FEDERAL UNIVERSITY OF SANTA CATARINA
TECHNOLOGICAL CENTER
DEPARTMENT OF AUTOMATION AND SYSTEMS

Mateus Giovani Ewert Bonet

A Comparison of Model Checking Techniques for Cause and Effect Matrix Based Controller Logic of Safety Instrumented Systems

Florianópolis

2019

Mateus Giovani Ewert Bonet

A Comparison of Model Checking Techniques for Cause and Effect Matrix Based Controller Logic of Safety Instrumented Systems

Report submitted to the Federal University of Santa Catarina as a requirement for approval in **DAS 5511: Projeto de Fim de Curso** of the Automation and Control Engineering Undergraduate Program.
Supervisor: Max Hering de Queiroz

Florianópolis
2019

Mateus Giovani Ewert Bonet

A Comparison of Model Checking Techniques for Cause and Effect Matrix Based Controller Logic of Safety Instrumented Systems

This final project was judged adequate for the attainment of the Title of Bachelor in Control and Automation Engineering, and was approved in its final form by the Automation and Control Engineering Undergraduate Program.

Florianópolis, December 11th of 2019.

Prof. Dr. Eng Hector Bessa Silveira
Course Coordinator
Federal University of Santa Catarina

Evaluation Board:

Max Hering de Queiroz
Supervisor
Federal University of Santa Catarina

Dr. Eng. Silvano Dal Zilio
Company Supervisor
Laboratoire d'Analyse et d'Architecture des Systèmes

Dr. Eng. Rodrigo Tacla Saad
Evaluator
Federal University of Santa Catarina

Agradecimentos

Gostaria de agradecer ao meu orientador e supervisor professor Max Hering de Queiroz, pelo auxílio profissional e pessoal fornecido durante todas as etapas do estágio, e pelo exemplo dado como profissional de ensino.

Aos meus pais, Sérgio e Elfrida, que me apoiaram durante as etapas da Universidade e todas as anteriores a ela.

Ao meu irmão, Eduardo, que serviu de inspiração para entrada no curso e alívio nas horas de estresse com papos descontraídos e vídeos da Patatje. Agradeço o Edu e também a Amanda pelo acolhimento que meu ofereceram na Europa, que foi crucial para que eu aproveitasse o período.

À minha namorada, Jéssica, por me ajudar a relevar os momentos difíceis e aproveitar os momentos bons, pelas noites de Skype separadas por um oceano inteiro, mas que fizeram toda a diferença. Obrigado por me ajudar a acreditar nas minhas capacidades quando eu me sentia abatido.

À meus grandes amigos, Henrique e Julia, por me ajudar a me descontrair e a aproveitar a vida mais levemente, e pelos conselhos nas horas difíceis.

À Petrobras, pela oportunidade de trabalhar neste projeto.

Ao Departamento de Automação e Sistemas da UFSC e aos professores que fazem parte dele, pelo incentivo e pelo exemplo a seguir.

E por fim, mas não menos importante, um obrigado a todos os meus amigos e familiares que fizeram parte dessa jornada.

Acknowledgements

I would like to thank my supervisor, Silvano Dal Zilio, who offered me this great opportunity and helped me overcome any eventual setbacks that could've prevented me from working at LAAS, and offered me a great example of a dedicated supervisor.

Also my great colleagues, Thomas, Nesredin, Xin and Karla for the great moments I lived in Toulouse, whether it was watching the sunset over the Garonne, rock-climbing, foosball or simply having a coffee break to catch up, it was great being with you.

Abstract

Safety Instrumented Systems (SIS) are safety critical mechanisms that seek to reduce the probability of dangerous events within industrial plants. The behavior of said systems consists of measuring a process' state via a sensor, making a decision based on a pre-established control logic and activating actuators such that the process is guided back to a safe state in case an undesirable situation is encountered. During the controller's development phases, errors might be accidentally introduced, causing the behavior of the implemented logic to be incoherent with the one specified. Thus, methods to verify the controller software logic must be applied within the development methodology. Model Checking is a method that uses mathematical modeling to computationally search for errors within a system. In this project, methods for model generation from Ladder diagrams and property extraction from a Cause & Effect Matrix specification are outlined. Then, two Symbolic Model Checking methods, Binary Decision Diagrams (BDD) and Satisfiability Modulo Theory (SMT), have their accuracy and efficiency evaluated in the context of SIS logic with temporal requirements. It was found that while SMT based methods have difficulty with systems in which the timed logic is complex, BDD based methods were effective in determining the existence or absence of errors.

Keywords: Safety Instrumented Systems, Formal Verification, Cause & Effect Matrix, Bounded Model Checking, Interlock Logic, Temporal Logic.

Resumo Expandido

Resumo

Sistemas Instrumentados de Segurança (SIS) são mecanismos críticos que visam diminuir a probabilidade de eventos perigosos em plantas industriais. O princípio de funcionamento de tais sistemas consiste em medir o estado de um processo através de sensores, tomar uma decisão com base em uma lógica de controle pré-estabelecida e acionar atuadores de forma a guiar o processo a um estado seguro caso necessário. Durante as etapas de desenvolvimento do controlador, erros podem ser acidentalmente introduzidos, de modo que o comportamento do sistema seja incoerente com o especificado. Portanto, meios de verificação da lógica implementada devem ser aplicados na metodologia de desenvolvimento. *Model Checking* é um método que utiliza modelagem matemática para procurar computacionalmente erros dentro de um sistema. Neste projeto, métodos para geração de modelos a partir de diagramas *Ladder* e extração de propriedades de matrizes Causa e Efeito são delineadas. Em seguida, dois métodos simbólicos de *Model Checking*, Diagramas de Decisão Binária (BDD) e Teoria de Módulo de Satisfatibilidade (SMT), tem sua eficiência e acurácia avaliadas dentro do contexto de lógica de SIS com requisitos temporais. Foi encontrado que enquanto métodos SMT tem dificuldade com sistemas nos quais a lógica temporal é complexa, métodos BDD foram efetivos em determinar a presença ou ausência de erros.

Palavras-chave: Sistemas Instrumentados de Segurança. Verificação Formal. Matriz Causa e Efeito. Bounded Model Checking. Lógica de Intertravamento, Lógica Temporal.

Introdução

Sistemas de Segurança Críticos são sistemas que, em caso de mal-funcionamento, podem causar danos a pessoas, equipamentos e ao meio ambiente (por exemplo em plataformas de extração de petróleo, usinas nucleares, etc.). Por esse motivo, é comum a existência de diversos níveis de atuação (Figura 2) dedicados à prevenção de riscos e à contenção de danos (American Institute of Chemical Engineers, 2001).

Sistemas Instrumentados de Segurança (SIS) são mecanismos de *hardware* e *software* que operam unicamente dentro do escopo da prevenção de riscos, de modo a meramente reduzir a probabilidade de ocorrência de estados indesejáveis. Por serem o último nível de prevenção de riscos, o projeto de SIS deve contar com uma metodologia que empregue uma rotina de testes durante todas a fases de desenvolvimento e instalação. Para isso, normas técnicas como a IEC61508 (IEC61508, 2010) e IEC51511 (IEC61511, 2018) foram criadas.

Quanto maiores os graus de severidade e probabilidade de ocorrência de um risco, maior o Nível de Integridade de Segurança (SIL) que esse sistema deve possuir, dentro da faixa de 1 a 4. De acordo com a IEC61511, é recomendado que sistemas com SIL maior ou igual a 2 passem por processos de verificação formal para identificação de erros de implementação do código do controlador CLP. Existem dois tipos de erro dentro de um SIS:

- Ativação Extemporânea: Situação em que uma saída é acionada quando as entradas não formam uma combinação que deveria ativá-la, conforme a especificação. Também chamada de falha segura.
- Falha sob Demanda: Situação em que uma saída não é acionada quando as entradas formam uma combinação que deveria ativá-la, conforme a especificação. Também chamada de falha perigosa.

Um método formal que pode ser utilizado para verificar se o sistema está livre dessas falhas é o Model Checking, técnica na qual um sistema é representado por um modelo formal, geralmente uma Máquina de Estados Finita (FSM), e propriedades que se deseja conhecer sobre esse sistema são convertidas em lógica temporal (BIERE et al., 2003). Por fim, verifica-se se o modelo satisfaz todas as propriedades. No entanto, Model Checking apresenta suas próprias desvantagens, e portanto deve ser utilizado como técnica complementar a outros modos de verificação.

O objetivo deste trabalho é analisar diferentes técnicas de Model Checking dentro do contexto de SIS expressos via Matrizes Causa e Efeito (CEM), sobre a perspectiva de acurácia e eficiência de cada método em diferentes cenários.

Nesse trabalho as especificações são expressas majoritariamente na forma de Matrizes Causa e Efeito. Esse meio de representação da lógica de intertravamento consiste na disposição visual das relações entrada(linhas)-saída(colunas) de um sistema de controle na forma de uma matriz. Isso facilita o reconhecimento rápido da complexidade da lógica de controle, bem com quais entradas estão relacionadas a quais saídas. Nesse caso, a lógica de ativação da saída por uma entrada é escrita na intersecção da linha e coluna correspondentes a tais sinais. Outra vantagem desse formalismo é a fácil tradução da lógica especificada para código *Ladder*, que por ser feita de maneira direta e intuitiva, reduzindo a probabilidade de erros na etapa de programação.

