UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

LUCAS AUGUSTO TANSINI

Model-based Design Code Generator
Effects on Codes Reliability

Porto Alegre
2021

LUCAS AUGUSTO TANSINI

Model-based Design Code Generator Effects on Codes Reliability

Work presented in partial fulfillment of the requirements for the degree of Bachelor in Computer Engineering

Advisor: Prof. Dr. Paolo Rech

Porto Alegre
2021

CIP — CATALOGING-IN-PUBLICATION

Tansini, Lucas Augusto

Model-based Design Code Generator Effects on Codes Reliability / Lucas Augusto Tansini. – Porto Alegre: 2021.

39 f.

Advisor: Paolo Rech

Trabalho de conclusão de curso

– Universidade Federal do Rio Grande do Sul, Escola de Engenharia. Curso de Engenharia de Computação, Porto Alegre, BR–RS, 2021.

1. Fault injection. 2. Simulink. 3. Scade. 4. DO-178. 5. Algorithms. 6. Safety-critical. 7. Embedded. 8. Model-based. I. Rech, Paolo, orient. II. Título.

Dedico este trabalho a meu pai e minha mãe. Por toda ajuda durante este período de graduação.

ACKNOWLEDGEMENTS

To start, I'd like to thank my family for all the support I've had to arrive here. Especially to my father, mother, sisters, and Carolina, who were present throughout my entire graduation and immensely supported me.

I'm also profoundly grateful for all the help that my advisor gave me during all this time, whether it's with articles, life advice, or even when teaching me a lot about the fault tolerance world.

I can not forget to thank all my dear colleagues that were with me during all this journey - which also came along with me until the end. Thanks a lot for the uncountable programming nights, cheap talking, and, also, for supporting me in various moments of my graduation.

During this period, I can not forget to thanks all my colleagues from the Junior Enterprise (IDE), all my colleagues from the Embedded systems laboratory (LSE), and also all the folks from AEL Sistemas, who helped me a lot.

AGRADECIMENTOS

Gostaria de começar agradecendo por todo apoio que obtive para que pudesse chegar até aqui, onde, principalmente minha família fez grande parte. Em especial ao meu pai, minha mãe, irmãs e Carolina que sempre estiveram comigo durante o período da faculdade e que muito me apoiaram.

Sou imensamente grato por toda ajuda que meu orientador Paolo Rech teve comigo durante todo esse tempo, sendo ajudando a escrever artigos, dando conselhos sobre a vida e também me ensinando muito sobre o vasto mundo de tolerância a falhas.

Não posso deixar de agradecer também, durante toda a minha caminhada, aos meus queridos colegas que ingressaram comigo e chegam, hoje, até o final junto comigo. Muito obrigado pelas inúmeras noites programando, jogando conversa fora e, também, por terem me apoiado em diversos momentos da faculdade.

Durante minha caminhada pela graduação, não posso deixar de agradecer a todos os colegas da Empresa Júnior da Computação (IDE), todos os colegas do laboratório de sistemas embarcados (LSE) e também a todos amigos e colegas da AEL Sistemas, que muito me ajudaram.

Je pense, donc je suis.

— RENÉ DESCARTES, LE DISCOURS DE LA MÉTHODE

ABSTRACT

Modern embedded safety-critical applications are utilizing tools to help the software development deal with safety-critical guidelines. Simulink and Scade are examples of these tools, often used to design flight control, engine control, automatic pilots, and fuel management systems. To generate a code, first, the software is modeled and, then, the models are translated into an automatic generated C code. Unlike the Simulink tool, Scade's resulting C code is guaranteed to be compliant with safety-critical regulations.

This work evaluates the impact of the use of code generation tools on the overall code reliability. Each tool generates the code based on specific directives, with possibly a significant impact on the code sensitivity to transient faults. Four different algorithms are considered, and each one is implemented in three different versions: Manual, Simulink generated and Scade generated. To evaluate the version's reliability, more than 3,500 faults were injected into the programs. Results show that, while increasing the execution time, Simulink reduces, on average, 79% the SDC rate and 61% the DUE rate. Scade reduces the SDC rate of 52% but, unfortunately, increases the DUE rate of 5%.

Keywords: Fault injection. simulink. Scade. DO-178. algorithms. safety-critical. embedded. model-based.

Efeitos de geradores de códigos baseados em modelos na confiabilidade de códigos

RESUMO

Aplicações modernas de sistemas críticos utilizam cada vez mais ferramentas para auxiliar o desenvolvimento de *software*, que ajudam a lidar com guias de desenvolvimento de *software* em sistemas críticos. Simulink e Scade são exemplos dessas ferramentas, uma vez que são frequentemente utilizadas para realizar o *design* de projetos como: controladores de vôo, controladores de motores, pilotos automáticos e também sistemas que gerenciam abastecimento de combustível. Para gerar o código, primeiramente o *software* é modelado, e, depois, traduzido para um código gerado automaticamente na linguagem de programação C. Diferente da ferramenta Simulink, o código gerado pela ferramenta Scade é garantido a seguir os guias determinados pelas regulamentações. Este trabalho avalia o impacto do uso de códigos gerados por ferramentas em algoritmos. Cada ferramente gera o código automaticamente, baseado em certas diretivas, particulares a cada ferramenta, tendo assim um possível impacto significativo na tolerância a falhas dos algoritmos. Quatro algoritmos diferentes são considerados, e cada um é implementado em três diferentes versões: Manual, Simulink e Scade. Para avaliar as confiabilidade dos diferentes códigos em suas diferentes versões, injetou-se mais de 3.500 falhas nos programas. Os resultados mostram, que, enquanto a ferramenta Simulink reduz, em média, 79% a taxa de SDCs e 61% a taxa de DUEs. Enquanto isso, a ferramenta Scade reduz a taxa de SDCs em 52%, mas, infelizmente, aumenta a taxa de DUEs em 5%.

Palavras-chave: injeção de falhas, simulink, Scade, DO-178, algoritmos, sistemas críticos, embarcado, baseado em modelo.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

FAA	Federal Aviation Administration
ANAC	National Civil Aviation Agency - Brazil
DAL	Design Assurance Level
MBD	Model-Based Design
PSAC	Plan for Software Aspects of Certification
SWDP	Software Development Plan
SWVP	Software Verification Plan
SETs	Single Event Transients
SDC	Silent Data Corruption
DUE	Detected Unrecoverable Error
IFFT	Inverse Fast Fourier Transform
FFT	Fast Fourier transform
PVF	Program Vulnerability Factor
FIT	Failure in Time

CONTENTS

1 INTRODUCTION

Modern safety-critical systems are commonly highly complex and require extensive documentation to have their reliability qualified. Several regulatory organs like the FAA (Federal Aviation Administration) and ANAC (National Civil Aviation Agency - Brazil), to qualify a product as sufficiently reliable, often require that companies follow certain guidelines and standards, such as the DO-178B/C (SC-205 RTCA, 2011), which highly suggests that a project implements the DO's objectives to guide and help the management of the system's life cycle and development.

Typically, the DO-178 classifies software according to their criticality level, defined as the Design Assurance Level (DAL). In each DAL, the software is classified according to its failure conditions and is also categorized by the effects of these conditions on the aircraft, passengers, and crew. Therefore, the number of objectives that the project must follow in the guideline is dependent on the DAL, as depicted in Table 1.1. Usually, military and civil aviation projects are categorized with DAL A and DAL B.

Table 1.1 – DO-178B Failure Conditions, objectives and Failure Rate for each level.

Level	Failure Conditions	Objectives	Failure Rate
A	Catastrophic	66	$10^{-9}/h$
B	Hazardous	65	$10^{-7}/h$
C	Major	57	$10^{-5}/h$
D	Minor	28	$10^{-3}/h$
E	No Effect	0	N/A

Source: DO-178

In general, projects begin the life cycle with the understanding of the contract, which results in a list of system requirements and, then, each requirement is translated into constraints for the hardware and/or software to be designed and developed. In particular, software documentation for safety-critical systems that utilize such guidelines requires several artifacts for the completion of the objectives, such as test procedures, test execution reports, test results, static code analysis, and structural coverage analysis. These artifacts need to be traced from the system requirements to the software implementation (source code).

