The thread method is sensitive (called) by reset and positive clock edge event. The count value is stored in a class variable type of *sc_uint<8>* called “count”. If the variable enable is set up and the reset value is set to zero, the counter increments the value and writes it in the module output port (*counter_out*) each time that the *incr_count()*

is called. This implementation is a high-level abstraction of 8-bit counter developed in SystemC. It does not contain any fault tolerance mechanism nor considers the time in operations to perform as real counter. Therefore, many improvements can be made to this implementation to increase the system dependability and reliability.

```
SC_MODULE (counter8bit) {
  sc_in_clk clock ;
  sc_in<bool> reset, enable;
  sc_out<sc_uint<8> > counter_out;
  sc_uint<8> count;

  void incr_count() {
    while(true) {
      if (reset.read() == 1) {
        count = 0; counter_out.write(count);
      } else if (enable.read() == 1) {
        count = count + 1;
        counter_out.write(count);
      }
      wait();
    }
  }
  SC_CTOR(counter8bit) {
    SC_THREAD(incr_count);
    sensitive << reset << clock.pos();
  }
};
```

Source: The author. Adapted from (SHAFIK; ROSINGER; AL-HASHIMI, 2008).

Listing 4: 8-Bit counter SystemC implementation.

Misera, Vierhaus & Sieber (2007) show a mutant component applied to SystemC language. Mutant is a well-known strategy to perform fault injection in systems described in a Hardware Description Language (HDL, such as VHDL and Verilog). In this technique, a modified component replaces the original component, the mutant. The mutant component allows for changing the behavior of the original system. Listing 5 shows a mutant implementation for the previously present 8-bit counter. The mutant implementation inserts an additional control input (*fault_activate*) that enables the fault in the output counter. This method is highly customized and many types of faults can be applied. The drawback of mutant fault injection is its intrusiveness. A new component is implemented with a different logic from the original code and thus, new errors that were not present in the original code can be introduced due to the code modifications needed to create the mutant component.

Besides the mutant approach, Misera, Vierhaus & Sieber (2007) show a simulated command fault injection technique. The simulated command technique adds new commands to the simulation that allow signal and variable modification. To implement this approach, the authors overwrote some SystemC library classes. For this approach, the ports, signals, and variables must be converted into type vectors. For instance, a variable *sc_logic* has to be

```
SC_MODULE (sabouter) {
  sc_in<sc_logic> fault_activate;
  sc_in<sc_logic> reset;
  sc_in<sc_logic> enable;
  sc_out<sc_logic> counter_out;
  sc_lv<8> count;

  void incr_count() {
    while(true) {
      if (reset.read() == 1) {
        count = 0;
      } else if (enable.read() == 1) {
        count = count.to_int() + 1;
      }
      if (fault_activate.read() == 1) {
        counter_out.write(0);
      } else {
        counter_out.write(count);
      }
      wait();
    }
  }
  SC_CTOR(counter8bit) {
    SC_THREAD(incr_count);
    sensitive << reset << clock.pos();
  }
};
```

Source: The author.

Listing 5: 8-Bit counter using mutant fault injection presented in (MISERA; VIERHAUS; SIEBER, 2007).

modified to *sc_lv<8>*. This conversion can be automated by scripts, decreasing the manual effort necessary.

Listing 6 shows the modification needed to perform fault injection in the 8-bit counter using simulator command. Although the authors use a (perl-) script to convert the SystemC data types, it requires original code modification which results in extra manual effort. In Listing 6 the original variable *sc_uint<8> count* was converted into an logic vector type *sc_lv<8>* due to the limitation of Misera, Vierhaus & Sieber (2007) approach (SHAFIK; ROSINGER; AL-HASHIMI, 2008). Although the simulation command needs less code modification, it is an intrusive technique and also requires high effort to extend SystemC. Besides, the (MISERA; VIERHAUS; SIEBER, 2007) simulator has some limitations, such as the modification of *sc_uint* data type.

Another simulator command technique is presented in (SHAFIK; ROSINGER; AL-HASHIMI, 2008). This other approach replaces the original C++/SystemC data, and signal types to fault enabled types; then, it uses commands to control the fault injection. For instance, in Listing 7 the *sc_uint<8> count* variable type is swapped to a fault type *UIntReg<8>*. By the command “FIMgr::getInstance().setFaultLocation(loc)” a fault location is set, where the

```
SC_MODULE (counter8bit) {
  sc_in<sc_logic> reset ;
  sc_in<sc_logic> enable;
  sc_out<sc_logic> counter_out;
  sc_lv<8> count;

  void incr_count() {
    while(true) {
      if (reset.read() == 1) {
        count = 0;
        counter_out.write(count);
      } else if (enable.read() == 1) {
        count = count.to_int() + 1;
        counter_out.write(count);
      }
      wait();
    }
  }
  SC_CTOR(counter8bit) {
    SC_THREAD(incr_count);
    sensitive << reset << clock.pos();
  }
};
```

Source: The author. Adapted from (SHAFIK; ROSINGER; AL-HASHIMI, 2008).

Listing 6: 8-Bit counter using simulator command presented in (MISERA; VIERHAUS; SIEBER, 2007).

"loc" parameter represents the fault location object from DUT, the fault model is described as a sequence of commands. Although this technique does not have fault type limitations, such as the variable types present in (MISERA; VIERHAUS; SIEBER, 2007), it also requires DUT source code modifications and manual effort to extend the SystemC language.

To perform fault injection using the technique proposed, it is necessary to create a UVM Testbench environment. Although it is a specific requirement for our fault injection approach, the works discussed so far in this section also require the development of a Testbench in order to run the tests, that could be a simple Testbench or other standard methodologies such as UVM. UVM improves the system tests with modular Testbench components and decreases the time and effort to create the test environment.

First, we created a Testbench with UVM to run the original counter (Listing 4), along 10 ns of simulation. In this test environment, the clock period is 1 ns; initially, the reset value is 0, and the enable value is 1. Figure 14 shows the waveform diagram for the simulation. In such simulation no fault injection was performed. Instead, it simply exercises the designed 8-bit counter with UVM in normal operation.

With a UVM setup ready, we integrated the proposed fault injector (Section 4). We envisaged to set the DUT internal counter variable to 0 along with a specific interval which means setting a stuck-at-0 fault in the counter. For that, we created a method *FaultTrigger-Timer(from_time, to_time)* that returns true if the simulation time, in nanoseconds, is between

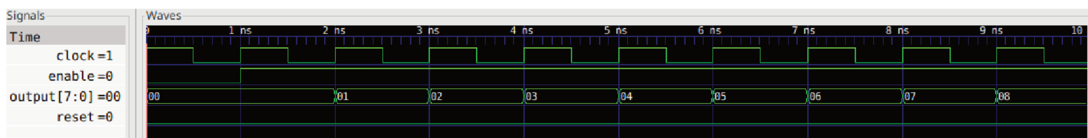
```
#include "fim/FIReg.h"
SC_MODULE (counter8bit) {
  sc_in_clk clock ;
  sc_in<Reg<bool> > reset ;
  sc_in<Reg<bool> > enable;
  sc_out<UIntReg<8> > counter_out;
  UIntReg<8> count

  void incr_count() {
    while(true) {
      if (reset.read() == 1) {
        count = 0;
        counter_out.write(count);
      } else if (enable.read() == 1) {
        count = count + 1;
        counter_out.write(count);
      }
      wait();
    }
  }
  SC_CTOR(counter8bit) {
    SC_THREAD(incr_count);
    sensitive << reset << clock.pos();
  }
};
```

Source: The author. Adapted from (SHAFIK; ROSINGER; AL-HASHIMI, 2008).