Existem diversas regras de construção e utilização de Matrizes Causa e Efeito dentro dos diversos ramos da indústria. Nesse trabalho, será utilizada uma semântica mista entre as utilizadas pela Petrobras, pela Associação Americana de Petróleo (API) e a descrita em (LAZARO, 2018).

Devido à complexidade natural dos comportamento de intertravamento de SIS, como alto número de sinais e lógica temporizada, testes funcionais do controlador implementado não garantem um veredito completo e positivo. Nesse contexto métodos de verificação formal, tais como o Model Checking, surgem como ferramentas complementares para verificar a precisão da lógica implementada (IEC61508, 2010).

Na medida que o número de estados de um modelo cresce exponencialmente com o número de sinais de entrada, técnicas de abstração e métodos de verificação além da enumeração completa de estados se tornam necessários. Um método para tal é a representação simbólica de um modelo. Duas técnicas possíveis são Diagramas de Decisão Binária (BDD) e Teoria do Módulo de Satisfatibilidade (SMT). Enquanto o primeiro trabalha com diagramas acíclicos para codificar uma expressão binária que representa os estados do modelo, o segundo utiliza teorias de primeira ordem para aplicar métodos altamente eficientes de redução.

Solucionadores SMT são baseados em técnicas de Satisfatibilidade Booleana (SAT), que vem ganhando espaço para verificação de sistemas como alto número de estados, como processadores computacionais, através da aplicação do Bounded Model Checking (BMC). O BMC é um método de verificação formal que utiliza solucionadores SAT ou SMT, e que consiste em "desenrolar"a FSM dentro de um parâmetro k, limite superior do tamanho de contra-exemplos permitidos. O objetivo da abordagem BMC é focar na falsificação de propriedades, sem uma preocupação inicial grande com a integralidade da análise. No entanto, existem extensões baseadas em SMT e SAT que auxiliam na prova de propriedades temporais.

Metodologia

Para avaliar e eficiência e eficácia de solucionadores BDD e SMT, uma ferramenta foi escolhida para cada técnica (NuSMV e Kind2) respectivamente. Além disso, três casos de teste foram desenvolvidos com base em lógicas reais de SIS da Indústria de Petróleo e Gás e outros campos. Cada teste é composto por uma especificação em formato Matriz Causa e Efeito e uma implementação hipotética com discrepâncias predeterminadas em relação à especificação, visando avaliar um tipo específico de lógica de intertravamento.

Todas as implementações foram traduzidas em Máquinas de Estado Finitas para cada método, e as propriedades temporais foram extraídas diretamente das Matrizes Causa e Efeito. Em ambos os casos a passagem do tempo é representada implicitamente na Máquina de Estado Finita, de modo que cada ciclo do Controlador Lógico Programável (PLC) corresponda a 150ms. Desse modo, o componente *TON* pode ser implementado como um contador, que mantém registro do número de ciclos em que a entrada do temporizador foi mantida ativa.

O verificador escolhido para avaliar performance de BDDs para verificação de lógica de intertravamento de SIS foi o NuSMV (CIMATTI et al., 2002) versão 2.6, que utiliza a o pacote CUDD como mecanismo base. Versões mais recentes incluem solucionadores SAT, mas essa opção foi desativada em todos os testes. O NuSMV utiliza uma linguagem própria para escrita do modelo e propriedades temporais a analisar.

Já para estudar a performance de verificadores com base SMT a ferramenta escolhida foi o Kind2 (CHAMPION et al., 2016). É uma ferramenta mais recente, que opera através da execução paralela de variados mecanismos de solução complementares, como k-induction, IC3 e Geração de Invariantes. O Kind2 utiliza a linguagem Lustre para representação do modelo e propriedades temporais.

No total foram desenvolvidos três testes, cada um com um objetivo específico:

- Análise de tempo de ativação de temporizadores: Para avaliar a influência do tempo especificado de ativação de temporizadores na lógica de intertravamento, um pequeno exemplo com 3 entradas e 2 saídas foi desenvolvido. Nesse sistema, o valor do tempo de ativação dos temporizadores é parametrizado, de forma a permitir que vários cenários com tempos diferentes sejam analisados.

- Análise da interação entre temporizadores e lógica de intertravamento: Para avaliar como os solucionadores operam sobre um sistema com uma lógica de controle mista entre expressões Booleanas e temporizadores, esse teste foi desenvolvido com 6 temporizadores e alto grau de relacionamento entre as 16 entradas e 8 saídas.

- Análise de lógica não temporizada: Para avaliar como os solucionadores se comportam perante um sistema sem lógica temporal, um caso de teste foi elaborado com 12

entradas e 4 saídas, somente com lógicas Booleanas.

Em cada teste foram introduzidos erros propositais para avaliar a eficácia dos métodos em encontrar erros. A comparação foi realizada em termos de acurácia dos resultados, tempo de execução e uso de memória de cada ferramenta.

Resultados e Discussão

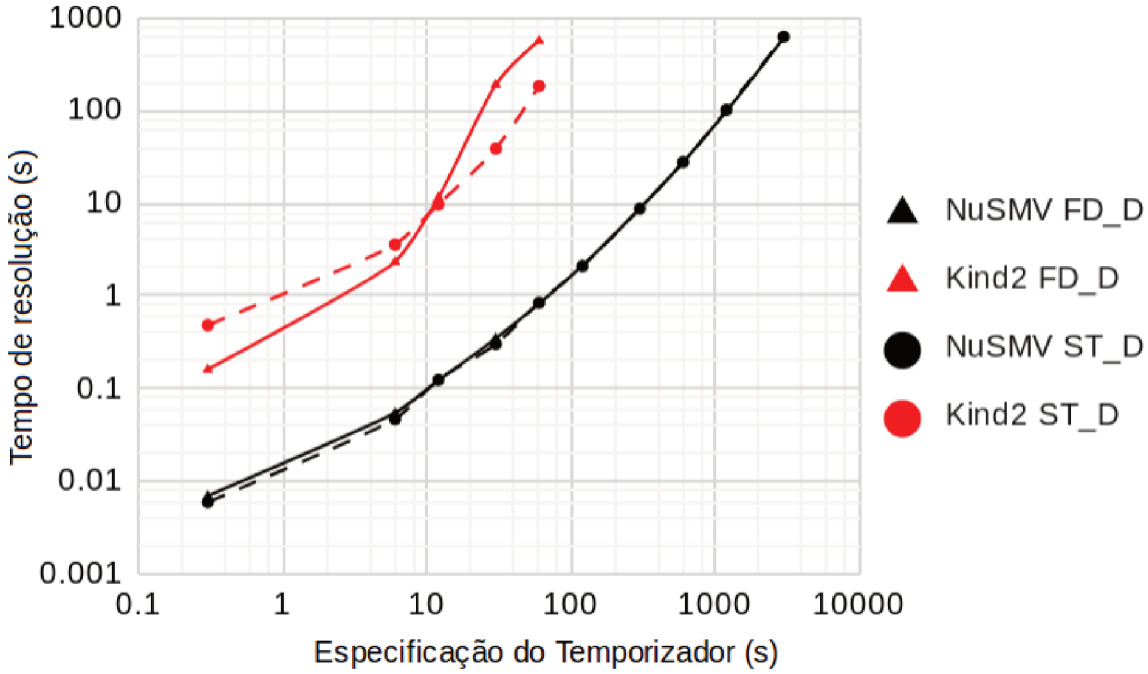
Os resultados do primeiro teste podem ser vistos na Tabela 1 e na Figura 1. Ambas as ferramentas não erraram ao afirmar sobre a validade de alguma propriedade e usaram uma quantidade semelhante de memória computacional. No entanto, percebe-se que o tempo de solução da ferramenta Kind2 para casos com alto valor do tempo de ativação do temporizador foram muito maiores que os obtidos pelo NuSMV (inclusive impossibilitando que o Kind2 completasse a análise dentro do prazo de 20 minutos para cada propriedade). Uma possibilidade é que conforme a razão entre o valor do tempo de ativação do temporizador e o ciclo do PLC (150ms) aumenta, o número de estados a serem percorridos até a falha aumenta consideravelmente, reduzindo a eficiência de solucionador Kind2 baseado em BMC.

Tabela 1 – Comparação dos tempos de solução para o caso teste 3x2. O caractere '-' significa que a ferramenta não foi capaz de avaliar a propriedade dentro do tempo alocado de 20 minutos.

Parâmetros do Modelo		Tempo de Solução			
Especificação do Temporizador (s)	**Implementação do Temporizador (s)**	**NuSMV FD_D**	**Kind2 FD_D**	**NuSMV ST_D**	**Kind2 ST_D**
0.3	0.45	7ms	162ms	6ms	486ms
6	6.6	55ms	2.34s	47ms	3.59s
12	13	123ms	11.57s	125ms	9.78
30	33	350ms	191s	305ms	39.00s
60	66	830ms	564s	849s	184s
120	132	2.13s	-	2.10s	-
300	330	8.81s	-	8.76s	-
600	660	27.76s	-	27.88s	-
1,200	1,320	98.40s	-	102s	-
3,000	3,300	606s	-	628s	-

Fonte: Autor.