The traditional process of producing all the artifacts in each phase of the project is extensive and complex. Hence, Model-Based Design (MBD) tools are becoming more attractive to companies. MBD tools are a modern and efficient alternative to achieve the required objectives in the development phase.

The use of different code generators in embedded software can have a direct impact on sensitivity to transient faults, for better or for worse. To guarantee to be compliant with specific constraints, the tool can significantly modify the source code (e.g., removing all loops or all indexes). The code is generated, then, will be compliant with the regulation but might have a different sensitivity to transient faults w.r.t. the traditional implementation. Unfortunately, while the compiler effects on various applications have already been observed in literature (F. M. LINS L. A. TAMBARA; RECH, 2017; J. GAVA V. BANDEIRA; OST, 2019), the impact of model-based design tools in embedded code reliability is still largely unclear.

The goal of this work is to evaluate and understand if, and how, MBD tools impact the transient faults sensitivity of different codes. The interest in the reliability impact of MBD tools grows significantly when dealing with aerospace applications. In fact, on one side aerospace markets require the highest reliability standards, thus impose the use of strict MBD tools. On the other side, as aerospace applications operate at high altitudes, their levels of radiation are much higher than at the earth's surface. This makes the applications more susceptible to radiation effects (BAUMANN, 2005).

This work has the purpose of analyzing the effects of code generation tools used in safety-critical applications industries on the reliability of the generated codes. Four different algorithms are considered. For each algorithm, the automatically generated code (with Scade and Simulink) will be compared to the manually written code (C implementation). To evaluate the sensitivity to transient fault of the different implementations of the codes we use a software fault injector (CAROL-FI), designed at INF-UFRGS (OLIVEIRA; PILLA *et al.*, 2017a). We inject more than 40,000 faults across all algorithms and see that the generated code from the Scade tool is 5% more susceptible to crashes and hangs, probably because the generated code utilizes pointers throughout most function calls. Both the Scade and Simulink version are shown to be less susceptible to experience silent data corruptions with respect to the manually generated code (52% and 79%, respectively). We also observe that the Simulink version of the algorithms, in particular, tends to mask the majority of the injected faults.

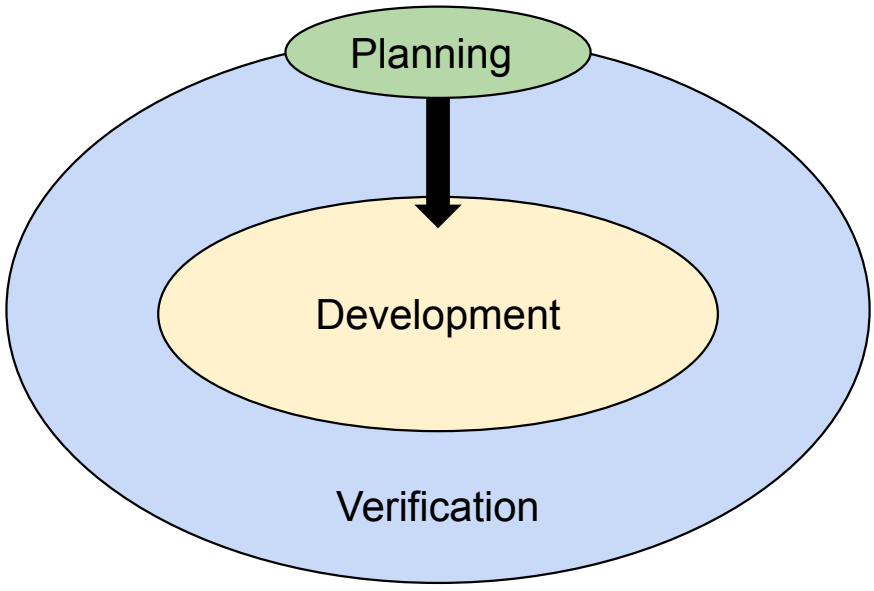
2 BACKGROUND INFORMATION

2.1 DO-178

The DO-178, which has a title of "Software Considerations in Airborne Systems and Equipment Certification", first developed by the ’80s, is considered a friendly standard that was written by technical personnel, which guides the development of airborne software. The DO itself avoids telling software developers "how" to develop the software, instead, it focuses on "what" is required to do so.

Overall, projects that follow the DO-178 must, first, classify their software according to their criticality level (DAL). After the classification is made, the project’s managers and developers must establish exactly which objectives must be satisfied for the software that will be produced. In general, the DO-178 can be separated into three parts: Planning, development, and verification. Even though the development process comes after the planning phase, the verification phase must be present throughout all the development phases, as depicted in Figure 2.1.

Figure 2.1 – Overall development parts throughout DO-178 software development.



Source: Image provided by author.

The planning process of the DO-178 consists of planning all the phases and defining plans and standards to be utilized throughout the development of the project. In this phase, documents like the Plan for Software Aspects of Certification (PSAC), Software Development Plan (SWDP), Software Verification Plan (SWVP), and other documents

Table 2.1 – Sample of DO-178 objectives with their corresponding DAL.

Objective	Applicability by DAL
Source code complies with low-level requirements	A,B,C
Source code complies with software architecture	A,B,C
Source code is verifiable	A,B
Source code conforms to standards	A,B,C
Output of software integration process is complete and correct	A,B,C

must be written. Each one has its key components and relevant actions to the project itself. Along with all the plans, standards like the Software Design Standard and Software Coding Standard must be created also.

After all the planning phase is done, the project can start the development phase, which comes with the verification phase as well. In this phase, all the software is developed, along with testing, verifications, and analysis. Within all three phases, according to the DAL of the project, the project must accomplish the necessary objects that the DO specifies (some of them are depicted in the Table 2.1).

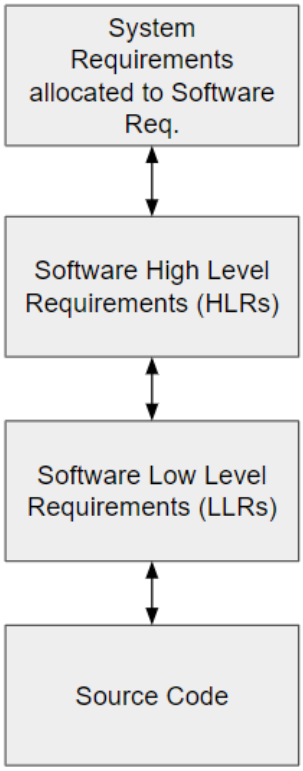
So, overall, the DO-178 guides projects to be reliable and perform well in an airborne environment, requiring extensive documentation and the accomplishment of several objects (up to 66 objects in DAL A projects).

2.2 Safety Critical Software Development

In the traditional development of safety critical embedded applications (depicted in Figure 2.2), all the product life cycle is developed manually, with little help of modern development tools. The manual development of software applications can be prone to errors, since software developers can introduce bugs in the project/development phases, such as in the source code development.

These errors could result in reworking some phases of the project, for example, in the development of Low Level Requirements and source code. Hence, MBD tools, such as Scade (SCADE..., 2021 (accessed January 7, 2021)) and Simulink (SIMULINK..., 2021 (accessed January 7, 2021)), were introduced to allow companies and researchers to design models that can be later translated to source code (ANICULAESEI; VORWALD; RAUSCH, 2019). This translation is done with automatic code generators, as depicted in Figure 2.3. The usage of models and automatic code generation can significantly facilitate

Figure 2.2 – Simplified required tracing between certification artifacts for DO-178B/C



Source: Image provided by author.