Listing 7: 8-Bit counter using simulator command presented in (SHAFIK; ROSINGER; AL-HASHIMI, 2008)

Figure 14 – 8-Bit counter waveform diagram results with UVM simulation.



Source: The author.

the values passed through parameters. Otherwise, it returns false. Listing 8 shows the fault model description for this example. We set the fault trigger interval range from 5 ns to 9 ns; the fault location is the internal variable "count" set to be 0 during the interval.

```
sc_uint<8> a = 0;
*engine << fm(cnst(FaultTriggerTimer(5,9)),
              var<sc_uint<8>*>("dut_count", "*").set(cap(a)));
```

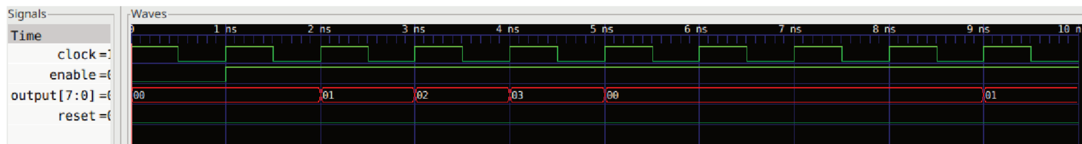
Source: The author.

Listing 8: Registering a fault condition for 8-bit counter.

Figure 15 shows the waveform diagram for the 8-bit counter with the fault injection

execution. The only modification made after creating the UVM Testbench is the integration with the fault model described in Listing 8. No modifications in the original 8-bit DUT were necessary. The resulting waveforms in Figure 15 demonstrate that in the simulation time (5 ns to 9 ns) the value of count output was set to 0. After that faulty period, the counter returned to the correct behavior.

Figure 15 – 8-Bit counter wave diagram results with UVM-FI simulation.



Source: The author.

Saboteurs and mutants require manual effort to make modifications in the original design. Those modifications usually require an expensive manual effort, and as they need behavior re-design, the faults can introduce new bugs that do not represent the real system behavior. In this section, we presented a related work that employs the mutant approach and demonstrated the high intrusiveness level required by it. We also presented two simulator command approaches, both requiring SystemC simulator modification and source code modification as well. Those simulator commands are less intrusive than saboteur and mutants, but they are not easy to implement, and require to change the language kernel. The UVM with fault injection does not require any change of the original DUT. In addition, it is not necessary to overload or extend the SystemC source code and the UVM-FI is a completely non-intrusive technique. The fault model is described only in the test environment and many fault locations, triggers, and types can be developed and efficiently performed to DUT.

5.2 EVALUATION OF THE EXPRESSIVENESS AND EASE OF USE: THE IMPACT OF THE PROPOSED DSL

This section evaluates the technique presented in Chapter 4 with respect to expressiveness and ease of use. We compare an XML based description of fault injection test with our DSL fault description and analyze those approaches in terms of the effort necessary to describe a fault experiment and the description expressiveness to perform fault conditions. Each fault injection approach is based on specific methods, such as reflective wrapper, mutants, simulated commands and so on.

The evaluation employs three scenarios of fault injection, in order of increasing complexity. Three fault model description techniques are applied to each of these scenarios. The first technique is the XML-based proposed in (MICHAEL; GROSSE; DRECHSLER, 2011). This was the only approach for which we could obtain some code describing fault injection. However, the chosen scenarios are limited by the absence of the full language specification and the actual tool. The second technique is the proposed fault injection integrated with UVM,

which was published in (LOHMANN et al., 2018b). It is worth noting that this version does not employ the proposed DSL. The third one is the proposed technique improved by using the proposed DSL.

Scenario A consists of data modification. Listing 9, adapted from (MICHAEL; GROSSE; DRECHSLER, 2011), specifies such scenario. In this example, one byte, starting at position 200, is modified. As the modification is an `or` mask, the effect is an assignment of that byte.

```
<data>
 <transfer start pos="200" length="1">
     <or mask="0xff"/>
   </transfer>
</data>
```

Source: The author. Adapted from (MICHAEL; GROSSE; DRECHSLER, 2011).

Listing 9: Scenario A: XML fault model.

Listing 10 shows the same fault model description using the proposed fault injection technique without the DSL. Therefore, before the fault condition is defined, the objects used in the description must be in the UVM database. Any accessible DUT component from the Testbench can be stored in the UVM config database, such as signals, ports, and other SystemC types.

```
bool truth = true;
engine->register_fault_condition(new uvm_var_ct_tpl<bool>(1),
    new uvm_fault_set(new uvm_var_tpl<bool>("tgt", "*"),
        &truth, 0, 1));
```

Source: The author.

Listing 10: Scenario A: fault model without DSL.

The size of the code in Listing 10 hinders its readability. Listing 11 shows the fault model description from scenario A expressed in the DSL. The proposed DSL generates the same effects but with a more compact and intuitive syntax. As the condition `true` alone is not a DSL expression, it requires wrapping. The fault type is to set a char value to a variable `tgt` of type `char`. In general, any DSL expression, in addition to C++ values, can be used as argument of `set()`. For example, `var<char>("x") & var<char>("y")`.

```
*engine << fm(cnst(true), var<char>("tgt").set('\xff'));
```

Source: The author.

Listing 11: Scenario A: fault model with the proposed DSL.

Scenario B presents a more complex fault condition, introducing a probability. In this scenario, the variable is set in a certain percentage of its triggers occurrences. To describe this type of conditions, we use the `unif` function, that creates a uniform random distribution. Listing 12 uses random value in such distribution to build an expression whose value is `true` 90% of the situations the expression is evaluated.

```
*engine << fm(unif(1,10) <= 9, var<char>("tgt").set('\xff'));
```

Source: The author.

Listing 12: Scenario B (probabilistic): DSL description.

Faults with probability can also be created using the XML approach presented in (MICHAEL; GROSSE; DRECHSLER, 2011). To do that, the XML attributes `random=1` and `percentage=90`, must be added to the `transfer` tag at the second line of Listing 9.