Figura 1 – Comparação gráfica dos tempos de solução para o caso teste 3x2. Escalas logarítmicas.



Fonte: Autor.

Os resultados do segundo teste podem ser vistos na Tabela 2 e apresentam características semelhantes aos do primeiro teste. Ambas as ferramentas avaliaram corretamente todas as propriedades, mas novamente a ferramenta Kind2 se mostrou consideravelmente menos eficiente que a NuSMV para casos com temporizadores com tempo de ativação mais alto. Curiosamente, embora a saída J possuísse um tempo de ativação maior que o temporizador da saída I, o tempo de análise das propriedades referentes a I foi significativamente menor. Isso ocorreu pois o mecanismo k-induction da ferramenta Kind2 foi capaz de identificar que a propriedade seria válida sem a necessidade de "desenrolar"as Máquina de Estados Finita sobre o grande espaço de estados. O motivo particular pelo qual o algoritmo k-induction foi mais eficiente nesse caso específico não foi identificado durante o estágio.

Tabela 2 – Comparação dos tempos de solução para o caso teste 16x8. Um veredito de "válida"implica na implementação ser segura em relação ao tipo de falha correspondente.

Parâmetros da Análise		Tempo de Solução		Análise
Sinal de Saída	**Propriedade de segurança**	**NuSMV**	**Kind2**	**Veredito**
G	FD	17ms	106ms	válida
G	ST	19ms	231ms	válida
H	FD	91ms	258ms	falsificável
H	ST	85ms	4.35s	válida
I	FD	134ms	70.87s	falsificável
I	ST	153ms	48.90s	válida
J	FD	273ms	4.13s	válida
J	ST	257ms	4.56s	válida
K	FD	21ms	1.34s	válida
K	ST	14ms	1.17s	falsificável
L	FD	37ms	4.38s	válida
L	ST	38ms	5.07s	válida
M	FD	22ms	4.71s	válida
M	ST	17ms	108ms	falsificável
N	FD	13ms	120ms	válida
N	ST	5ms	133ms	válida

Fonte: Autor.

Os resultados do terceiro teste são apresentados na Tabela 3 e revelam que para ambas as ferramentas, a verificação de Sistemas Instrumentados de Segurança sem lógica temporal não é problema, de modo que todas as propriedades foram verificadas em menos de 150ms. Como o tempo de execução e uso de memória foram muito baixos, o comparativo entre ambas as ferramentas nesse contexto não foi avaliada.

Tabela 3 – Comparação dos tempos de solução para o caso teste 12x4. Um veredito de "válida"implica na implementação ser segura em relação ao tipo de falha correspondente.

Parâmetros da Análise		Tempo de Solução		Análise
Sinal de Saída	**Propriedade de segurança**	**NuSMV**	**Kind2**	**Veredito**
E	FD	3ms	120ms	falsificável
E	ST	3ms	123ms	válida
F	FD	1ms	118ms	falsificável
F	ST	8ms	122ms	válida
G	FD	3ms	138ms	válida
G	ST	3ms	142ms	falsificável
H	FD	8ms	124ms	válida
H	ST	4ms	132ms	válida

Fonte: Autor.

Conclusão

A partir da análise dos resultos dos testes executados neste trabalho, as seguintes conclusões podem ser tomadas com relação às ferramentas estudadas:

- A eficiência de técnicas Bounded Model Checking precisam ser melhoradas para que elas possam ser aplicadas para sistemas com temporizadores com altos valores de tempo de ativação. Maneiras para que isso seja alcançado incluem identificar casos em que as técnicas complementares (k-induction, geração de invariantes, etc.) se mostrem mais abrangentes, reduzindo a sobrecarga de casos não ideais para Bounded Model Checking. Outra opção seria o desenvolvimento de técnicas de representação explícita de restrições temporais via SAT ou SMT;

- A ferramenta NuSMV se mostrou bastante eficaz para Model Checking de Sistemas Instrumentados de Segurança, e embora a construção manual dos arquivos SMV seja massante, o uso de *templates* para geração semi-automática deles facilita o processo de verificação.

A partir disso, sugere-se que as seguintes tarefas sejam concluídas com o objetivo de extender a compreensão da aplicabilidade de cada uma das ferramentas para a verificação da lógica de Sistemas Instrumentados de Segurança:

- Implementação de solucionadores próprios em baixo nível, para avaliar independentemente o benefício de cada método auxiliar, principalmente se tratando de técnicas de

Bounded Model Checking. O aluno deu início ao desenvolvimento de tais ferramentas, mas por falta de tempo dentro do estágio não foi possível um avanço significativo.

- Implementação de um método híbrido com representação temporal explícita, como o utilizado pelo Fiacre/TINA, em que restrições temporais são representadas via um conjunto de restrições por equações.

List of Figures

List of Tables

List of abbreviations and acronyms

BDD	Binary Decision Diagrams
SAT	Boolean Satisfiability
BMC	Bounded Model Checking
CEM	Cause & Effect Matrix
CTL	Computation Tree Logic
FAT	Factory Acceptance Test
FD	Failure on Demand
FSM	Finite State Machine
IEC	International Electrotechnical Commission
LTL	Linear Temporal Logic
MC	Model Checking
PLC	Programmable Logic Controller
ROBDD	Reduced Ordered Binary Decision Diagram
SIF	Safety Instrumented Function
SIL	Safety Integrity Level
SIS	Safety Instrumented System
SMT	Satisfiability Modulo Theory
ST	Spurious Trip
TON	Timer on Activation

List of symbols

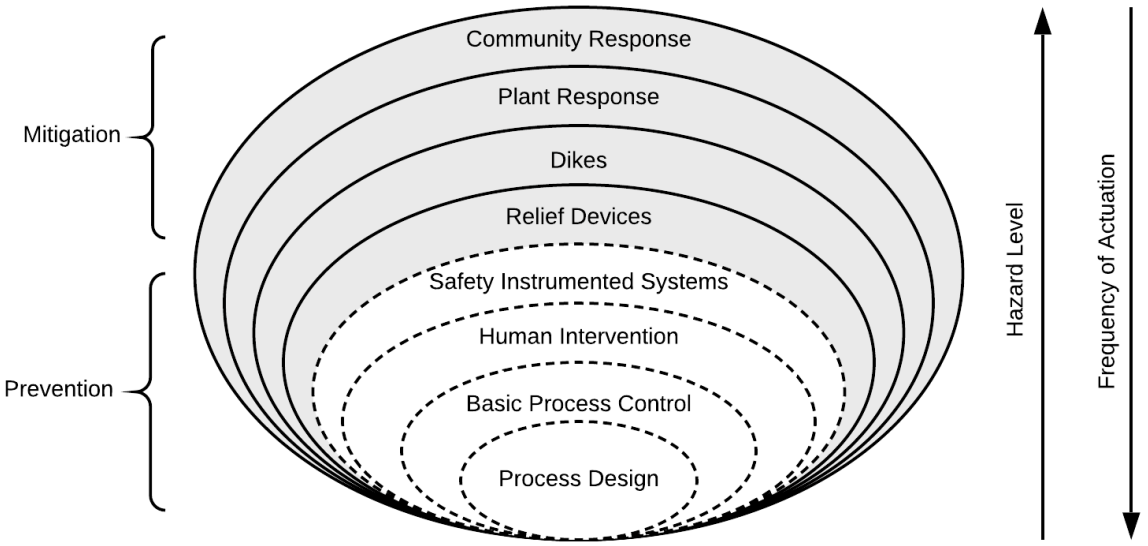
$\neg$	Logical NOT
$\vee$	Logical OR
$\wedge$	Logical AND
$\Longrightarrow$	Logical implication
AG	Always Globally (Globally along all paths)

Contents

1 Introduction

Safety Critical Systems (SCS) are systems that, when malfunctioning, may cause serious injury or death to people, breakage to equipment or harm to the environment (e.g. Offshore Oil & Gas Platforms, Nuclear Plants, etc.). For this reason, it is common to have several layers (Figure 2) of actuation dedicated to either risk prevention (reducing how often it occurs) and damage mitigation (reducing consequences if it occurs) (American Institute of Chemical Engineers, 2001). These layers range from visual warnings in low threat situations (handled by the Basic Process Control) to plant and community responses in case of scenarios with high levels of Safety, Health and Environmental (SH&E) adversities.

Figure 2 – Layers of Protection.