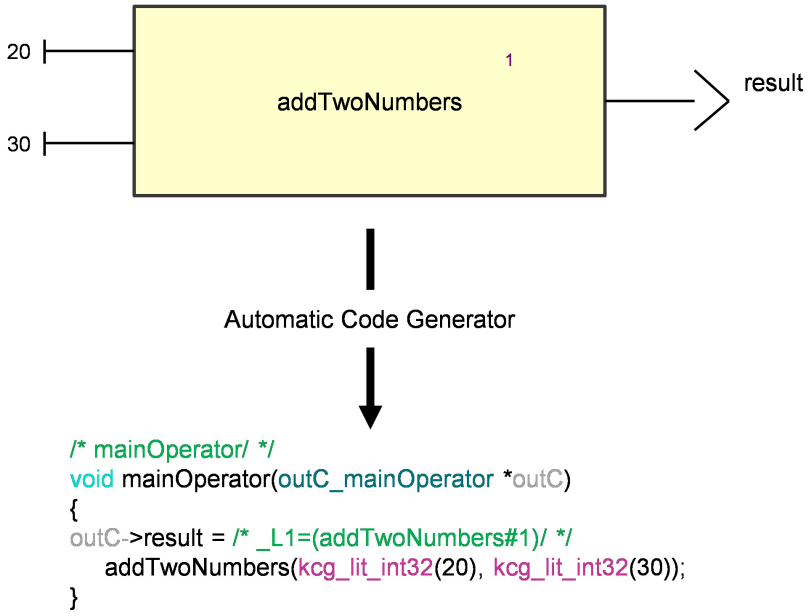
the development of the projects, since the traceability between artifacts (BOUALI; DION, 2005), prototyping and testing can be significantly reduced in time.

In particular, Scade is a certified tool and follows embedded code constrains such as static memory allocation, the removal of recursion and static bounded loops. Scade can therefore be adopted in projects that needs to follow DO-178B/C or other very strict standards. Simulink, while not yet certified for projects that follow the DO-178B/C standards, is a tool that can be utilized to facilitate the development of the project, similarly to Scade - modeling and generating source code. The source code then, can be manually certified against the DO's objectives by software developers.

2.3 Model-Based Design tools

Model-based design tools can help safety-critical projects to reduce their development time, take credit from models as artifacts, test early and often and the most important - generate automatic code. When utilizing automatic code generators, the source code that is generated ends up being highly unfriendly for the programmer to read, since variables are often re-used and have non-intuitive names. Also, the generated code usually has more

Figure 2.3 – Translation of a model to source code in Scade



Source: Image provided by author.

lines of code than the manual version, since the tools will add more logic to the generated code. These characteristics are observed both in Scade and Simulink generated codes, and occur even more as the complexity of the model increases. Figure 2.4 depicts a small piece of code for the Inverse Fast Fourier Transform algorithm, automatically generated by Scade. The variable **tmp** ends up being reused, as seen in lines number 8 and 22.

In general, each MBD tool has intrinsic and unique characteristics in its code generator. When considering the generated code from the Scade, it is possible to see that the tool generates a type-specific global variable that is utilized as a context for the whole program. This variable, so-called the program's context, holds information of all steps of the entire algorithm, such as accumulators, indexes, and even some auxiliary variables that are utilized in the program's calculation. The context is also passed by as a pointer throughout several function calls. Figure 2.5 depicts the generated code that defines the type of the global context variable.

When considering the generated code from the Simulink tool, it is possible to notice that the tool also utilizes global variables to hold information about the algorithm. Simulink, however, can generate one or more of these variables, as depicted in Figure 2.6. Each one of these variables is utilized in specific parts of the code, for example, holding states of the algorithm and also signals that are utilized in sub-calculations.

Figure 2.4 – Scade automatic generated code example for a Inverse Fast Fourier Transform algorithm.

```
1       .
2       .
3   for (idx = 0; idx < 9; idx++) {
4       /* _L55=(numberOfBitsNeeded#3)/ */
5       numberOfBitsNeeded(
6         /* _L55= */(kcg_uint32) idx,
7         numSamples,
8         &tmp,
9         &_2_noname[idx]);
10      _L55 = /* _L55= */(kcg_uint32) (idx + 1);
11      /* _L55= */
12      if (!tmp) {
13        break;
14      }
15    }
16    /* _L60= */
17    for (idx = 0; idx < 512; idx++) {
18      kcg_copy_accumulatorInputImagAndReal(&acc, &tmp1);
19      /* _L60=(reverseBits#1)/ */
20      reverseBits(/* _L60= */(kcg_uint32) idx, &acc, _L55, &tmp1);
21    }
22    tmp = /* _L106=(setRealVector#1)/ */
23    setRealVector(&tmp1.outputReversedReal);
24    /* _L90= */
25    if (tmp) {
26        .
27        .
```

Source: Image provided by author.

2.3.1 Use Cases

As the benefits of MBD tools make them more and more attractive to companies, several use cases have been broadly reported. Automotive companies [1] make use of Scade tool to save development time, support long-standing software development, and guarantee the necessary safety. Applications in the field of autonomous system developments [2] have also reported the use of MBD tools since end-to-end traceability can be achieved, whilst being compliant with regulations.

[1] Companies can utilize MBD tools to achieve high speed and great quality software, automating its development process: https://www.ansys.com/resource-center/article/taking-control-aa-v13-i3

[2] https://www.ansys.com/applications/autonomous-system-development

Figure 2.5 – Context variable type, automatically generated by Scade tool.

```
/* ======================== context type ========================== */
typedef struct {
  /* --------------------------- outputs ---------------------------- */
  whileLoopAccumulator /* _L13/, outAcc/ */ outAcc;
  /* ------------------------ no local probes ----------------------- */
  /* ------------------------ no local memory ----------------------- */
  /* -------------------- no sub nodes' contexts -------------------- */
  /* ----------------- no clocks of observable data ----------------- */
} outC_mainOperator;
```

Source: Image provided by author.

Figure 2.6 – Global variables generated by Simulink that hold information about specific parts of the program.

```
/* Block states (auto storage) for system '<S4>/For Iterator Subsystem' */
typedef struct {
  real_T gold_PreviousInput;           /* '<S21>/gold' */
} rtDW_ForIteratorSubsystem_basic;

/* Block signals for system '<S4>/For Iterator Subsystem' */
typedef struct {
  real_T Add;                          /* '<S21>/Add' */
} rtB_ForIteratorSubsystem_basicM;
```

Source: Image provided by author.

The aerospace industry can also benefit from its applications by utilizing MBD tools [3], once they can generate automatic source code for logic and display applications. Several healthcare studies have also reported the use of MBD tools such as Simulink, where researchers can benefit from the outstanding simulation properties, once they can analyze and monitor real-life sensitive data [4].