Scenario C sets the value of a variable `tgt` to the average of the last 3 values observed for the signal `m_out1`. Listing 13 shows the DSL fault mode description for this scenario. This scenario relies on the `call` helper to compute the moving average with a user-defined function `mov_avg`. The moving average is updated every time the `m_out1` signal changes. In the code, `mov_avg` is a template function that computes the moving average from a history vector pointer (`&hist`), a window size (3) and a sample value (`m_out_1` current value). This scenario can not be implemented in the approach of Michael, Große & Drechsler (2011). Furthermore, the ability to call user-defined stateful functions as part of fault injection is not present in any fault injection tool, to the best of our knowledge.

```
*engine << fm(evt<sc_signal<sc_int<32> >*>("m_out1"),
    var<sc_int<32> >("tgt").set(call(&mov_avg<sc_int<32> >,
        cnst(&hist), cnst(3), var<sc_int<32> >("m_out1"))));
```

Source: The author.

Listing 13: Scenario C (Moving average): DSL description.

There are no absolutely fair metrics to compare XML and C++, as the languages do not share common elements such as statements, declarations or expressions. Comparing line numbers is also not fair as C++ lines are often longer than XML ones. A reasonable metric is character count (ignoring all optional spaces and line breaks), which is shown in Table 2 for each scenario and technique combination. In all scenarios, the DSL shows the smallest value. The main advantages of the DSL, however, become clear in a qualitative comparison. First, expressions (for fault trigger or fault type specification) are more compact and readable in C++, especially for expressions with more than a single operator. In XML, expressions must be represented in the form of a tree. In contrast, the DSL automates the generation of a similar tree implicitly. Second, `var` and `evt` objects, as well as whole DSL expressions, can be stored in

local variables and reused across several fault models. Third, `call` and the ability to mix C++ and DUT objects in the fault model allows for greater flexibility, and allows the test engineer to program highly specific fault models which cannot be foreseen by fault injection tool designers.

Table 2 – Character count per Technique per Scenario

	Technique	**Scenario A**	**Scenario B**	**Scenario C**
Characters	(MICHAEL; GROBE; DRECHSLER, 2011)	78	104	-
	Without DSL	141	183	381
	DSL	55	60	159

Source: The author.

5.3 FAULT INJECTION SCENARIOS

This section demonstrates the capabilities of the proposed fault injection technique performing in two scenarios. The first scenario is a matrix multiplication with and without fault tolerant mechanism. In the second scenario, we perform fault injection in a virtual platform with MSP430 microcontroller Instruction Set Simulator (ISS) with a TLM memory model. For both scenarios, we successfully applied our technique and validate the versatility and capability of our technique.

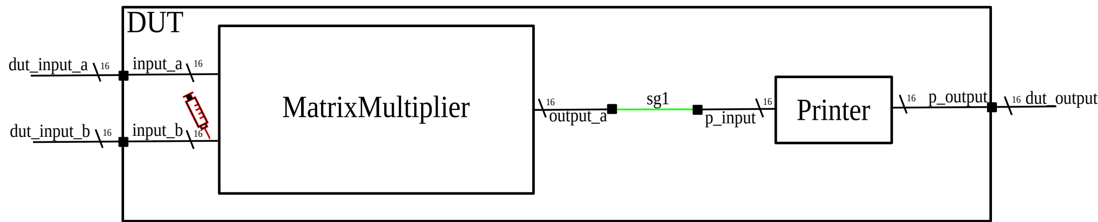
5.3.1 Matrix Multiplication

In this section, we perform a fault injection test in a matrix multiplier algorithm with and without triple modular redundancy (TMR) mechanism. For that, we implement a 4x4 matrix multiplier module that receives two input matrices and returns the resulting multiplied matrix. For the TMR example, we replicate the matrix multiplier module and connect the output of each module to a voter that chooses the most frequent matrix result, as a strategy for fault tolerance.

For both matrix examples, the input and output elements of matrices are vectors of *sc_dt::sc_int<16>* numbers, each vector represents a 4x4 matrix. The connections between the matrix multiplier and other modules are made using *sc_signal* standard communication. Figure 16 shows the matrix DUT with one matrix multiplier module and a printer used to show the matrix result. No fault tolerant mechanism was applied in this case.

Firstly, the DUT shown in Figure 16 was tested using the UVM Testbench without any fault injection tool. After creating the UVM Testbench the fault injection mechanism proposed in this dissertation was easily added to the Testbench. The first step to integrate the fault injection was compiling and linking the FI fault injection library developed and presented in Chapter 4. Secondly, we added the DUT resources needed to perform the fault injection to

Figure 16 – Matrix multiplier DUT.



Source: The author.

the UVM database. Listing 14 shows the addition of a trigger *input_b[15]* and the fault location *sig_data[15]* of DUT. Finally, we integrate the fault model into the UVM architecture, for which we create a new component class *vip_fault_env* that extends from *uvm_base_fault_env*. The *vip_fault_env* contains all the fault injection model description.

```
sc_in<sc_int<16> > *trigger = &(my_dut->matrix_multiplier.input_b[15]);
uvm_config_db<sc_in<sc_int<16> > **>::set(0, "*", "input_b_event", &trigger);

sc_signal<sc_int<16> > *f_locale = &(dut_if_in_B->sig_data[15]);
uvm_config_db<sc_signal<sc_int<16> > **>::set(0, "*", "dut_input_b", &f_locale);
```

Source: The author.

Listing 14: Registering fault model resources.

Listing 15 shows the fault model described in *vip_fault_env* class. For the simple matrix example, we set a fixed integer value to the *du_input_b* signal every simulation execution. The fault model is composed of a fault trigger, fault location, and a fault type. In this example, we set as trigger the event of writting in the 15 element (recovered from the data share mechanism as *input_b_event*) of the input matrix B, and the fault will be the same input element. The fault type is described by the function “set” that inserts the value in the desired fault location (*dut_input_b*).

```
sc_int<16> a = 777;
evt<sc_in<sc_int<16> >*> input_evt("input_b_event","*");
*engine << fm(input_evt,
              var<sc_signal<sc_int<16>>*>("dut_input_b", "*").set(cap(a)));
```

Source: The author.

Listing 15: Registering a fault condition.

In the matrix multiplication example without any fault tolerant mechanism, an injected fault is propagated to the output, meaning that a system failure occurs when a fault is introduced. Table 3 shows the tests results for our fault injection. Since the simple matrix does not have any tolerance mechanism, all effective faults injected are propagated to the output. For this

example, we defined that for each simulation, one fault occurs, and no fault trigger probability was set. Due to this fault model, Table 3 shows the relation of one fault injected for simulation, which demonstrates the expected behavior of our fault trigger implementation.