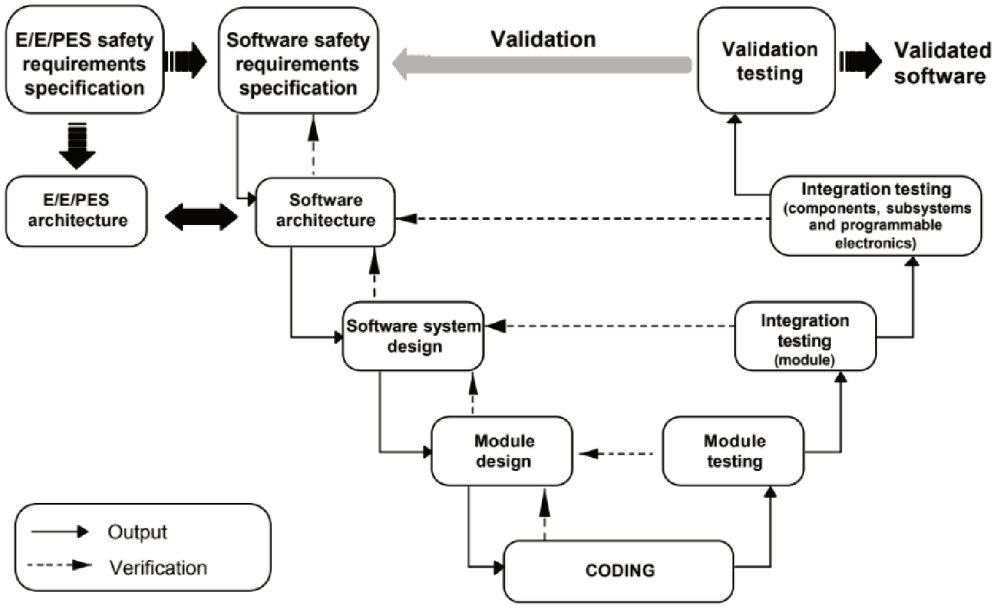
Source: Adapted from (American Institute of Chemical Engineers, 2001).

Safety Instrumented Systems (SIS) are hardware and software mechanisms that operate solely within the scope of prevention, being focused on reducing the probability of occurrence of undesired unsafe events. Moreover, SIS are often the last layer of risk prevention, and as such their design phases require a great level of detail during specification and testing. To assure that the design methodology is able to identify the risks presented by an industrial processes and that the implemented SIS is able to correctly react to undesired situations, industrial standards such as the IEC65108 (IEC61508, 2010) and IEC61511 (IEC61511, 2018) were created.

In the context of Oil and Gas Facilities, SISs are composed by a set of sensors

designed to measure the system's state, one or more Programmable Logic Controllers (PLCs) in which the control logic is implemented and a set of outputs designed to lead the system back to a safe state if required. Due to the critical nature of the conditions in which a Safety Instrumented System must operate, there are several requirements (IEC61508, 2010) that must be fulfilled during the Software Development phases, which are presented in the form of a V-model (Figure 3):

Figure 3 – V-Model of Software Design.



Source: IEC61508 (2010)

Within this model, it can be seen that verification procedures are crucial during development, as the closer to the implementation that an error is found, the greater the number of steps that have to be redone (increasing time spent and costs). The incorrect implementation of a SIS controller can cause two types of errors, which are both highly undesirable due to risking the well being of people, equipment and the environment:

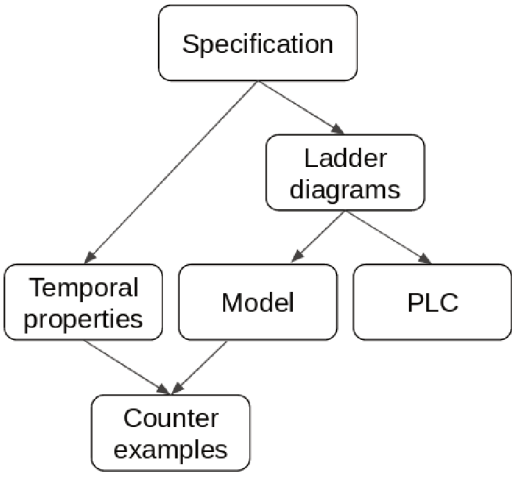
- Spurious Trips (ST): Controller incorrectly activates or maintains active an output signal when it should be deactivated, according to the specification. Also called Safe Failures.
- Failures on Demand (FD): Controller incorrectly deactivates or maintains deactivated an output signal when it should be activated, according to the specification. Also called Dangerous Failures.

Furthermore, since SIS are dormant systems (only actuating on rare occasions), it is difficult to diagnose these errors by only observing the system already in operation.

SIS operate by executing several Safety Instrumented Functions (SIF), each of which has a unique Safety Integrity Level (SIL), based on the probability of occurrence and the severity of the associated risk. The SIL ranges from 1 to 4 and dictates the measures taken to reduce the errors in the implementation of the control logic in a SIS. For this reason, the IEC recommends that systems with SIL of 2 or higher undergo formal verification (IEC61508, 2010).

One example of verification procedures is the application of Formal and/or Semi-Formal methods (Figure 4), in which the control system in question is represented by a mathematical model, whose correctness is evaluated by a verification engine. One such method is called Model Checking, where the engine checks whether properties (extracted from the specification) are satisfied within the model.

Figure 4 – Verification Methods in Safety Instrumented System design.



Source: Author.

Another method that is frequently used in the industry is the Factory Acceptance Test (FAT). The FAT consists of manually checking if the system behaves as expected, on site, during or after implementation. One way this can be done is supplying the system with a set of inputs and verifying if the outputs correspond to what is predicted by the specification (also known as *black-box* testing).

Due to the complexity of the systems in question, an exhaustive analysis through Factory Acceptance Tests is rarely feasible, and generally the level of automation for them is very low. Therefore, Model Checking can also add value to the testing phases, by being able to:

- Exhaustively validate the given model through mathematical proofs;
- Be applied before implementation, reducing the cost of correcting errors.

- Have a good level of automation, speed up testing and reduce manual labor.

Still, it is worth recalling that Model Checking can only be applied over the mathematical model, which in of itself is an incomplete (as possibly incorrect) representation of the system, meaning errors could go unnoticed. Moreover, application of formal methods is not yet a trivial task for most companies, as currently they are often performed only by professionals specifically well versed in its concepts.

When it comes to the actual execution of Model Checking, there are several tools and methods developed over the past decades that may be used, each with its advantages and drawbacks. When it comes to Safety Instrumented Systems, there are two main problems that arise, both of which impact heavily on the complexity of the mathematical model:

- Number of input and output signals: As the number of signals increases, so does the amount of states required to represent every possible combination of them along time.
- Temporal logic: When there are temporal requirements within safety logic (eg. unlock a door 3 seconds after there is no electrical current flowing through a test bench), the passage of time must also be represented within the model.

As the complexity increases with the number of states, the harder it is to check them all for errors, to the point where computationally analyzing each state could take years. Thus a need for efficient methods of formal verification arises, with several tools possibly being able to fill this gap.

Two of these methods are Binary Decision Diagrams (BDD) and Satisfiability Modulo Theory (SMT), both of which are detailed in this document, with their accuracy and efficiency being evaluated via the analysis of hypothetical test cases, which were created by adapting real life SIS logic specifications. These test cases aim to emulate the conditions present in real SIS logic where these two Model Checking tools could have difficulty operating in, such as large input sets and time based logic.

In Chapter 2, the Cause & Effect Matrix is presented, outlining its applications and semantics. In Chapter 3, the concepts of Model Checking for SIS are detailed further, along with the description of two Symbolic Model Checking Methods. Then, in Chapter 4, the methodology of testing and result analysis of the comparison between both methods is presented. Lastly, a conclusion of the work done during the internship and suggestions for further development are outlined.

2 Cause & Effect Matrix

In this chapter, the core ideas and semantics behind the Cause Effect Matrix (CEM) are presented, together with the motivation as to why it is an adequate method to represent the specification for the logic of a SIS.

As seen on Figure 3, the Software safety requirements specification is very important during the design, as all following steps are based upon it. Thus, it is essential that whoever designs the safety logic has a clear, unambiguous, reliable and easy to understand format in which to specify it in. In this context, Cause & Effect Matrices (CEM) are a way to represent interlock logic between a large number of input and output signals in a visual yet compact way. While the exact semantics differ between applications, its simple structure stays similar while aiding the communication of the safety logic requirements between the process engineers, PLC programmers and safety experts.

As exemplified by Figure 5, the CEM consists of a set of rows (input signals or causes), a set of columns (output signals or effects), while the intersection between each pair represents the activation logic of the output by a given input. This structure is adequate for representing stateless logic, while also being capable of representing timed constraints within the control logic, which makes them suitable for designing SIS control logic. Moreover, the ambiguity of the specified logic when written in this format is is very low, albeit errors can still occur during the later steps of Software design.

Figure 5 – Example of a Cause and Effect Matrix. Rows represent causes (inputs) and columns represent effects (outputs).

	Effect	E	F
Cause			
A		X	
B		T20	N
C		A1	X
D		A1	

Source: Author.

The combination of Boolean relationships and timed logic constraints is able to represent most SIS interlock logic requirements. However, in situations where the safety

functions contain a high amount of state based logic, the CEM formalism might not be recommended. Examples of cases in which this method is used range from Offshore Oil Extraction facilities (VEIGA et al., 2017) to test benches for particle accelerator equipment (FERNANDEZ et al., 2019).

It is worth noting that only digital inputs can be used as causes and effects in a CEM. To use an analog input as a cause in a matrix, one must instead use a digital signal that is active when the value of the analog signal is within a certain range.