2.4 Transient Faults

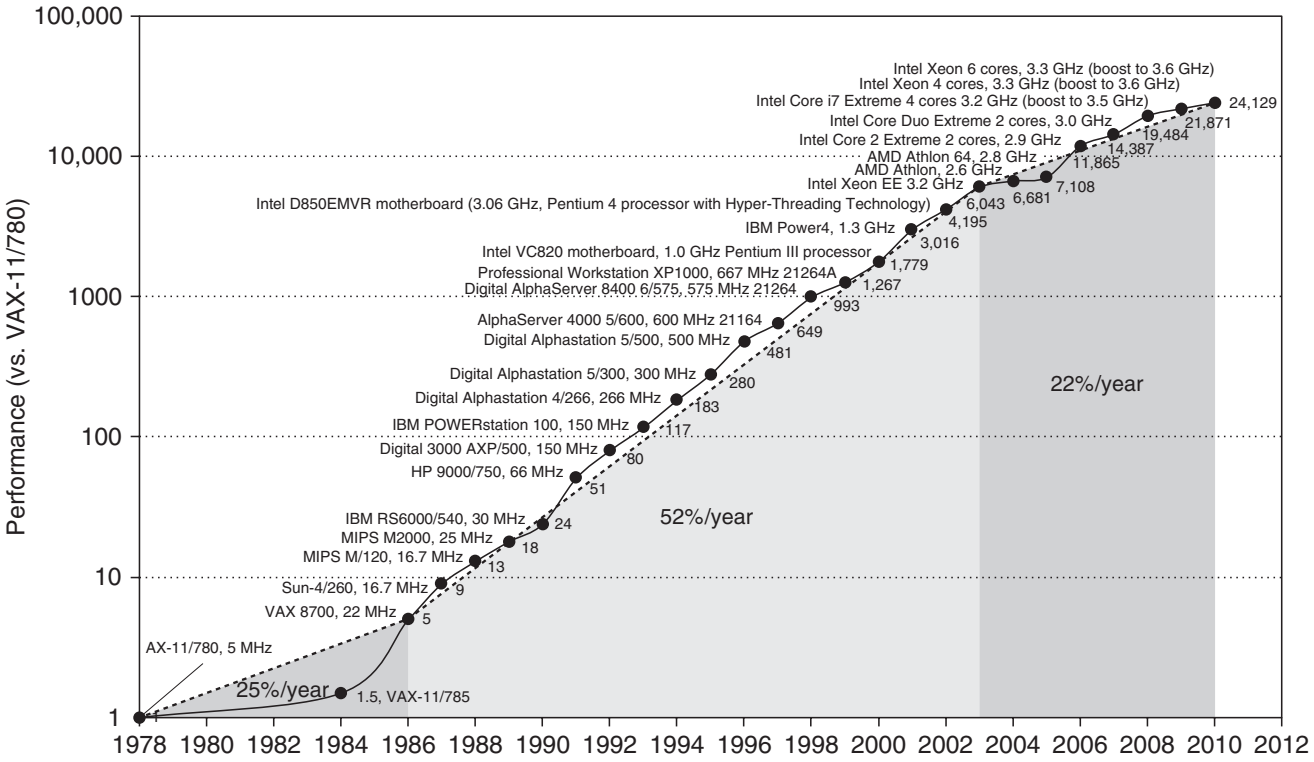
Over the years, consumers' desire for powerful electronics has made the industry continuously increase the performance of computer electronics, as seen in Figure 2.7.

[3] Embraer utilizing Scade display tool to develop cockpit displays. Available at https://www.intelligent-aerospace.com/avionics/article/16540151/embraer-selects-scade-arinc-661-solutions-from-esterel-for-cockpit-display-development

[4] Human Brain Mapping study from University College of London, available at https://nl.mathworks.com/company/newsletters/articles/using-machine-learning-to-predict-epileptic-seizures-from-eeg-data.html

In the initial years, the performance of the processors was highly driven by technology, where the increments in performance were about 25% per year. After the mid-'80s, the great spike in performance gain can be related to architectural concepts, going up to 52% per year. Beyond the so-called golden era, processors have been improving their performance by about 22% per year, where power limits are observable in all processor projects.

Figure 2.7 – Growth in processor performance since the late 1970s.

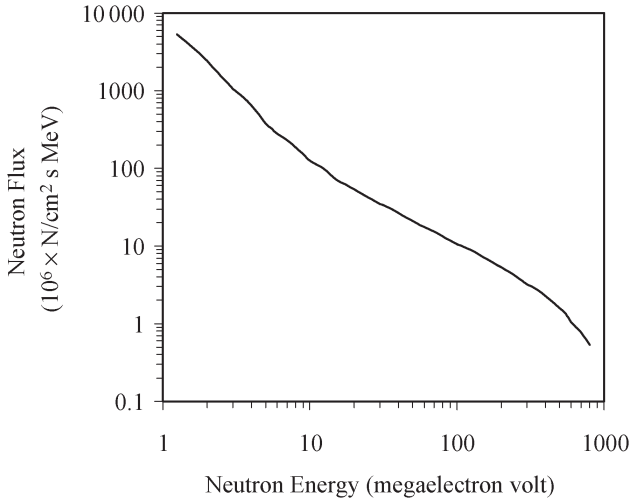


Source: (HENNESSY; PATTERSON, 2011)

Along with the desire for performance, processors have been reducing their power consumption, increasing the transistor density inside them (MOORE, 1965) and also decreasing the transistor's feature size. The feature size is defined as the minimum length of the gate, between the transistors source and drain, as depicted in Figure 2.11. The decrease in the feature size of transistors allows more transistors to be placed in chips, granting more performance. However, their shrinking can be dangerous, as the necessary energy to make a transistor switch its state and generate a current is reduced.

As the size of computer electronics is reduced more and more to match these ambitious performances and power consumption standards, their sensitivity to radiation increases, as the reversed-biased junction is a sensitive part of circuits and can be easily disturbed by an ionizing radiation event (BAUMANN, 2005). High energetic ions can can pass through the transistor, forming a high concentration of carriers. Then, when the

Figure 2.8 – Cosmic ray differential neutron flux as a function of neutron energy at sea level. Adapted from (ZIEGLER; LANFORD, 1980).

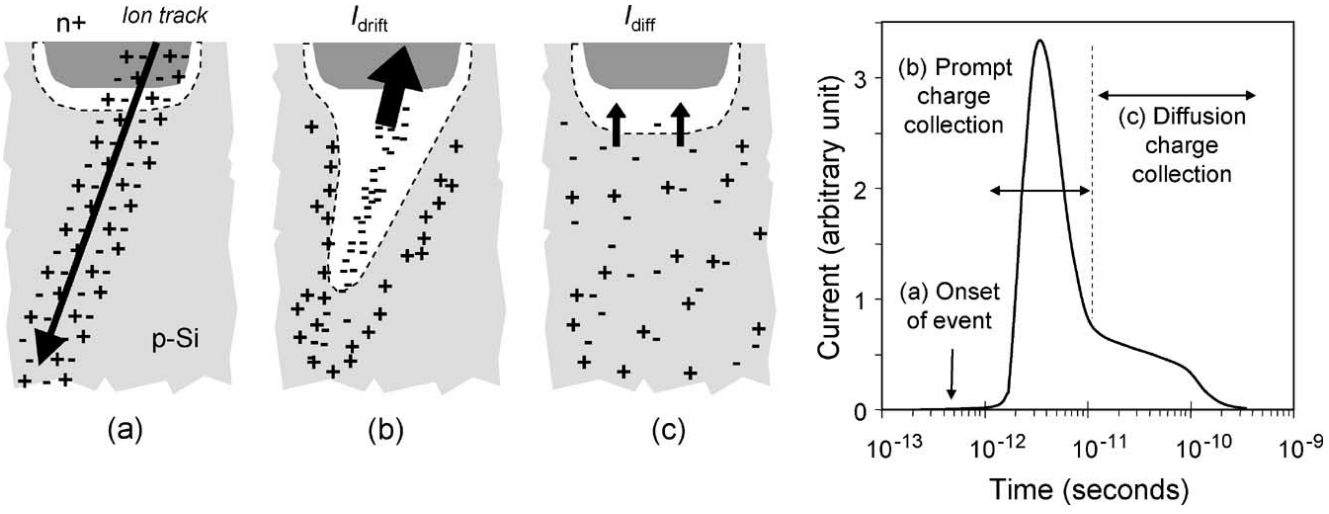


Source: (BAUMANN, 2005)

ionization travels back to the depletion region, those formed carriers are gathered by the resulting electric field, creating a current spike at the observed node.

When these radiation events occur, a current spike can happen, as depicted in Figure 2.9. So, as transistors work like a switch, this resulting current spike can change the transistors' internal state, thus generating unwanted behaviors.

Figure 2.9 – Charge generation and collection phases in a reverse-biased junction and the resultant current pulse caused by the passage of a high-energy ion.