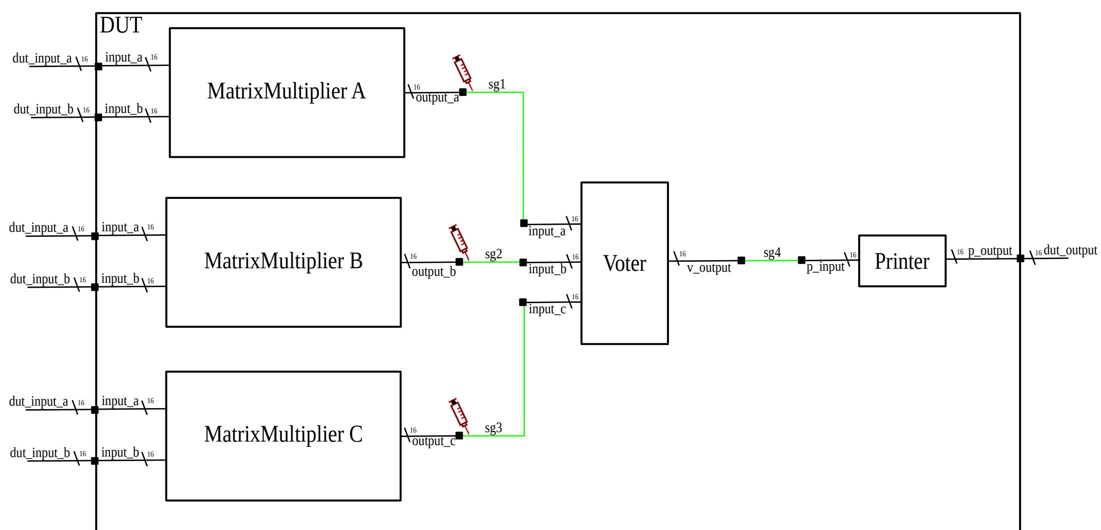
Table 3 – Fault injection results for a simple matrix multiplier.

Sim. length	Injected faults	Failures	Success
50	50	50	-
100	100	100	-
500	500	500	-
1000	1000	1000	-

Source: The author.

Figure 17 illustrates the block diagram for the triple modular redundancy (TMR) matrix multiplier. The DUT receives as input two matrices, performs three independent multiplication algorithms and chooses the most frequent matrix result.

Figure 17 – TMR matrix multiplier DUT.



Source: The author.

Table 4 shows the fault model applied in our example. Each fault registered is composed of a fault trigger, a fault type, and a fault location. In our example, a fixed value "*fixed n*" is set at signals connecting the matrix multipliers ("A", "B" and "C") to the voter component, for each multiplier input event with a uniform probability of 33%. This means that each matrix multiplier has the probability of 33% of a fault occurs independently, in one execution instance faults can be injected in multiples outputs.

Listing 16 shows a fault condition expression for the fault model in the first line of Table 4. The `unif` is a factory function for a random variable, and is part of the DSL.

After registering the entire fault model of Table 4, we run the experiment for simulations with length of 50, 100, 500, and 1000. Besides, we perform the test for simple matrix with the same fault model trigger of TMR matrix. For each simulation the fault injector set a

Table 4 – Fault model specification.

Fault trigger	Fault type	Fault location
evt<output_a>("output_a",scp) unif(0,100) <0.33	set_value(n)	sg1[12]
evt<output_b>("output_b",scp) unif(0,100) <0.33	set_value(n)	sg2[12]
evt<output_c>("output_c",scp) unif(0,100) <0.33	set_value(n)	sg3[12]

Source: The author.

```
sc_int<16> a = 323;
*engine << fm(evt<sc_out<sc_int<16> >*>("output_a", "*")
        && 33 > unif(0,100),
        var<sc_signal<sc_int<16> >*>("sg1", "*").set(cap(a)));
```

Source: The author.

Listing 16: Registering a fault condition.

fault in a probability of 33 %. Table 5 presents the results for those experiments. In this table, a failure is determined by a DUT output matrix (after the voter in TMR case) that is different from the correct multiplication of the input matrices. Success is a correct multiplication matrix output, even when faults are injected. From those results, one can note that the fault injector was able to apply the faults and the tolerant matrix algorithm is able to tolerate around 84% of all introduced faults.

Table 5 – Fault injection results

Sim. length	Injected faults	Failures	Success
4x4 matrix multiplier			
50	16	16	34
100	33	33	67
500	166	166	334
1000	330	330	670
TMR 4x4 matrix multiplier			
50	41	8	42
100	85	16	84
500	484	94	406
1000	1001	187	813

Source: The author.

This example shows that performing fault injection using the DSL is effective and intuitive. Many other fault models can be described using this technique with the SystemC modeling. Different kinds of triggers, fault types, and locations can be combined to create more efficient fault injection campaigns.

5.3.2 Virtual Platform Target Application

A virtual platform is a piece of software that can run a software for a target hardware in another system (AARNO; ENGBLOM, 2014). For example, a hardware model of ARM processor can be simulated and the software for that model can be run on another host computer that may not have the same hardware architecture. A virtual platform is not restricted to a single processor but also has interconnection protocols and peripherals.

The development of virtual platforms made possible to establish different abstraction levels, from more accurate such as cycle-accurate to untimed models, i.e. models without associated time information. The virtual platform is useful because the binaries of software can run until the real hardware is available allowing for embedded software development early, and even faster than the real hardware. The hardware model can offer access to register and variables, making possible to observe and control the hardware execution, thus facilitating the verification task.

There are many virtual platforms available and many tools to help virtual platform development. Garanhani (2015) developed a toolset called MPSoCBench that provides a completely open source simulation infrastructure including scalable hardware and software components. With MPSoCBench one can create virtual platforms for multiprocessors with data and instructions caches, shared memory, and several IPs. MPSoCBench is implemented in SystemC, and is integrated with the MiBench (GUTHAUS et al., 2001) benchmark suite. There are four processors available (ARM, MIPS, SPARC, and PowerPC) to create and test Multiprocessor Systems on Chip (MPSoCs) examples.

Zijlstra (2015) presents a virtual platform to validate the system architecture and to support the embedded software development for a Cubesat type nanosatellite. The virtual platform was developed in SystemC and the TI MSP430 microcontroller model was generated by the architecture description language ArchC, and I^2C communication module. Cubesat platform is based on MPSoCBench and also has common peripheral, such as the TLM memory. For this scenario, we change the virtual platform presented in (ZIJLSTRA, 2015) to support the new target architectures. Our focus is on the UVM-FI capabilities and not in the fault modeling. Although we applied a simple and not realistic fault model, other fault models can be design and performed for both the MPSoCBench and the Cubesat virtual platform, once this example shows that the integration with those platforms is feasible .

Memory faults can be injected directly in the simulated memory cell array or in the memory decoder circuit. In this preliminary tests, we focus on memory cell arrays in order to corrupt data by randomly or specific flipping bits to simulate overheating, radiation and other physical factors that can effect memory content. We categorized the faults in two types, as follows:

- **Bit-flip Fault:** Memory bits states are changed;
- **Coupled-Fault:** When one memory address is changed, other memory address can be

influenced. Coupling faults occur because of the mutual capacitance between cells or the current leakage from one cell to another.

In addition to the types of faults, we need to specify when the fault should be activated. A fault trigger can be based on simulation time, on events or random. At high-level models, timestamp trigger approach is not easy to monitor and control. We recommend for this type of simulation model the event triggers, where the faults are injected when a specific event occurs. For instance, the memory access event that changes a memory value when a specific memory bit is triggered.