2.1 Semantics

The semantics of the CEM vary greatly between companies and industry types, with the IEC recently publishing guidelines for use in engineering activities (IEC62881, 2018). The CEM format and semantics in this report were based on the ones used by Petrobras, by the American Petroleum Institute and in Lazaro (2018), respecting the following rules:

- X: A cause with this entry, when active, activates the output (*OR logic*).
- N: A cause with this entry, when inactive, activates the output (*OR NOT logic*).
- Ai: For $i = 1, 2, ...$, the effect will be triggered when any group i satisfies its activation conditions, given by an *AND* logic between the causes marked with the Ai entry AND an *AND* logic between the negation of each cause marked with the NAi entry.
- Tn: The cause related to this entry must be active for n seconds/milliseconds to activate the corresponding effect (TON logic as described in IEC61131 (2013)).
- A logic in *{Ai, NAi}* (*prefix*) can be combined with Tn (*suffix*), so that the group i will check the timed version of the cause signal (eg. A1T20s).
- XooY: The cause in the corresponding row is the result of a vote of X out of Y sensors within the voting group.

In this case the logic represented by the CEM in Figure 5 can be expressed as:

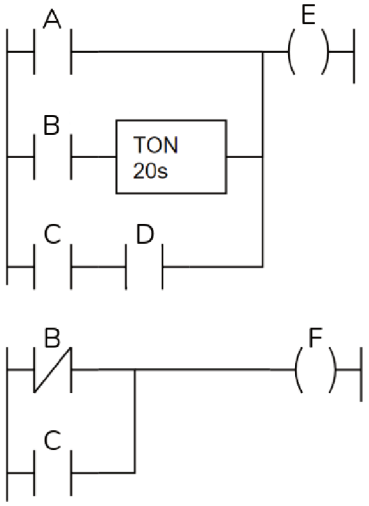
$$\begin{bmatrix} E \\ F \end{bmatrix} = \begin{bmatrix} A \vee TON(B, 20s) \vee (C \wedge D) \\ \neg B \vee C \end{bmatrix} \tag{2.1}$$

These expressions can be written as:

- The output signal E should be active when any of the following three situations are satisfied. The first is whenever the input signal A is active, the second is if the input B has been active without interruption for at least 20 seconds and the third is if the input signals C and D are active simultaneously. If none of the conditions are met, E should be inactive.
- The output signal F should be active when either B is not active or C is active. When these conditions are not met, F should be inactive.

As seen on Figure 5, the CEM formalism provides an outlook of the logic complexity and the input-output relationships at a glance, which is very useful for systems with large sets of sensors and actuators. The logical relationships between causes and effects can also be easily translated into PLC Ladder code, reducing the probability of errors in the programming phase. The ladder code representing the implementation of the logic described in Figure 6 is:

Figure 6 – Ladder code implementation of Cause and Effect Matrix logic.



Source: Author.

In Figure 6 it can be observed that the translation of a CEM specification into a PLC ladder code is quite direct. This aids in the reduction of errors introduced in the programming phase, as well as making the CEM a formalism that professionals in the industry have more ease of adapting into their design methodologies.

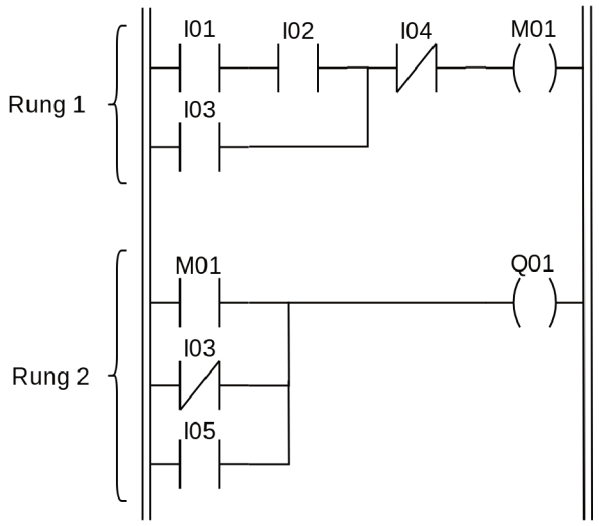
3 Model Checking

In this chapter, the rules of model creation from the Ladder Diagram and property extraction from the Cause & Effect Matrices are presented. Furthermore, the Model Checking techniques analyzed during the internship are described in further detail, as well as the tools that implement them.

3.1 Model Definition

To model the implemented controller, we must take into consideration the platform in which it will run. For the vast majority of Safety Instrumented Systems in the industry, that would be a PLC or a Safety PLC, and thus we will model each test case implementation as a code running in a PLC. For that, we must describe the hypothetical implementation in a format which is coherent with the platform where it would run on, and so, the Ladder Logic Notation was chosen, as it is the main language in which Safety PLCs are programmed in the industry. An example of Ladder Notation can be seen in Figure 7:

Figure 7 – Ladder diagram example.



Source: Author.

To capture the behavior of the PLC within the model, we must take into account how it behaves in the real world. First of all, a PLC can be described as a synchronous system, which means its operations are coordinated by a clock and all of the steps run cyclically in what is called a *scan cycle*. There are 5 steps that are executed in each *scan cycle*:

- 1. Input Reading: Here, the PLC takes the values of all signals considered as inputs (sensors) and stores them into an input memory, which will be held constant until the next reading, regardless of whether the signals change during execution of the remaining 4 steps;
- 2. Program Execution: In this step, the current internal states and temporary input states will be used to calculate the internal states and temporary output states for the next iteration. This is where the control logic is implemented, with each *rung* (as seen on Figure 7) being executed in order from top to bottom;
- 3. Handling communication: In this step, the PLC reads incoming communication requests, processes them and send messages to whoever requested. This is how integration with other controllers or supervisors is done in practice, and is not directly related to the control logic.
- 4. Run CPU diagnostics: Here, information about the physical state of the PLC will be stored in a buffer to be read if necessary.
- 5. Write outputs: Here, the values stored inside the temporary output memory during step 2 will be written into the actual outputs of the PLC, activating actuators of the control systems.

The cycles of the Ladder diagram in Figure 7 would then follow the simplified sequence below:

while *true* **do**
    (I01, I02, I03, I04, I05) := read_inputs(); `# Step 1`
    M01 := ((I01 & I02) | I03) & !IO4; `# Step 2, rung 1`
    Q01 := M01 | !I03 | I05; `# Step 2, rung 2`
    handle_communication(); `# Step 3`
    run_diagnostics(); `# Step 4`
    set_outputs(Q01); `# Step 5`
end

The duration of the scan cycles may vary due to the model of the PLC, the size of the code it is running and other external (and possibly random) factors. It is however safe to assume that all operations should be executed in the order of milliseconds if there are no errors or exceptions during code execution.

If one was to attempt to model every single aspect of the PLC behavior, the model would be far too complex to be able to extract useful information regarding to the control logic. Consequently, abstractions must be made, where parts of the system behavior are

omitted from the model, in a way that it captures only what is of interest in the analysis. When verifying an implementation through model checking, one must be wary of this, as it means errors that can happen in the real world might not be apparent within the model.

When modeling the PLC behavior during the internship, the following abstractions were made:

- Only steps 1, 2 and 5 described earlier are taken into account by the model, as they are responsible for the control logic, with a variable being added solely for the purpose of keeping track of the current step of the cycle;
- In addition, an extra step is added to the model, in which the variables representing the PLC's states are not updated. This extra step is virtual in the sense that it has no correspondence to the real world, but is when we analyze the safety properties, since we are not interested whether the implementation respects the specification in the intermediate steps of the cycle;
- All variables begin execution with a value of *0* (inactive), and are then calculated in the first cycle (similar to PLC behavior);
- At the beginning of each cycle, all inputs can assume values of either *0* (inactive) or *1* (active), creating branches in the state space.
- Constant scan cycle duration of 150ms. This is helpful when the control logic contains timed restrictions;
- Timers are represented as finite counters that count the number of cycles in which their *enable* signal has been active, and compare it to how many cycles are needed for activation.

This structure belongs to the class of transition systems, where the present information is stored as a state (or combination of states) and the change from one step to the other is called a transition. The structure described above can be represented by a Finite State Machine (FSM), or more formally, as a Kripke Structure, with the following definition:

$$M = (S, I, T, L) \tag{3.1}$$

where S is a finite set of states, $I \subseteq S$ is a finite set of initial states, $T \subseteq S \times S$ is a transition such that $\forall s \in S : \exists s' \in S : (s; s') \in T$, and $L : S \to 2^{AP}$ is the labeling function, which assigns observations (propositional variable values) to each state in the system. The set $R \subseteq S$ is the set of reachable states, or, in other words, the set of states such that there is at least one valid path of transitions from the initial state I.

In this context, M is a model both in the mathematical sense and in the engineering sense, where it serves as a simplified mock-up of the real system in which the desired properties can be more easily verified.

3.2 Property Extraction

In Model Checking, there are two main groups of properties to be evaluated:

- Safety Properties: Describe properties that should always hold, or conversely, never hold within the set of reachable states.
- Liveness Properties: Describe something that should eventually hold, or conversely, eventually not hold within the set of reachable states.

These properties are defined as propositions, which can assume values of either *True* or *False* and depend only on the observations assigned to each state by the labeling function L. With this structure, we can use temporal logic such as Linear Temporal Logic (LTL) or Computation Tree Logic (CTL) to represent the desired properties of the SIS, which can be extracted directly from the Cause & Effect Matrices that define the specification.