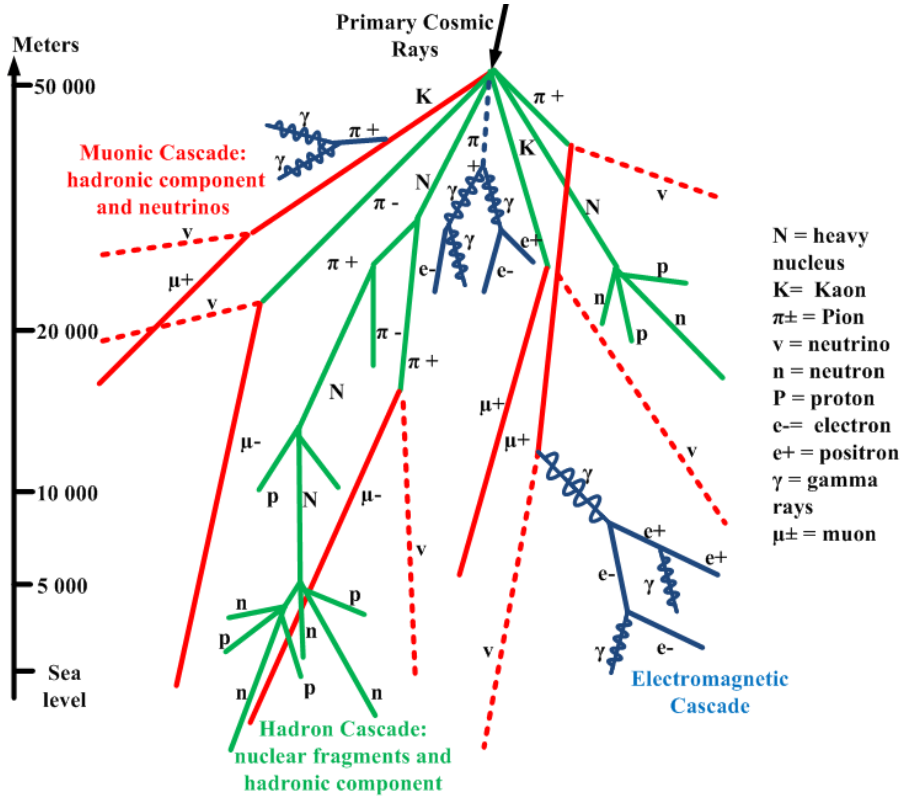


Source: (BAUMANN, 2005)

As electronic devices are exposed to the atmospheric environment at sea level and even higher levels (in the case of airborne applications), they can be affected by radiation effects depending on the altitude. Figure 2.8 illustrates the amount of neutrons over the

energy range that are incident at the sea level. One of the greatest sources of radiation on the earth is high-energy cosmic rays. These high-energy cosmic rays have a galactic origin, and react with the earth's atmosphere, creating lots of secondary particles, as depicted in Figure 2.10. Consequently, as devices are often exposed to highly energetic particles, caused by these interactions, particles can deposit or induce enough charge, interacting with the transistor's silicon active area to generate a current spike (BAUMANN, 2005), (GOLDHAGEN, 2003).

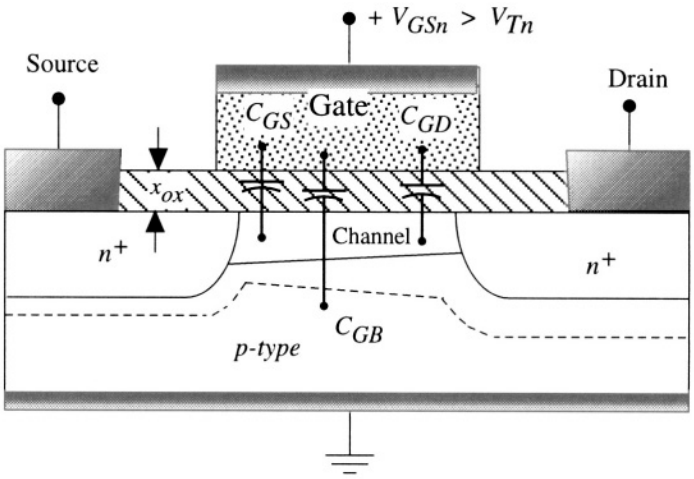
Figure 2.10 – Showers of cosmic ray reactions with particles of the atmosphere.



Source: (GOMEZ TORO, 2014).

If the spike is latched, a Single Event Transients (SETs) occurs. The SET can corrupt the output of an operation or it can switch the logical value of bits inside memory, inducing a bit(s) flip. If propagated to the software visible state or the application output, these faults can either manifest as a Silent Data Corruption (SDC), i.e. the application finishes but its result is wrong, or a Detected Unrecoverable Error (DUE), i.e. a system crash or reboot. Otherwise, the fault is masked, without any observable effect in the code execution.

Figure 2.11 – Gate-channel capacitances in a MOSFET.



Source: (UYEMURA, 1999)

2.5 Related Work

Recent works have shown that compilers can affect the vulnerability of codes (J. GAVA V. BANDEIRA; OST, 2019), as they have their unique characteristics, and can impact code reliability with optimization flags as well. Although the authors focus on compiler effect in multi-core processors, the observed compiler effects on single core processors are still relevant, even for our work, utilizing automatic code generators.

In addition, as this work focus on codes that will be executed in embedded processors, previous works (F. M. LINS L. A. TAMBARA; RECH, 2017) have shown that modern embedded processors also have impact on different compilers and optimization flags as well, analyzing the AVF and different execution characteristics, such as time, clock cycles and memory usage. Besides that, authors have also investigated (OLIVEIRA; PILLA *et al.*, 2017a) code reliability impact on high performance processors, such as the Intel Xeon Phi, showing that even high performance processors, dedicated for parallel and high-end tasks have an intrinsic vulnerability for different codes running on them.

Authors in (WALDE; LUCKNER, 2016) propose a method of translating Simulink/Stateflow models into Scade utilizing the Ansys Scade Suite Simulink Importer, however, the effect of transient faults is not taken into count on the resulted translation.

Authors in (KRIZAN *et al.*, 2014) and (NETLAND; SKAVHAUG, 2013) have shown, indeed, that Simulink and Matlab can be utilized for developing safety critical applications and embedded platforms, although the authors don't evaluate the code generation impact in the code reliability. It is expected that the main modification to the code

imposed by Simulink and Scade, can have a significant impact on the resulting code soft error rate.

3 PROPOSAL

This chapter will detail the motivations for this work, the details of the codes selected for the evaluation, their different implementations, and will describe the selected software fault injector.

3.1 Motivation

MBD tools can offer good alternatives to reduce the bureaucracy and help throughout several software development phases, as they can reduce the extensive documentation demanded by guidelines (such as the DO-178). These tools can also aid in the testing, prototyping, and especially generating automatic embedded code. Even though MBD tools can offer a wide aspect of benefits regarding the documentation process and code generation, their generation of code can significantly modify the overall code in the project. The goal of this work is to investigate if these modifications in the generated code are beneficial or not when considering transient faults.

3.2 Algorithms and different versions

To analyze the impact of transient faults in the generated code, we consider four different algorithms implemented in three different versions (manual, Scade, and Simulink).

3.2.1 Algorithms

Four different algorithms, with different computing characteristics, are selected for the fault analysis. During fault-injection we repeat the execution of each code several times. Each algorithm has the same number of iterations across its different versions (manual, Scade, and Simulink). This is necessary to increase the execution time, allowing a more accurate measure of the impact of the injected faults by the fault injector tool. As real-world codes are not publicly available (because of industry restrictions), we choose algorithms that have computational characteristics that resemble real algorithms that would be present in a safety-critical application. The selected algorithms are:

- **Inverse Fast Fourier Transform (IFFT):** The Fast Fourier transform (FFT) algorithm computes the discrete Fourier transform of a given sequence, or its inverse. Particularly, for this work, we chose to perform the *Inverse* Fast Fourier transform (IFFT). This algorithm is chosen due to its application in Filtering Algorithms, that are highly present in embedded systems. In particular, the Simulink IFFT version is built with modeling blocks and also MATLAB pieces of code that are embedded in Simulink, in order to facilitate the implementation of the algorithm. The input for the IFFT consists in two sequences of 512 single precision floating-point values of 32 bits, statically allocated.
- **Matrix Multiplication (MxM):** The Matrix Multiplication algorithm is commonly used in benchmarks due to its memory bound aspect, which can suffer from several cache misses when large matrices are used. This algorithm is chosen for this paper due to its common utilization in embedded systems. The matrix multiplication algorithm is usually utilized to represent large amounts of data, such as tables with pre-determined values and coordinates. Four 32x32 statically allocated matrices of double-precision floating-point elements (64bits) are utilized as inputs for this algorithm. The matrices are multiplied, two by two, and then their results are also multiplied, forming the final matrix.
- **Basic Math:** The Basic Math algorithm computes several angle conversions (from degrees to radians and vice-versa), cubic equations, and integer square roots. This algorithm is chosen due to its presence in embedded systems, where angle conversions are extremely frequent, so as equations. The inputs for this algorithm consist of basic angle conversions and predefined cubic polynomial equations to be solved.
- **Quicksort:** The Quicksort algorithm is commonly utilized in overall software due to its efficient computation when ordering vectors. The inputs for this algorithm consist of the ordering of 5000 statically allocated 32-bit integers, disposed of in descending order. Due to restrictions of the utilization of recursion in embedded software, the Quicksort implementation is made with an iterative approach.