To perform the experiments, we used a virtual platform built with an MSP430F249 processor and a TLM memory model. The virtual platform was implemented with SystemC and TLM communication. The processor Instruction Set Simulator (ISS) was implemented using the architecture description language, ArchC.

The MSP430 memory space is designed as a von-Neumann architecture (INSTRUMENTS, 2015). Figure 18 shows the MSP430 family memory map structure. The start address of ROM depends on the amount of Flash/ROM contained in each device. Besides, the ROM memory is divided in two sections. The first section represents the main code memory, including the interrupt vector memory space. The second part of ROM memory is for information and boot memory.

Figure 18 – MSP430 memory map.

Address	Region
1FFFFh – 10000h	Flash/ROM
0FFFFh – 0FFE0h	Interrupt Vector Table
0FFDFh	Flash/ROM
0200h	RAM
01FFh – 0100h	16-Bit Peripheral Modules
0FFh – 010h	8-Bit Peripheral Modules
0Fh – 0h	Special Function Registers

Source: The author. Adapted from (INSTRUMENTS, 2015).

Table 6 shows the memory map address for MSP430 processor specifically used in these experiments. We run the experiments in ROM code section that starts at 0xFFFF and ends at 0x1100. The RAM start address is 0x0200 and the end address is 0x09FF (INSTRUMENTS, 2012). The memory position from 0x0000 up to the RAM initial address is reserved for 8- and 16-Bit Peripheral Modules and Special Function Registers. We denoted this memory section as SFR/PM.

Table 6 – MSP430F249 Memory Organization.

-	-	MSP430F249 MSP430F2491
Memory	Size	60KB
Main: interrupt vector	Flash	0xFFFF to 0xFFC0
Main: code memory	Flash	0xFFFF to 0x1100
Information memory	Size	256 Byte
	Flash	0x10FF to 0x1000
Boot memory	Size	1KB
	ROM	0x0FFF to 0x0C00
RAM	Size	2KB 0x09FF to 0x0200
Peripherals	16 bit	0x01FF to 0x0100
	8 bit	0x00FF to 0x0010
	SFR	0x000F to 0x0000

Source: The author.

The target application is the Advanced Encryption Standard (AES) algorithm with a block size of 128 bits, and Cyclic Redundancy Check (CRC). Both algorithms are used in embedded applications, especially by systems that require a strong fault tolerance mechanism or high security, such as satellites or autonomous vehicles.

The experiments are performed for aleatory bit-flip and specific coupling addresses. For the bit-flip test, the faults are injected at three different memory regions: only data stored, code section and for the whole memory. The bit-flip faults are injected per program instance in a proportion of 100 and 200 bits flipped each execution. The coupling faults are triggered when a write operation occurs: 2 randomly bits are flipped in a specific space address (RAM and all memory) per write event in all memory.

Firstly, the MSP430 executes the AES algorithm to encrypt 128 bits, the first 8 bits are generated by UVM sequence and received at digital I/O for the processor. The other 120 bits and the AES key are stored as a local variable in the MSP430 memory. The output of the first 8 bits is stored in P3 output register. The algorithm is only checked the first 8 bits of encrypted data. So if the algorithm performs the encryption of the 8 bits received by digital input correctly, it succeeds; otherwise, the scoreboard asserts an error. When the system is not able to return the value due to a segmentation fault we count it as a crash of the system.

Listing 17 shows the fault condition expression for couple faults. For each write in the P1 input an event trigger is set by the TLM memory (“p1_input_event”). The faults are applied by the “rand_bit_flip” method, called in the fault model. The method responsible for the flip bits receives as arguments the number of bits to flip, the range memory addresses (from_adr and to_adr) and the TLM memory pointer (extracted from the data share mechanism). To insert coupling faults or implement new type of faults it is necessary to change the method called by the fault injector, Listing 18 demonstrates the modification.

Table 7 presents the simulation results for fault injection tests in the AES algorithm.

```
evt<sc_event*> mem_evt("p1_input_event", "*");
*engine << fm(mem_evt,
    var<tlm_memory*>("mem_test").set(call(&rand_bit_flip, cnst(100), cnst(from_adr),
        cnst(to_adr), var<tlm_memory*>("mem_test"))));
```

Source: The author.

Listing 17: Registering a fault condition for bit-flip scenario.

```
evt<sc_event*> mem_evt("men_write_event", "*");
*engine << fm(mem_evt,
    var<tlm_memory*>("mem_test").set(call(&couplin, cnst(100), cnst(from_adr),
        cnst(to_adr), var<tlm_memory*>("mem_test"))));
```

Source: The author.

Listing 18: Registering a fault condition for coupling faults.

Fault injection at RAM memory for 100 and 200 faults per program did not cause crashes on the system, although the increase of faults causes more failures on the AES results. The most significant crashes happened for modifications of the ROM section. As the ROM represents a huge amount of space in memory, the probability of system crashes increase. When faults are distributed in the whole memory, the system had an increased number of crashes and AES correct results decreased. Besides, the number of coupling faults depends on the number of writes in the memory, when the system crashes the number of writes can not be estimated as the program does not return from the execution.

Table 7 – AES algorithm results.

Fault type	Location	Number of Faults	Wrong	Right	Crash
Bit-Flip	RAM	100	52.3%	47.7%	0%
	RAM	200	78.2%	21.8%	0%
	SFR/PM	100	19.5%	80.2%	0.2%
	SFR/PM	200	68.5%	31.3%	0.4%
	ROM	100	34.2%	0%	65.8%
	ROM	200	14.1%	0%	85.9%
	ALL	100	21.5%	9%	77.6%
	ALL	200	12.3%	0%	87.7%
Coupling	RAM	1244*	7.9%	92.1%	0%
	ALL	-*	4%	0%	96.0%

* The number of faults depends on the execution, the table shows only the number of faults when the execution gave a correct result. In case of no successful results, the number of faults is not estimated.

Source: The author.

The second fault injection campaign made with the CRC algorithms was designed

in the same way as the AES experiment. The CRC receives as input two 8-bit numbers, the MSP430 receives these numbers as a digital input on through ports P1 and P2, the output is a unique 8-bits value and is stored in P3 register.

For the CRC, since the algorithm is not using many local variables and the amount of RAM data space is not that large, it is more difficult for a bit-flip fault to cause a failure, as shown in Table 8. However, when we use more ports for data input, the bit-flip faults in digital I/O ports cause an increase in incorrect CRC results.

Table 8 – CRC algorithm results.

Fault type	**Location**	**Number of Faults**	**Wrong**	**Right**	**Crash**
Bit-Flip	RAM	100	0%	100%	0%
	RAM	200	0%	100%	0%
	SFR/PM	100	85.5%	14.3%	0.2%
	SFR/PM	200	94.3%	5.4%	0.3%
	ROM	100	8.3%	85.4%	6.3%
	ROM	200	28.1%	62.2%	9.7%
	ALL	100	13.9%	77.3%	8.8%
	ALL	200	20.3%	60.2%	19.5%
Coupling	RAM	284*	0%	100%	0%
	ALL	284*	4.6%	88.5%	6.9%

* The number of faults depends of the execution, the table show only the number of faults when the execution gave a right result.