When evaluating the implementation of PLC controllers for SIS, we are more interested in analyzing safety properties, as the objective is to know if there is a state in which there is a Spurious Trip or Failure on Demand. Thus, we can encode these two types of failures as propositions, and verify whether they are satisfied or not satisfied in any reachable state. If they are not, then we know that the system as described by the model M is coherent with the specification.

To extract the properties from the specification, the following proposition templates for the two failure types were used:

$$FD_i = cause_i \wedge \neg effect_i \tag{3.2}$$

$$ST_i = \neg cause_i \wedge effect_i \tag{3.3}$$

where, for each output i of the specification, FD_i determines a failure on demand for a given state of the model and ST_i does the same but for spurious trips. The term $effect_i$ corresponds to the value of the output signal i (effect) in a given state and the term $cause_i$ corresponds to the logic that expresses the activation of i in regards to the input signals (causes), according to the Cause & Effect Matrix.

Thus, a failure on demand is defined as the output signal, calculated by the implemented controller, assuming the value *false* while, simultaneously, its activation expression given by the CEM assumes the value *true*. Conversely, a spurious trip is defined as the output signal, calculated by the implemented controller, assuming the value *true* while, simultaneously, its activation expression given by the CEM assumes the value *false*.

However, as mentioned earlier, we only want to analyze these properties if the cycle is in a virtual step between updating the outputs and reading the inputs, since we do not care about incoherence in the intermediate steps. Thus, a new variable *write*, which represents that the model is in that step, is added to 3.2 and 3.3, resulting in:

$$FD_i = write \wedge cause_i \wedge \neg effect_i \tag{3.4}$$

$$ST_i = write \wedge \neg cause_i \wedge effect_i \tag{3.5}$$

The propositions 3.4 and 3.5 will only evaluate over the initial state if used in a model checker. Therefore, we must use some form of temporal logic notation to express that these propositions should never be true in any state in the set of reachable states. For this situation we used the temporal operators AG from CTL to encode the propositions that represent the properties that we want to evaluate, in the following manner:

$$FD\_Safe_i = AG\neg(write \wedge cause_i \wedge \neg effect_i) \tag{3.6}$$

$$ST\_Safe_i = AG\neg(write \wedge \neg cause_i \wedge effect_i) \tag{3.7}$$

where FD_Safe_i and ST_Safe_i indicate that, in the model, the output i will never have a failure on demand or spurious trip, respectively.

3.3 Obstacles for Model Checking

In most cases of real world Safety Instrumented Systems, the extensive enumeration of states of a controller model is not viable, due to the great amount of input and output signals and necessity of representing temporal logic constraints. This is known as the "state explosion problem", and in an attempt to solve it, many approaches have been developed aiming to evaluate properties without building the complete state transition graph.

One such technique is Symbolic Model Checking, where the model is written and evaluated in a symbolic representation, in which the state graph is implicitly described in a propositional quantifiable logic formula. This allows for a more efficient analysis of the model's properties.

3.4 Binary Decision Diagrams

Binary Decision Diagrams (BDD) are data structures in the format of acyclic graphs used to represent Boolean expressions (AKERS, 1978). However, BDD is a term that almost always refers to Reduced Ordered Binary Decision Diagram (ROBDD), which is a canonical representation for a given function and ordering. This method can be used for symbolic formal verification by encoding the FSM in a Boolean expression rather than an exhaustive list of states. In this case, the Boolean expression itself can be manipulated, greatly reducing the number of operations needed to verify the complete model.

3.5 Satisfiability Modulo Theory

Boolean Satisfiability (SAT) solvers work by evaluating the satisfiability of a set of Boolean expressions, being considered symbolic methods in the sense they operate directly over Boolean expressions. However, unlike BDD they do not use canonical forms and do not suffer from state explosion (BIERE et al., 2003), which has led to an increased growth of its application in the industry, such as in the context of processor hardware verification.

Satisfiability Modulo Theory (SMT) extends SAT through an assortment of first-order theories and by increasing efficiency through several optimization tools and theory solvers before using a SAT solver to evaluate literal assignments (de MOURA; BJØRNER, 2008). In addition, an international initiative has developed a standardized input and output language for SMT solvers, called SMT-LIB (BARRETT; STUMP; TINELLI, 2010), which has aided in the rise of the method.

3.5.1 Bounded Model Checking

Bounded Model Checking (BMC) is a verification method based on Boolean Satisfiability (SAT) solvers, which consists of unrolling the FSM inside a parameter k, the upper bound of the length of a counter example. As SAT does not possess the ability to eliminate variables, the goal of the BMC approach is to focus on falsifying properties without an initial regard for completeness of the analysis. There are however extensions that can help prove temporal properties (BIERE et al., 2009), such as k-induction (SHEERAN; SINGH; STåLMARCK, 2000) and IC3 predicate abstraction (CIMATTI et al., 2013). Recent tools (CHAMPION et al., 2016) have successfully implemented the BMC method using a Satisfiability Modulo Theory (SMT) engine.

4 Comparison of Verification Methods and Tools

In this chapter, the methodology for selection and implementation of each of the three tests is shown, with their results following accordingly. For each test, two tools (NuSMV and Kind2) were used to test the accuracy and efficiency of BDD and SMT based methods, respectively. It was found that while NuSMV was able to solve all cases without much trouble, Kind2 had shortcomings when analyzing systems containing timers with high activation values.

For evaluating the efficiency and accuracy of BDD and SMT based solvers, one tool was picked for each method and three test cases were devised, adapted from real logic of Safety Instrumented Systems from the Oil & Gas Industry and other domains. Each test consists of a CEM specification and an implementation with predetermined discrepancies against the specification, and seeks to evaluate a different type of SIS logic.

All implementations were translated into FSM models for each method, and the temporal properties were extracted directly from the Cause and Effect Matrices. In both cases the passage of time is represented implicitly within the FSM, in a way that every cycle of the PLC corresponds to 150ms. This way, the *TON* component can be implemented as a counter, which keeps track of the number of cycles for which the input of the timer is active.

4.1 NuSMV

The chosen model checker to evaluate BDD performance when verifying SIS interlock logic was NuSMV (CIMATTI et al., 2002) version 2.6, which uses the CUDD solver package as the BDD base engine. More recent versions of NuSMV also allow for SAT based Bounded Model Checking, but for the purposes of comparing SAT and BDD based methods, this option was turned off in all tests.

NuSMV has its own input language, described in the user manual (NUSMV, 2010), but it not designed to represent synchronous reactive systems. For this reason, the sequential behavior of the PLC was implemented manually into the model file. NuSMV uses a single processor thread to perform the operations. For implementing the tests, a SMV template and a Python 3 script were developed, to avoid having to manually write an unique file for every variation of each test and to allow batch execution of tests. The SMV templates contain pre-generated patterns for timers, voting blocks and representation of the cyclical process of PLC input sweep, internal memory calculation and output updates,

which are all filled in by the Python 3 scripts before execution by NuSMV.

The basis of the model representation into the input language of NuSMV consists of declaring constants, variables and modules (which encapsulate behavior for modularity and can be instantiated), where the module named *main* is the entry point for execution. Then, the transitions are written by using the "$next(x) := y$" command, where x is the variable name and y is its value for the next iteration. For variables that represent inputs, the transition is written as "$next(x) := \{FALSE, TRUE\}$", indicating that at each new cycle they may assume either value. The variable *step* has the role of keeping track of which step in the scan cycle the system is at any given moment. The safety properties to analyze during the execution are given by the command "*SPEC proposition*", where *proposition* is the property extracted from the CEM as in equations 3.7 and 3.6.

An example of a SMV template containing all functionality used during the tests can be found in Appendix A.

4.2 Kind2

For evaluating the performance of SMT based model checkers, Kind2 (CHAMPION et al., 2016) was chosen. It operates by running an assortment of complementary verification techniques in parallel, such as BMC, k-induction, IC3, invariant generation and others. It uses the Z3 solver (de MOURA; BJØRNER, 2008) as the default SMT solver. The input language is Lustre, a synchronous and formally defined data-flow language. Similarly to the NuSMV case, Lustre templates and Python 3 scripts were developed to aid in the execution of tests.

Since Lustre is designed for programming reactive systems, the representation of the sequential nature of the scan cycles is far simpler and more intuitive than with NuSMV. Within the file, the constants are declared and the variables are declared within the nodes, which contain input and outputs and perform operations. Unlike in NuSMV, the node that is the point of entry is determined by the command "*- -%MAIN;*".

For Kind2, the safety properties are expressed differently than for NuSMV. The entire behavior of the PLC is encapsulated into a node called *scancycle* and furthermore there is no need to use a *write* variable, since the Lustre language is by itself used to describe synchronous systems. Within the main node the properties are written as follows:

$$FD_i = cause_i \implies effect_i \tag{4.1}$$

$$ST_i = effect_i \implies cause_i \tag{4.2}$$

where i once again indicates the effect. The safety properties to be analyzed must then be

specified via the commands "- -*%PROPERTY FD_i;*" or "- -*%PROPERTY ST_i;*".