3.2.2 Versions

Each presented code is implemented in three different versions, within the same inputs and and generated outputs:

- **Manual written:** The manual version is the most common way of writing source code. A Software Developer manually writes source code, with the implementations that she/he may find appropriate. It is important to notice that, with this method of writing the code, one can introduce unwanted errors, due to the manual writing. For this paper, all algorithms manual implementations are taken from the universally adopted benchmarks suites Polybench (THE..., 2021) and MiBench (MIBENCH..., 2021).
- **Simulink Generated:** The second version consists of the implementation and design of each algorithm, utilizing the Simulink modeling tool. After the implementation is ready, the resulting model automatically generates the code. Although Simulink is not a certified tool, it is broadly used to design and prototype software and hardware projects, and it can even be utilized to generate automatic code for critical applications, yet not replacing some of the verification needed (as Scade does). Simulink code was generated utilizing Simulink Coder version 8.3 (R2012b) version.
- **Scade Generated:** The last version consists in the implementation of each algorithm with automatic generated source code from the certified tool Scade, in the 2021 R1 student version, with the KCG Code Generator, developed by ANSYS (ANSYS..., 2021 (accessed January 7, 2021)). First, the representative models of each algorithm are designed, and then, the code will be generated. Scade Suite is mainly utilized to design critical software, such as flight control, engine control, and automatic pilot systems. Scade drastically reduces the certification costs of a project, by simplifying the design and automating the code generation and verification. The Scade code generator (KCG) is qualified to be utilized with the DO-178B level A, since the tool was also certified according to the DAL A objectives.

3.3 CAROL-FI Fault injector

In order to assess the program's reliability, this work adopts a fault injector named CAROL-FI(OLIVEIRA; FRATTIN *et al.*, 2017), publicly available in (CAROL-FI..., 2018 (accessed January 7, 2021)). The injector is built on top of GNU GDB and performs bit-flips on the memory of an executing program. The injector loads the program's symbols in debug version and utilizes the symbols to identify each allocated memory segment

of the program.

As a result of the compiling of the programs in debug mode, we are not able to introduce code optimizations, however, this is not a problem since safety-critical applications usually don't utilize compiler optimizations on the embedded code.

CAROL-FI's process of injecting a fault in any given program is simple and consists of two scripts. The first script is utilized for monitoring the program execution and identifying whether the program surpassed a user-defined time limit, for example. At any given time, the program can be interrupted, and the second script randomly chooses variables to perform bit-flips, according to four pre-defined modes. The models are:

- **Single**: flips only a random bit from the selected variable
- **Double**: flips two random bits from the selected variable
- **Random**: flips every bit by a random value from the selected variable
- **Zero**: all bits of the variable are set to zero

The utilized fault injector does not differentiate between logic and memory errors, as we inject faults in allocated memory, considering transient errors that could propagate from transistors to a memory value. These transient faults can include errors that originated from the lowest architectural level (caches, registers, flip-flops) to the highest (memory variables). So, for the purpose of this work, we utilize the random mode for all injections. The Random mode is the more suitable to simulate the effect of faults occurring in the computation that updates a variable (OLIVEIRA; PILLA *et al.*, 2017b).

We inject more than 3,500 transient fault in each selected code in its corresponding version. That is, more than 40,000 faults were injected. After the injection of the fault is completed, we identify the effect of the fault in: DUE, SDC, and Masked. The Masked category identifies that the injected fault had no impact on the code. Then, we compare the Program Vulnerability Factor (PVF) (SRIDHARAN; KAELI, 2009) of the different codes and implementations. The PVF is the probability for a fault affecting a software visible state to propagate to the output.

4 EXPERIMENTS

This section has the purpose of analyzing the different results for each algorithm, at each compiled version, using the proposed methodology described in Section 3.

4.0.1 Silent Data Corruption

Figures 4.1, 4.2, 4.3, and 4.4 show the PVF of SDCs, DUEs and Masked for all the algorithms. In general, the manual version of all algorithms shows the highest number of SDCs, up to 56.37% of the injected faults in the Basic Math algorithm. The higher PVF for SDCs in the manual version of algorithms is given by the structure of the code itself. Manual codes tend to not re-utilize variables, whereas Simulink and Scade code generators utilize a greater amount of variable re-use. The variable re-use impacts the code reliability, since one fault will be masked if the affected (obsolete) variable is overwritten.

The smaller SDCs PVF in the Simulink version compared to the other implementations for each algorithm is justified by the higher amount of variables that are generated in the code (see Table 4.2). Most of those variables are generated to be utilized in calculations, such as additions and divisions. As these variables are re-utilized in different calculations and, then, are overridden, faults occurring after a read operation and before the next calculation are masked. Additionally, the code generated by the Simulink tool tends to utilize variables that represent signals and states for accumulators inside loops, that are also overridden as the program executes. Our results show this effect when observing the Matrix Multiplication algorithm, that heavily utilizes loops, but have a lower number of generated variables. In addition, the Quicksort algorithm presents a greater number of masked and also a reduced number of SDCs in the Scade version. As the quicksort algorithm consists of swapping elements, finding positions, and iterating through the vector of elements, its generated code remains similar throughout the three versions - as swapping of the elements and index variables are commonly structured the same. However, in the Scade version, conditional statements and variables utilized inside them happen to be re-utilized - resulting in a higher number of masked results.

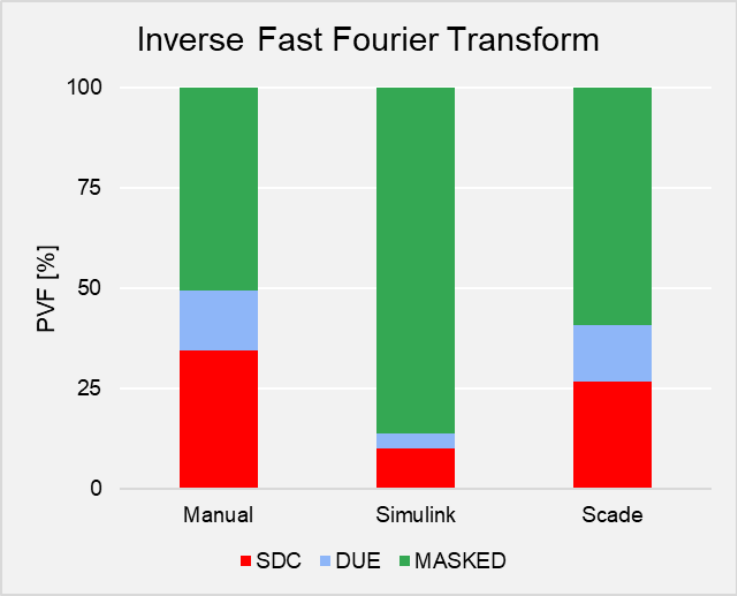


Figure 4.1 – PVF for the Inverse Fast Fourier Transform algorithm.

4.0.2 Detected Unrecoverable Errors

When analyzing the PVF of DUEs, the Scade version of all algorithms presents the highest amount of crashes and hangs, up to 36.04% higher when compared to the Basic Math algorithm, as seen in Figure 4.4. The high amount of DUEs in the Scade version of the algorithm is justified by the higher amount of loop index variables and the overall *program context* that are generated by the tool with respect to the other implementations, as seen in Figure 2.3. Faults in these variables and context stuck the program, leading to DUEs.