Source: The author.

The coupling faults depend on the execution flow of each algorithm. In both cases, the coupling fault injector algorithm introduces more faults than we test for bit-flip, and the AES and CRC algorithms result in more hits than the bit-flip. We only test for one specific coupling faults and this result can be better adapted to more accurate trigger events.

Table 9 shows the simulation performance of coupling fault injection example in virtual platform. Our fault injection does not result in significant overhead for this simulations. The injector works based on events notification and it only actives the fault function, in this case "coupling" method. Most of our examples cannot compare time because it simulations run too fast hampering accurate run-time measures and thus, run time comparisons.

Table 9 – UVM and UVM-FI simulation time for coupling fault execution.

Experiment	**UVM Sim. time (ns)**	**UVM-Fi Sim. time (ns)**
AES	69245000	69366900
CRC	14925000	14925001

Source: The author.

The results show what happens with AES and CRC algorithms in the presence of memory faults for MSP430 microcontroller. Both algorithms have failures and the designers have to consider using fault-tolerant mechanisms, especially in safety-critical applications. More fault models can be implemented based on our approach to test other systems, even for systems designed without an ISS or for another level of abstraction, such as SystemC RTL models.

With the modular UVM-FI, the created Testbench components could be used in both examples and to perform different fault models just the base fault injection component (extended from our UVM-FI) was modified, showed in Listing 17 and Listing 18. The UVM-FI extends from the UVM the capacity to create modular and reusable Testbench to the perform fault injection. Moreover, another types of faults can be easy created just by modifying the function called in our fault model. The fault type method is the Testbench level and it is not necessary to change the UVM, SystemC or any library.

6 CONCLUSION

This Master work proposed, validated and evaluated a new non-intrusive fault injection technique named UVM-FI that is fully integrated with UVM. UVM-FI consists of an open-source library, written in C++/SystemC, containing basic fault types, triggers and fault locations that can be integrated into a UVM environment using the plug-and-play principle. The integration of UVM-FI into UVM relies on the UVM data share mechanism and enables reusability. The case studies validated this new fault injector and demonstrated its capability, allowing a comparison with other approaches. Faults can be injected in DUT public variables and signals, as shown in Chapter 5. The UVM-FI can be easily connected to UVM environments. As shown in Chapter 4, the addition of the fault module component does not require modifications in the Testbench.

Moreover, the UVM-FI has an efficient fault models description with the SystemC DSL. The case studies developed in this work demonstrated that the DSL is less verbose and more readable than the other XML-based approaches. Notwithstanding, this work also evaluated the same fault injector framework with and without the DSL, and demonstrated that the DSL is more usable than directly creating C++ objects that describe a fault model. The DSL improves the fault injection tests, providing a better way to describe the fault model. This approach contributes to improving system dependability evaluation and system verification. The tests performed on case studies demonstrate the intuitiveness and effectiveness of our approach.

Although the various advantages, the proposed technique presents some limitations. Template metaprogramming is widely known to provide additional strain to the compiler. Another possible limitation is that our approach uses polymorphism relying on heap memory (new operator). However, as the Testbench typically runs on high-end PC systems instead of embedded systems, these characteristics have negligible effect. Furthermore, our approach requires no changes to the DUT and therefore, the same code that yields the synthesized final system can be tested on the Testbench. However, if for any reason the Testbench is required to run in an embedded system, the use of the DSL should be avoided.

6.1 SUMMARY OF THE RESEARCH ACHIEVEMENTS

During the development of the software related to this Master work, two papers were published, as summarized in Appendix A. The first paper describes a simple UVM component developed to allow the fault injection and its use to build UVM Testbenches. The second paper describes the proposed Domain-Specific Language (DSL) targeting the creation of compact and readable fault model descriptions. Both contributions were essencial for this work to achieve the final status described in this dissertation. The next paragraphs briefly discuss both of them.

The lack of available techniques that perform system dependability tests associated with Testbench generation methodologies motivated the development of the first UVM-FI version, which was presented in (LOHMANN et al., 2018b). That paper also presented the vali-

dation of such approach that served as basis for this Master work. Starting from the standard UVM environment, the paper shows a fault injector component and a monitor to analyze the injected faults. The fault injector class was extended from the *uvm_component* class. This class provides interfaces for hierarchy, phasing, factory, process control reporting and recording. Although the fault injection is not part of the UVM flow, this extension allows for easy control of fault injection and synchronization of tests with the UVM environment. The paper explores the UVM data share mechanism that allows for storing data pointers from the DUT and restore them in the fault injector component.

The extension of UVM for fault injection was successfully applied to the first implementation of a hybrid fault-verification methodology to the following algorithms: Advanced Encryption Standard (AES) and Cyclic Redundancy Check (CRC). In those cases, memory failures were simulated in a virtual platform developed for the MSP430 microcontroller Instruction Set Simulator (ISS) and a TLM memory model. The results showed that the fault injection can be applied with UVM extension (LOHMANN et al., 2018b). However, the fault injector integration was not a plug-and-play implementation, that is, it was not optimized to generate the fault injector models. In addition, it required manual effort to do it.

In order to increase the compatibility between the UVM and the fault injection, this Master work further improved the library implementation, so as to reduce the effort to integrate the fault injection into the UVM. The implementation is based on an Event-Condition-Action (ECA) engine that receives fault models from the test engineer and manages the fault injection. The framework was baptized as UVM-FI.

Fault model description is an important issue in the existing fault tools that use SystemC. Many fault injection tools are based on the Extensible Markup Language (XML) or use a run-time console. The XML description of fault models is verbose, making it difficult to setup system verification under fault injection. To overcome this issue, the second publication (LOHMANN et al., 2018a) proposes a SystemC template metaprogrammed Domain-Specific Language(DSL) that helps designers to create fault models in SystemC, reducing programming effort and taking advantage of SystemC/C++ expressiveness. The fault model description with DSL improvements were also presented in that publication.

6.2 FUTURE WORK

The UVM-FI supports perfectly the most commonly used fault models, but new DUT locations can be supported by the fault injector, which can make the UVM-FI even more versatile. In the current version, the majority of the faults were injected in public signals, variables, and *sc_in* input ports. New methods to access and drive other types of ports can be explored. Besides that, TLM channels can be easily included in UVM-FI.

The DSL can be improved at the automation of the DUT variables insertion in the UVM database, and the ability to verify at compile time if all variables used in a DSL expression were inserted in the database.

Furthermore, more experiments with dense fault models should be conducted in order to provide a comparison in terms of simulation time. Exploration of such measure allows optimizations in our library and also other related work comparisons. Although our project is built with CMake [1] tool, there is not an automated unit test or continues integration, these features can be included in the library to improve the library code quality and prevent further problems.

Also, Accellera provides a UVM in SystemVerilog that aims to be compatible with UVM-SystemC. Due to some reserved keywords in C++, a few functions and methods differ. The UVM-FI can be extended for SystemVerilog without much effort and thus, the same concepts of this Master work can be used for SystemVerilog.