An example of a Lustre template developed during the internship can be found in Appendix B.

4.3 Test Cases

All tests were run on Ubuntu 18.04 using a local installation of each tool, with 2GB of available memory and a time limit of 20 minutes per property. For both tools 4 processor cores were made available, with Kind2 using all of them and NuSMV using only a single one.

4.3.1 3x2

This test consists of a small interlock logic with 3 inputs (A, B and C) and 2 outputs (D and E), and has the goal of evaluating timer activation thresholds on the time required to analyze the model. The specification in CEM format can be seen in Figure 8 and contains two timers, one of which has a parameterized time of activation n that will be different for every test scenario.

Figure 8 – Cause & Effect Matrix Specification for the 3x2 test case. Underlined cells represent mistakes intentionally introduced into the implementation.

Cause / Input Tag \ Effect / Output Tag	D	E
A	A1	A1
B	A1TON300	X
C	Tn	NA1

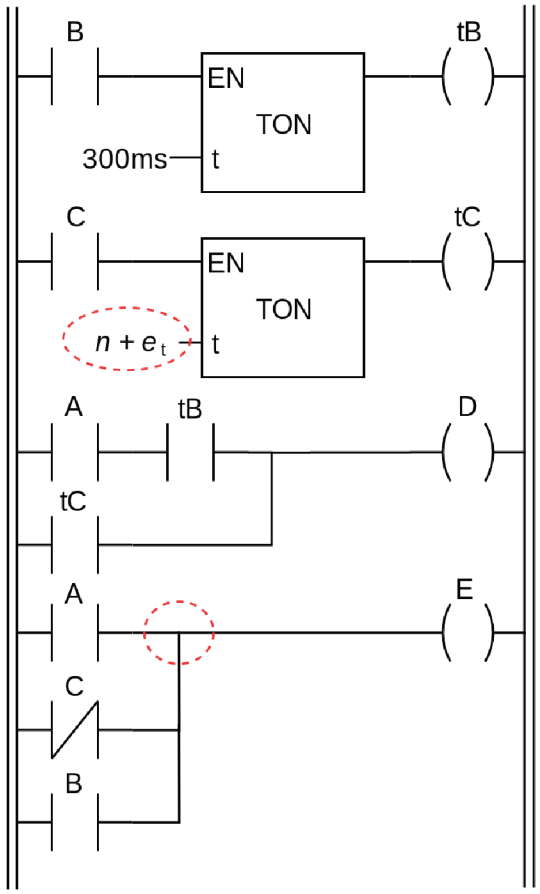
Source: Author.

The expression extracted from the CEM is as follows:

$$\begin{bmatrix} D \\ E \end{bmatrix} = \begin{bmatrix} A \wedge TON(B, 300) \vee TON(C, n) \\ A \wedge \neg C \vee B \end{bmatrix} \tag{4.3}$$

The ladder implementation for this test is show in Figure 9:

Figure 9 – Ladder diagram for faulty implementation of test case 3x2. Intentional mistakes in the logic are circled red.



Source: Author.

Which can also be written as:

$$\begin{bmatrix} D \\ E \end{bmatrix} = \begin{bmatrix} A \wedge TON(B, 300) \vee TON(C, n + e_t) \\ A \vee \neg C \vee B \end{bmatrix} \tag{4.4}$$

There are two discrepancies between the implementation and the specification in this scenario. The first is the addition of the term e_t to the threshold of activation of the timer corresponding to the input C. This term is expected to cause the timer to fire later than usual, allowing for an undesirable situation in which the output D is not triggered when it should (Failure on Demand). The second discrepancy is in the operator between A and $\neg C$, which is an AND ($\wedge$) in the specification and an OR ($\vee$) in the implementation. This should cause the output E to trigger unnecessarily in certain situations, characterizing a Spurious Trip.

Several scenarios, each with its own pair (n, e_t), were submitted to evaluation by the NuSMV and Kind2 model checkers. It is expected that with greater values of n, the

diameter of the FSM (distance of the farthest reachable state from the initial state) should also grow, and thus the time taken to prove or disprove the properties should increase.

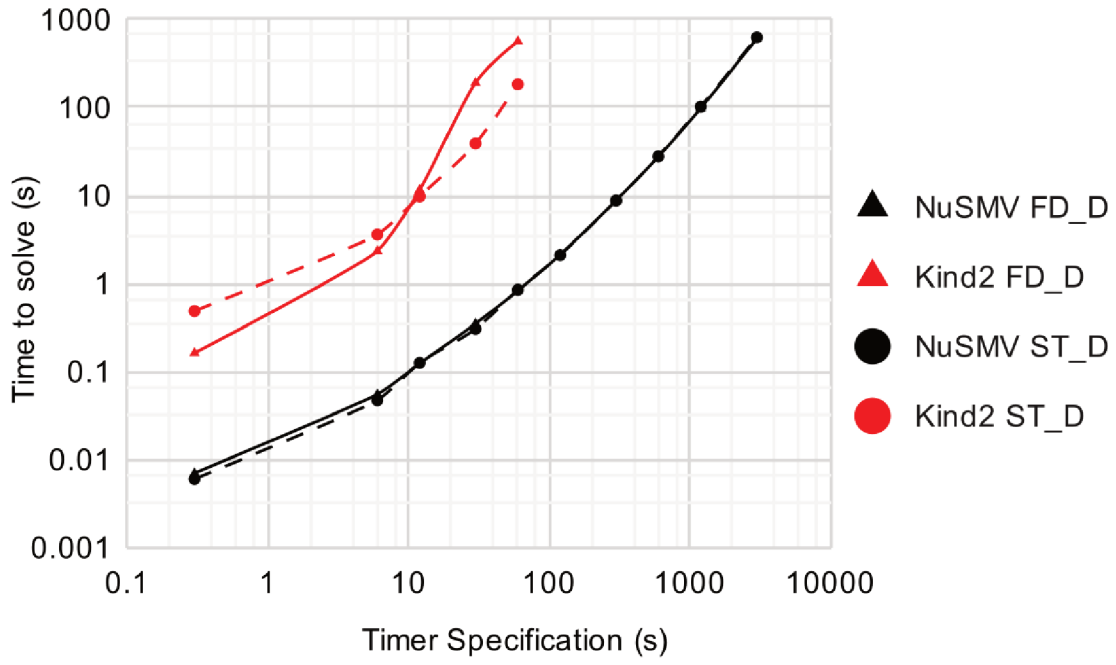
The results of several scenarios can be seen in Table 4 and Figure 10.

Table 4 – Comparison of solving times for the 3x2 test case. The character '-' identifies that the solver was unable to evaluate the property within the allocated time of 20 minutes.

Model Parameters		Solving Time			
Timer Specification (s)	**Timer Implementation (s)**	**NuSMV FD_D**	**Kind2 FD_D**	**NuSMV ST_D**	**Kind2 ST_D**
0.3	0.45	7ms	162ms	6ms	486ms
6	6.6	55ms	2.34s	47ms	3.59s
12	13	123ms	11.57s	125ms	9.78
30	33	350ms	191s	305ms	39.00s
60	66	830ms	564s	849s	184s
120	132	2.13s	-	2.10s	-
300	330	8.81s	-	8.76s	-
600	660	27.76s	-	27.88s	-
1,200	1,320	98.40s	-	102s	-
3,000	3,300	606s	-	628s	-

Source: Author.

Figure 10 – Comparison of solving time for the 3x2 test case. Logarithmic scales.



Source: Author.

As the existence of a ST of E (ST_E) and non existence of a FD of E (FD_E) were asserted consistently faster than 300ms for all scenarios and for both model-checkers, those results were omitted from Table 4 and from Figure 10. The maximum memory usage during execution was 1103MB with NuSMV and 1438MB with Kind2 (sum of the memory used by each instance of Z3).

When both methods were able to assert the properties within the given time restriction of 20min, they both returned the same results in regards to the safety properties, which also coincided with the results predicted during the test case design. However, in this test, it is clear that NuSMV achieved a considerably better performance than Kind2, being able to solve the scenarios in a much smaller time frame. This might be due to that as the ratio n by the cycle duration of 150ms increases, the number of steps to reach the time of activation of the TON component increases drastically, thereby reducing the efficiency of the BMC based Kind2 tool.

4.3.2 16x8

The second test is composed of 16 inputs (one individual input and 5 groups of 3 input voting groups), 8 outputs and 6 timers in the interlock logic. The CEM containing the specification can be seen on Figure 11:

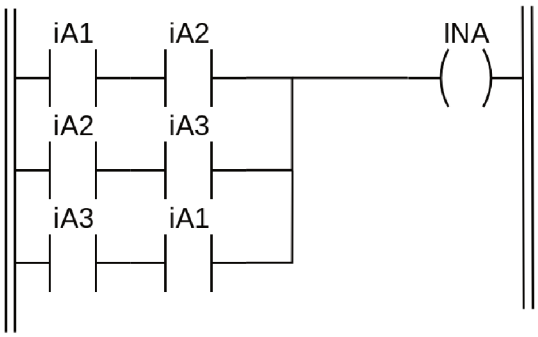
Figure 11 – Cause & Effect Matrix Specification for the 16x8 test case. Underlined cells represent mistakes intentionally introduced into the implementation.