The program context is a type-specific global variable automatically generated that holds information of all steps of the entire algorithm, such as accumulators, indexes and even some auxiliary variables that are utilized in the program's calculation. The context is also passed-by as a pointer throughout several function calls. Faults that affect the context memory address can cause severe harm to the program's execution, since they can corrupt a memory addresses that hold a lot of information. We also observe that some variables in the code have a higher PVF than others. For example, the context variable present in the Inverse Fast Fourier Transform generated code has a PVF of around 75%, while the variable _L64_blockEndIterator_1_then_IfBlock1 that stores the result of comparisons has the PVF of around 25%, as depicted in Table 4.1.

Finally, it is interesting to notice that, while exacerbating the PVF for DUEs, the Scade version presents 52.46% less SDCs, on the average, when comparing to the manual version. The use of Scade in safety-critical applications, then, should be carefully

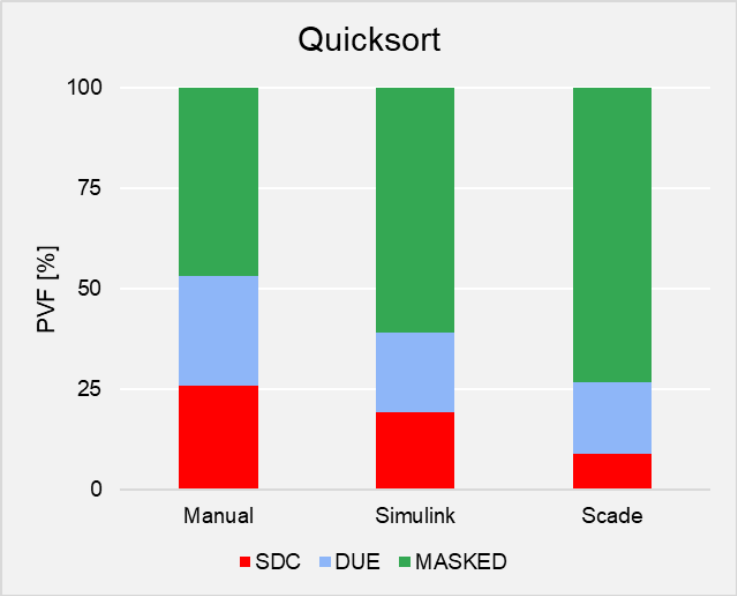


Figure 4.2 – PVF for the Quicksort algorithm.

Table 4.1 – Variable name and PVF for variables that caused DUEs in IFFT Algorithm.

Variable Name	**PVF (%)**
numSamples	96.34
outC	74.60
idx	38.02
_L64_blockEndIterator_1_then_IfBlock1	25.0
accumulator	30.76
accSecondBlockSizeIterator	8.33

engineered. The higher amount of DUE might not jeopardize the system reliability, once watchdog or other mechanisms are employed to deal with *detected* events. If such mechanisms are adopted, Scade might be a good solution, as it reduces SDCs, that, being *undetectable*, are much more harmful.

4.0.3 Execution time and Number of Variables

When looking at Table 4.2, we can notice that the execution time and number of variables of each version are very different, since each code generator is built in a different manner. The number of variables is composed of all local and global statically allocated variables that are present in the code. In particular, the execution time increases from the manual to the Scade version.

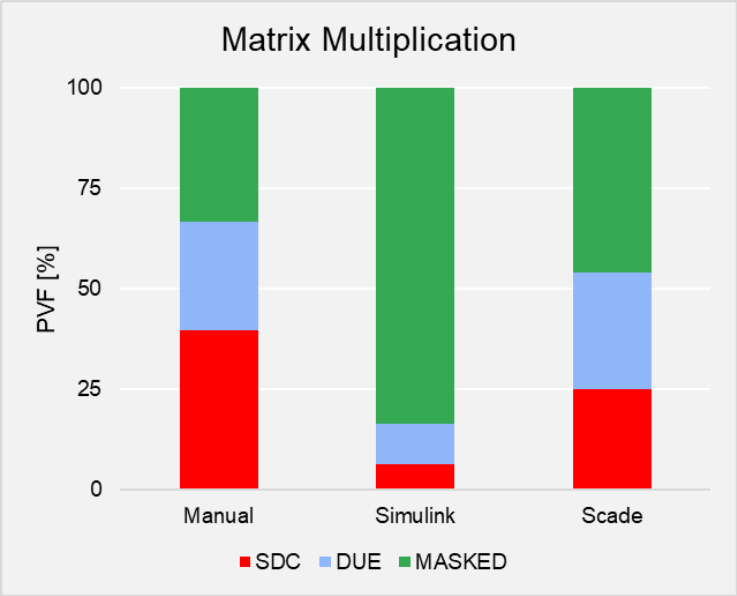


Figure 4.3 – PVF for the Matrix multiplication algorithm.

Observing the execution time for the Inverse FFT algorithm, we see that the Scade version takes longer since the generated code is much denser. Even though the Scade version takes the longest, results show a decrease in the probabilities of SDCs to occur, when comparing to the manual version. It is worth noting that our data reports the PVF, i.e., the probability for a fault to propagate to the output and generate an SDC or DUE. No information is given, as in any fault injection experiments, on the probability for the fault to occur. Such a probability can indeed be higher for codes with longer execution time (there is more time for particles to hit the device). As a result, the execution time increase might have a not negligible impact on the error rate of Scade and Simulink. In the future we will measure such an impact with beam experiments.

In general, applications that require lots of logic and calculations, such as IFFT and BasicMath, for example, are more likely to have an increased amount of variables utilized in the generated code. These applications also have an increase in the execution time, since the overall generated code and the required model-based logic are much more complex then the manual version.

It is worth noting that, for applications that require a simpler logic, as is the case of Matrix Multiplication algorithm, the use of pre-defined operators that are available in the design tools is possible and results in a quicker and cleaner generated code.

In particular, the Scade version of the Matrix Multiplication algorithm is faster than the manual version. Analyzing the generated code, we see that the specialized operators inside the tool are highly efficient. These operators make use of pointers so they can avoid variables copies throughout function calls, yet these pointers are dangerous when

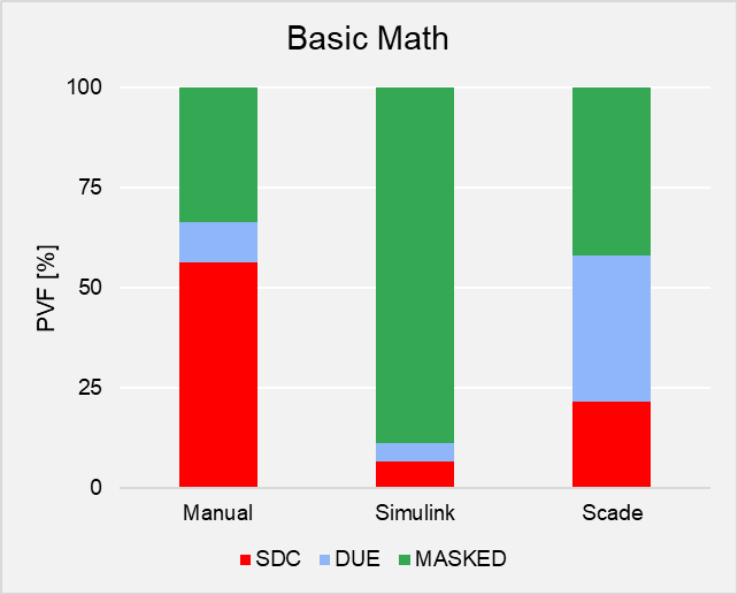


Figure 4.4 – PVF for the Basic Math algorithm.

bit flips change their memory values.

Table 4.2 – Execution time, SDC PVF, DUE PVF and number of variables for different configurations.