Finally, more studies should be conducted in order to investigate methods to extract SystemC DUT information and create automated fault injection executions. For example, a system that receives a DUT and returns possible faulty points or maybe returns the DSL fault model ready to be performed. Modules for fault coverage and statics can be included as well. With these features, the dependability analysis is complete from the design specification up to the automated dependability analysis.

[1] https://cmake.org/

BIBLIOGRAPHY

AARNO, D.; ENGBLOM, J. **Software and System Development using Virtual Platforms: Full-System Simulation with Wind River Simics**. [S.l.]: Morgan Kaufmann, 2014.

ABRAHAMS, D.; GURTOVOY, A. **C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)**. [S.l.]: Addison-Wesley Professional, 2004. ISBN 0321227255.

ALBERTINI, B.; RIGO, S.; ARAUJO, G. Computational reflection and its application to platform verification. **Design Automation for Embedded Systems**, Springer, v. 16, n. 1, p. 1–17, 2012.

ARLAT, J. et al. MEFISTO: a series of prototype tools for fault injection into VHDL models. In: **Fault injection techniques and tools for embedded systems reliability evaluation**. [S.l.]: Springer, 2003. p. 177–193.

ASSOCIATION, I. S. et al. IEEE Standard for standard SystemC language reference manual. **IEEE Computer Society**, 2012.

AVIZIENIS, A. et al. Basic concepts and taxonomy of dependable and secure computing. **IEEE transactions on dependable and secure computing**, IEEE, v. 1, n. 1, p. 11–33, 2004.

BELTRAME, G. et al. ReSP: A Non-Intrusive Transaction-Level Reflective MPSoC Simulation Platform for Design Space Exploration. **inProc. ASPDAC**, IEEE, p. 673–678, 2008.

BENSO, A.; PRINETTO, P. **Fault injection techniques and tools for embedded systems reliability evaluation**. [S.l.]: Springer Science & Business Media, 2003. v. 23.

BERGERON, J. **Writing Testbenches**: Function verification of hdl models. Second edition. New York, NY - USA: Springer, 2003. ISBN 1-4020-7401-8.

BLACK, D. C. et al. **SystemC: From the ground up**. [S.l.]: Springer Science & Business Media, 2009. v. 71.

BOLCHINI, C.; MIELE, A.; SCIUTO, D. Fault models and injection strategies in systemc specifications. In: IEEE. **Digital System Design Architectures, Methods and Tools, 2008. DSD'08. 11th EUROMICRO Conference on**. [S.l.], 2008. p. 88–95.

BUSHNELL, M.; AGRAWAL, V. **Essentials of electronic testing for digital, memory and mixed-signal VLSI circuits**. [S.l.]: Springer Science & Business Media, 2004. v. 17.

CALLAHAN, S. (Ed.). **Adrift: Seventy-Six Days Lost at Sea**. 1. ed. [S.l.]: Houghton Mifflin Harcourt, 2002. ISBN 0618257322.

CHANDRAN, P. et al. Parallelizing SystemC kernel for fast hardware simulation on SMP machines. In: IEEE COMPUTER SOCIETY. **Proceedings of the 2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation**. [S.l.], 2009. p. 80–87.

CHANG, K.-J.; CHEN, Y.-Y. System-level fault injection in SystemC design platform. In: **2007 Proc. 8th Int. Symposium on Advanced Intelligent Systems**. [S.l.: s.n.], 2007. p. 354–359.

COOPER, V.; MARRIOTT, P. Demystifying the uvm configuration database. **DVCon 2014**, 2014.

GAJSKI, D. D. et al. **Embedded system design: modeling, synthesis and verification**. [S.l.]: Springer Science & Business Media, 2009.

GARANHANI, L. D. D. MPSoCBench: a framework for high-level evaluation of multiprocessor system-on-chip tools and methodologies. 2015.

GRACIA, J. et al. Comparison and application of different VHDL-based fault injection techniques. In: IEEE. **Defect and Fault Tolerance in VLSI Systems, 2001. Proceedings. 2001 IEEE International Symposium on**. [S.l.], 2001. p. 233–241.

GROSSE, D.; DRECHSLER, R. **Quality-Driven SystemC Design**. [S.l.]: Springer, 2010.

GUTHAUS, M. R. et al. MiBench: A free, commercially representative embedded benchmark suite. In: IEEE. **Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on**. [S.l.], 2001. p. 3–14.

HSUEH, M.-C.; TSAI, T. K.; IYER, R. K. Fault Injection Techniques and Tools. **Computer**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 30, n. 4, p. 75–82, abr. 1997. ISSN 0018-9162.

HUDAK, P. Modular domain specific languages and tools. p. 134–142, Jun 1998. ISSN 1085-9098.

INSTRUMENTS, T. MSP430f249 Family Datasheet. **Texas Instruments, 2012.**, 2012. Disponível em: http://www.ti.com/lit/ds/symlink/msp430f249.pdf.

INSTRUMENTS, T. MSP430 Family User Guide. **Texas Instruments, 2015a.**, 2015. Disponível em: http://www.ti.com/lit/ug/slau144j/slau144j.pdf.

LAPRIE, J.-C. Dependability: Basic concepts and terminology. In: **Dependability: Basic Concepts and Terminology**. [S.l.]: Springer, 1992. p. 3–245.

LISHERNESS, P.; CHENG, K.-T. T. SCEMIT: A SystemC error and mutation injection tool. In: ACM. **Proceedings of the 47th Design Automation Conference**. [S.l.], 2010. p. 228–233.

LOHMANN, D. et al. A Domain-specific Language for Automated Fault Injection in SystemC Models. In: **25th IEEE International Conference on Electronics Circuits & Systems (ICECS)**. [S.l.]: IEEE, 2018.

LOHMANN, D. et al. Extending Universal Verification Methodology with Fault Injection Capabilities. In: **9th Latin American Symposium on Circuits & Systems (LASCAS)**. [S.l.]: IEEE, 2018.

LU, W.; RADETZKI, M. Efficient fault simulation of SystemC designs. In: IEEE. **Digital System Design (DSD), 2011 14th Euromicro Conference on**. [S.l.], 2011. p. 487–494.

LU, W.; RADETZKI, M. Concurrent and comparative fault simulation in SystemC and its application in robustness evaluation. **Microprocessors and Microsystems**, Elsevier, v. 37, n. 2, p. 115–128, 2013.

MARTIN, G.; BAILEY, B.; PIZIALI, A. ESL Design and Verification: A Prescription for Electronic System Level Methodology. Morgan Kaufmann Publishers Inc., 2007.

MICHAEL, M.; GROSSE, D.; DRECHSLER, R. Analyzing dependability measures at the Electronic System Level. In: IEEE. **Specification and Design Languages (FDL), 2011 Forum on**. [S.l.], 2011. p. 1–8.

MISERA, S.; VIERHAUS, H. T.; SIEBER, A. Fault injection techniques and their accelerated simulation in SystemC. In: IEEE. **Digital System Design Architectures, Methods and Tools, 2007. DSD 2007. 10th Euromicro Conference on**. [S.l.], 2007. p. 587–595.

RIGO, S.; AZEVEDO, R.; SANTOS, L. **Electronic system level design: an open-source approach**. [S.l.]: Springer Science & Business Media, 2011.

ROISER, S.; MATO, P. The SEAL C++ reflection system. CERN, 2005.

ROSENBERG, S.; KATHLEEN, M. **A Practical Guide to Adopting the Universal Verification Methodology (UVM) Second Edition**. [S.l.]: Cadence Design Systems, Inc., 2013. ISBN 9781300535935.

ROTHBART, K. et al. High level fault injection for attack simulation in smart cards. In: IEEE. **Test Symposium, 2004. 13th Asian**. [S.l.], 2004. p. 118–121.

SHAFIK, R. A.; ROSINGER, P.; AL-HASHIMI, B. M. SystemC-based minimum intrusive fault injection technique with improved fault representation. In: IEEE. **On-Line Testing Symposium, 2008. IOLTS'08. 14th IEEE International**. [S.l.], 2008. p. 99–104.

SIEH, V.; TSCHACHE, O.; BALBACH, F. VERIFY: Evaluation of reliability using VHDL-models with embedded fault descriptions. In: IEEE. **Fault-Tolerant Computing, 1997. FTCS-27. Digest of Papers., Twenty-Seventh Annual International Symposium on**. [S.l.], 1997. p. 32–36.

VELDHUIZEN, T. L. C++ templates are turing complete. 2003.

VENTROUX, N.; SASSOLAS, T. A new parallel SystemC kernel leveraging manycore architectures. In: EDA CONSORTIUM. **Proceedings of the 2016 Conference on Design, Automation & Test in Europe**. [S.l.], 2016. p. 487–492.

YAN, W. et al. A design flow with integrated verification of requirements and faults in safety-critical systems. In: IEEE. **System of Systems Engineering Conference (SoSE), 2017 12th**. [S.l.], 2017. p. 1–6.

ZIJLSTRA, D. Virtual satellite platform of on-board computers for space applications. 2015.

Appendix

APPENDIX A – PUBLICATIONS

A.1 PUBLISHED PAPERS DIRECTLY RELATED TO THE DISSERTATION SUBJECT

The proposal evaluation resulted in publications directly related to this dissertation subject. This Section describes the details of each publication.

A.1.1 *2018 IEEE 9th Latin American Symposium on Circuits & Systems (LASCAS)*

- **Qualis CC 2016:** B2
- **Title:** *Extending Universal Verification Methodology with Fault Injection Capabilities*
- **Authors:** Douglas Lohmann, Fabrizio Maziero, Elço João dos Santos Jr, and Djones Lettnin
- **DOI:** https://doi.org/10.1109/LASCAS.2018.8399945
- ***Abstract:*** *Embedded Systems verification has become a major challenge in recent years due to the increased hardware and software complexity and to the shorter time to market. In order to overcome these issues, new verification methodologies and fault injection techniques are strongly recommended to increase safety quality of complex systems. In this paper, we propose an integration methodology to extend the Universal Verification Methodology (UVM) with fault injection capabilities. For that, we extend the UVM components and use the UVM's data share resources to manipulate data. We have successfully applied this optimized hybrid fault-verification methodology to the algorithms: Advanced Encryption Standard (AES) and Cyclic Redundancy Check (CRC). Memory failures were simulated in a virtual platform developed for MSP430 microcontroller Instruction Set Simulator (ISS) and a TLM memory model. The results show that our approach scales to increase the system's dependability validation creating reusable Testbenches with fault injection test in UVM.*

A.1.2 *2018 25th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*

- **Qualis CC 2016:** B1
- **Title:** *A Domain-specific Language for Automated Fault Injection in SystemC Models*
- **Authors:** Douglas Lohmann, Alexis Huf, Djones Lettnin, Frank Siqueira and José Luís Güntzel
- **DOI:** https://doi.org/10.1109/ICECS.2018.8617838

- ***Abstract:*** *With the evolution of technology, electronic systems have become significantly more complex. As a consequence, design and verification of these systems evolved notably. Fault injection is a dependability evaluation technique that is strongly recommend during the verification step. Although there are a number of tools capable of injecting faults, many of them do not have a simple fault model description language and require considerable manual effort. In this paper, we propose a SystemC template metaprogrammed Domain-Specific Language (DSL) integrated with Universal Verification Methodology (UVM) to describe formal fault models that requires neither specific compilers nor code preprocessing tools. Unlike current approaches for fault injection, there is no need to create fault injection environment manually or to describe the system in an XML format. We evaluate our approach in terms of readability and effort required from a designer to describe a fault injection test. Our case study illustrates how the DSL helps designers to create fault models in SystemC, decreasing programming effort and taking advantage of SystemC/C++ expressiveness.*

APPENDIX B – DSL

The following tables lists the SystemC DSL capability. Table 10 shows the DSL operators that are overloaded from C++. Table 11 presents the functions provided by the DSL to help engineers.

Table 10 – UVM-FI SystemC DSL operators.

<table>
<tr><th>Operator</th><th>Description</th><th>UVM-FI representation</th></tr>
<tr><td>></td><td>greater than</td><td>uvm_gt</td></tr>
<tr><td>==</td><td>equals</td><td>uvm_eq</td></tr>
<tr><td>!=</td><td>different</td><td>uvm_neq</td></tr>
<tr><td>>=</td><td>greater than or equal to</td><td>uvm_gte</td></tr>
<tr><td><</td><td>less than</td><td>uvm_lt</td></tr>
<tr><td><=</td><td>greater than or equal to</td><td>uvm_lte</td></tr>
<tr><td>&&</td><td>AND</td><td>uvm_and</td></tr>
<tr><td>||</td><td>OR</td><td>uvm_or</td></tr>
<tr><td>&</td><td>logical AND</td><td>uvm_band</td></tr>
<tr><td>|</td><td>logical OR</td><td>uvm_bor</td></tr>
<tr><td>^</td><td>XOR</td><td>uvm_xor</td></tr>
<tr><td>+</td><td>plus</td><td>uvm_plus</td></tr>
<tr><td>-</td><td>minus</td><td>uvm_minus</td></tr>
<tr><td>/</td><td>division</td><td>uvm_divide</td></tr>
<tr><td>*</td><td>multiplication</td><td>uvm_mult</td></tr>
</table>

Source: The author.

Table 11 – UVM-FI helper functions provided by DSL.

Operator	Description
call(f, a1, ...)	returns of function `f` if called with arguments `a1, ...`
cap(x)	binds a C++ variable `x` by reference
evt(x)	represents a event `x` for ECA
fm(condition, injector)	creates a fault model with the `condition` trigger and the `injector` function
unif(a, b)	calculates the random value uniformly distributed between `a` and `b`
var(x)	represents a variable `x` for ECA

Source: The author.