Algorithm	Version	Variable #	Exe. Time (s)	PVF SDC (%)	PVF DUE (%)
Inverse FFT	Manual	36	4.92	34.6	15.02
	Simulink	61	6.59	10.08	3.62
	Scade	43	24.88	26.71	14.15
Matrix Multiplication	Manual	26	5.736	39.74	26.94
	Simulink	15	5.565	6.48	9.94
	Scade	22	4.576	25.00	29.00
Basic Math	Manual	24	5.203	56.37	10.11
	Simulink	100	5.822	6.71	4.54
	Scade	28	8.68	21.60	36.40
Quicksort	Manual	13	2.62	19.71	27.37
	Simulink	21	8.07	19.46	19.71
	Scade	31	3.98	8.86	17.8

5 CONCLUSION

This work investigated and discussed the impact on the reliability of codes generated in Scade and Simulink with respect to the manual, classical, implementation. We considered four different algorithms, with unique characteristics and complexity.

We observe that the generated code structure has a great impact on both the SDC and DUE rates, in all four algorithms. We also noticed some side effects regarding the utilization of MBD tools. The generated code has more variables and can also have more complex logic, whereas the manual code is cleaner - yet has a higher SDC rate.

Scade presents the greatest amount of DUEs among the three versions. This high amount of DUEs are related to Scade's code generator utilizing a global variable that holds most of the variables utilized throughout all the code - the program's context.

Even though a high amount of DUEs is observed in the Scade generated code, a significant decrease in the amount of SDCs is noticed. Since the tool is often utilized in safety-critical applications and our results show that the effect of transient faults in the generated code is minimized, the Scade tool is a promising tool to be utilized, since the project can take advantage of all certification aspects of the tool, and generate a more reliable code.

When looking at the relation of the execution time, the number of variables, and the associated reliability, we can see that the associated complexity of the algorithm to be utilized in each tool must be taken into account since utilizing an automatic code generator tool can bring extra costs, such as a slower execution time. Also, these code generator tools offer the programmer highly optimized built-in operators, yet they can make it difficult to elaborate complex logic, such as more elaborated algorithms.

In the future, we plan to see if the observed impact on the reliability of the use of Model-based Design frameworks holds also for different architectures and we also would like to expose these implementations to a neutron beam to measure the actual FIT rate dependence.

REFERENCES

SC-205 RTCA. Software Considerations in Airborne Systems and Equipment Certification. *In*.

ANICULAESEI, A.; VORWALD, A.; RAUSCH, A. Using the SCADE Toolchain to Generate Requirements-Based Test Cases for an Adaptive Cruise Control System. *In*. 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C). [*S. l.: s. n.*], 2019. p. 503–513. DOI: 10.1109/MODELS-C.2019.00079.

ANSYS. [*S. l.*], 2021 (accessed January 7, 2021). Disponível em: https://www.ansys.com/.

BAUMANN, R. C. Radiation-induced soft errors in advanced semiconductor technologies. **IEEE Transactions on Device and Materials Reliability**, v. 5, n. 3, p. 305–316, 2005. DOI: 10.1109/TDMR.2005.853449.

BOUALI, Amar; DION, Bernard. Formal Verification for Model-Based Development, abr. 2005. DOI: 10.4271/2005-01-0781.

CAROL-FI Fault Injector. [*S. l.*], 2018 (accessed January 7, 2021). Disponível em: https://github.com/UFRGS-CAROL/carol-fi.

F. M. LINS L. A. TAMBARA, F. L. Kastensmidt; RECH, P. Compiler Optimization Effects on Embedded Microprocessor Reliability. **IEEE Transactions on Nuclear Science**, IEE, 2017.

GOLDHAGEN, Paul. Cosmic-Ray Neutrons on the Ground and in the Atmosphere. **MRS Bulletin**, Cambridge University Press, v. 28, n. 2, p. 131–135, 2003. DOI: 10.1557/mrs2003.41.

GOMEZ TORO, Daniel. **Temporal Filtering with Soft Error Detection and Correction Technique for Radiation Hardening Based on a C-element and BICS**. Dez. 2014. Tese (Doutorado).

HENNESSY, John L.; PATTERSON, David A. **Computer Architecture, Fifth Edition: A Quantitative Approach**. 5th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. ISBN 012383872X.

J. GAVA V. BANDEIRA, R. Reis; OST, L. Evaluation of Compilers Effects on OpenMP Soft Error Resiliency. **2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)**, IEE, 2019.

KRIZAN, J. *et al.* Automatic code generation from Matlab/Simulink for critical applications. *In*. 2014 IEEE 27th Canadian Conference on Electrical and Computer Engineering (CCECE). [*S. l.: s. n.*], 2014. p. 1–6. DOI: 10.1109/CCECE.2014.6901058.

MIBENCH. [*S. l.*], 2021. Disponível em: http://vhosts.eecs.umich.edu/mibench/.

MOORE, Gordon E. Cramming more components onto integrated circuits. **Electronics**, v. 38, n. 8, abr. 1965.

NETLAND, Ø.; SKAVHAUG, A. Software Module Real-Time Target: Improving Development of Embedded Control System by Including Simulink Generated Code Into Existing Code. *In*. 2013 39th Euromicro Conference on Software Engineering and Advanced Applications. [*S. l.: s. n.*], 2013. p. 232–235. DOI: 10.1109/SEAA.2013.51.

OLIVEIRA, Daniel; FRATTIN, Vinicius *et al.* CAROL-FI: An Efficient Fault-Injection Tool for Vulnerability Evaluation of Modern HPC Parallel Accelerators. *In*. PROCEEDINGS of the Computing Frontiers Conference. Siena, Italy: Association for Computing Machinery, 2017. (CF'17), p. 295–298. ISBN 9781450344876. DOI: 10.1145/3075564.3075598. Disponível em: https://doi.org/10.1145/3075564.3075598.

OLIVEIRA, Daniel; PILLA, Laércio *et al.* Experimental and Analytical Study of Xeon Phi Reliability. *In*. PROCEEDINGS of the International Conference for High Performance Computing, Networking, Storage and Analysis. Denver, Colorado: Association for Computing Machinery, 2017. (SC '17). ISBN 9781450351140. DOI: 10.1145/3126908.3126960. Disponível em: https://doi.org/10.1145/3126908.3126960.

OLIVEIRA, Daniel; PILLA, Laércio *et al.* Experimental and Analytical Study of Xeon Phi Reliability. *In*. PROCEEDINGS of the International Conference for High Performance Computing, Networking, Storage and Analysis. Denver, Colorado: Association for Computing Machinery, 2017. (SC '17). ISBN 9781450351140. DOI: 10.1145/3126908.3126960. Disponível em: https://doi.org/10.1145/3126908.3126960.

SCADE Suite. [*S. l.*], 2021 (accessed January 7, 2021). Disponível em: https://www.ansys.com/products/embedded-software/ansys-scade-suite.

SIMULINK. [*S. l.*], 2021 (accessed January 7, 2021). Disponível em: https://www.mathworks.com/products/simulink.html.

SRIDHARAN, V.; KAELI, D. R. Eliminating microarchitectural dependency from Architectural Vulnerability. *In*. 2009 IEEE 15th International Symposium on High Performance Computer Architecture. [*S. l.: s. n.*], 2009. p. 117–128. DOI: 10.1109/HPCA.2009.4798243.

THE Polyhedral Benchmark suite. [*S. l.*], 2021. Disponível em: http://web.cs.ucla.edu/~pouchet/software/polybench/.

UYEMURA, John P. **CMOS Logic Circuit Design**. USA: Kluwer Academic Publishers, 1999. ISBN 0792384520.

WALDE, G.; LUCKNER, R. Bridging the tool gap for model-based design from flight control function design in Simulink to software design in SCADE. *In*. 2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC). [*S. l.: s. n.*], 2016. p. 1–10. DOI: 10.1109/DASC.2016.7778044.

ZIEGLER, J.; LANFORD, W. The effect of sea level cosmic rays on electronic devices. *In*. 1980 IEEE International Solid-State Circuits Conference. Digest of Technical Papers. [*S. l.: s. n.*], 1980. v. XXIII, p. 70–71. DOI: 10.1109/ISSCC.1980.1156060.