Cause: Input Tag	Cause: Voting	Effect: Output Tag G	H	I	J	K	L	M	N
A1	1oo3					X			
A2	2oo3	X	T15	A1	A1				
A3	3oo3						A1	A1	A1
B1	1oo3					X			
B2	2oo3	X	A1	T15	A2				
B3	3oo3						A1	A2	A1
C1	1oo3					X			
C2	2oo3	X	A2	A1	T24				
C3	3oo3						A1	A2	A2
D1	1oo3								
D2	2oo3	X	A1	A2	A2	T6			
D3	3oo3						A1	A1	A2
E1	1oo3					X			
E2	2oo3	X	A2	A2	A1		T15		
E3	3oo3								
F1	-	N				N		T6	

Source: Author.

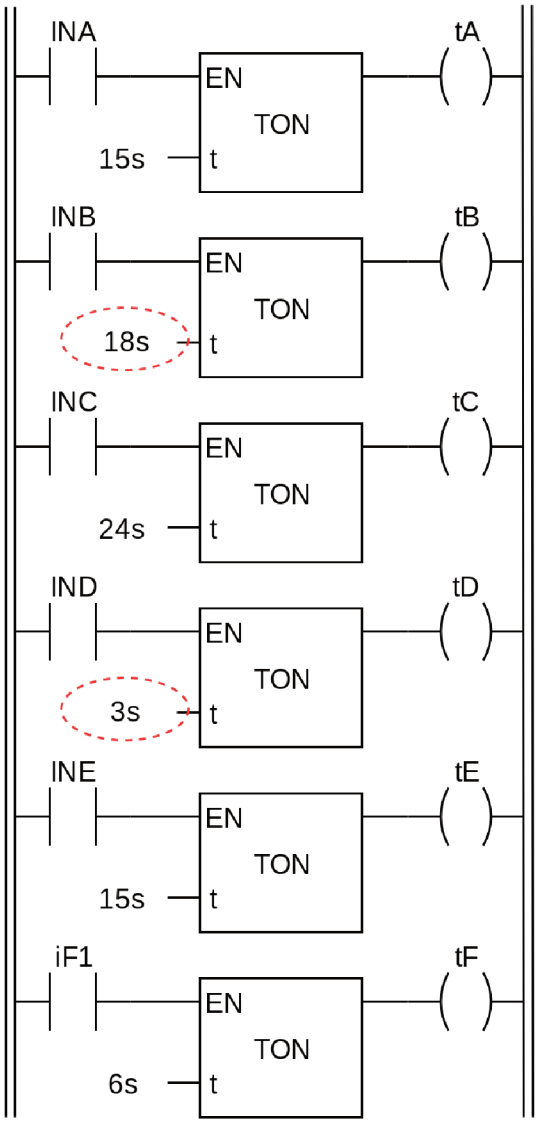
In Figures 12, 13 and 14, snippets showing the faults in the implementation code can be seen in Ladder diagram format:

Figure 12 – Ladder diagram for voting logic of test case 16x8. This code is repeated for inputs *B*, *C*, *D* and *E*.



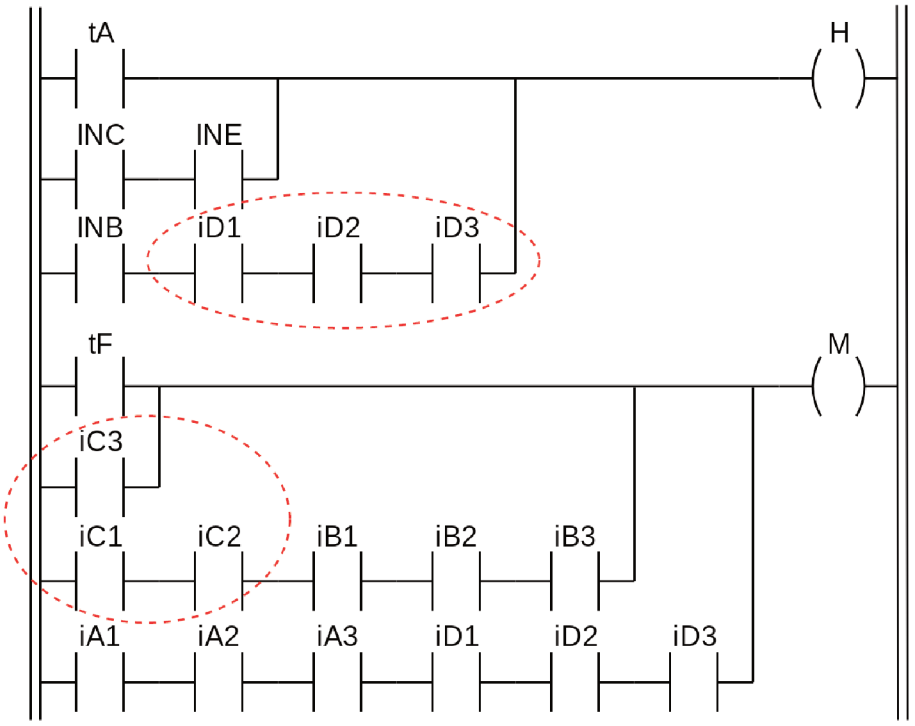
Source: Author.

Figure 13 – Ladder diagram for faulty timer logic of test case 16x8. Intentional mistakes in the logic are circled red.



Source: Author.

Figure 14 – Ladder diagram for faulty combinatorial logic of test case 16x8. Intentional mistakes in the logic are circled red.



Source: Author.

The goal of this test case is to evaluate the impact of multiple timers in conjunction with Boolean relationships as one would expect to find within a typical SIS specification. In the implementation, four errors were introduced, two of which are expected to cause Spurious Trips (ST_K and ST_M) and the other two are expected to cause Failures on Demand (FD_H and FD_I). The results of the verification can be seen in Table 5:

Table 5 – Comparison of solving times for the 16x8 test case. A verdict of valid implies that the implementation is safe in regards to the corresponding failure type.

Model parameters		Solving Time		Result
Output Signal	**Safety Property**	**NuSMV**	**Kind2**	**Verdict**
G	FD	17ms	106ms	valid
G	ST	19ms	231ms	valid
H	FD	91ms	258ms	falsifiable
H	ST	85ms	4.35s	valid
I	FD	134ms	70.87s	falsifiable
I	ST	153ms	48.90s	valid
J	FD	273ms	4.13s	valid
J	ST	257ms	4.56s	valid
K	FD	21ms	1.34s	valid
K	ST	14ms	1.17s	falsifiable
L	FD	37ms	4.38s	valid
L	ST	38ms	5.07s	valid
M	FD	22ms	4.71s	valid
M	ST	17ms	108ms	falsifiable
N	FD	13ms	120ms	valid
N	ST	5ms	133ms	valid

Source: Author.

The maximum memory usage during execution was 1430MB with NuSMV and 1014MB with Kind2 (sum of the memory used by each instance of Z3).

Both methods accurately identified the errors within the implementation. Even though both solvers managed to solve the problem in an acceptable amount of time, it can once again be observed that long times of activation of timers have, above any other considered variables, a clear negative effect on the efficiency of BMC based methods.

Curiously, even though the output J depends on a timer with activation at 24s, its safety properties were asserted considerably faster than the ones of the output I, which depends on a timer with an activation at 15s according to the specification and 18s in the implementation. This occurred because the k-induction engine in Kind2 was able to more quickly identify that the properties would be valid without needing to unroll the FSM over the large bound diameter. The specific reason why the k-induction algorithm was more effective in this particular case was not identified during the internship.

4.3.3 14x4

The third and final test consists of 12 inputs, all of which belong to voting groups, 4 outputs, and no timers in the interlock logic. The goal of this test case is to identify

differences between the model checking methods for cases where there is no complexity introduced in the FSM by timed constraints, as it is also common for a module within a SIS to not include any timed logic. The CEM specification can be seen in Figure 15:

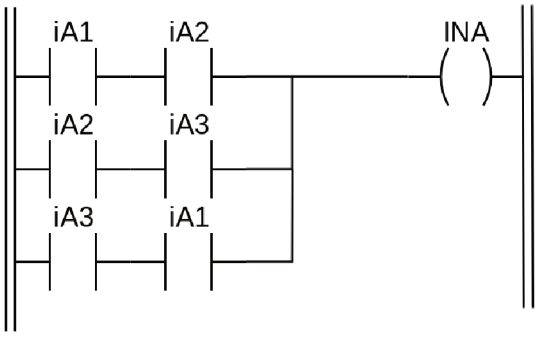
Figure 15 – Cause & Effect Matrix Specification for the 12x4 test case. Underlined cells represent mistakes intentionally introduced into the implementation.

Cause: Input Tag	Cause: Voting	Effect: Output Tag E	F	G	H
A1	1oo3				
A2	2oo3		A1		N
A3	3oo3			N	
B1	1oo3	X			
B2	2oo3		A2		A1
B3	3oo3			X	
C1	1oo3	X			
C2	2oo3		A2		A1
C3	3oo3			X	
D1	1oo3	X			X
D2	2oo3		A1		
D3	3oo3				

Source: Author.

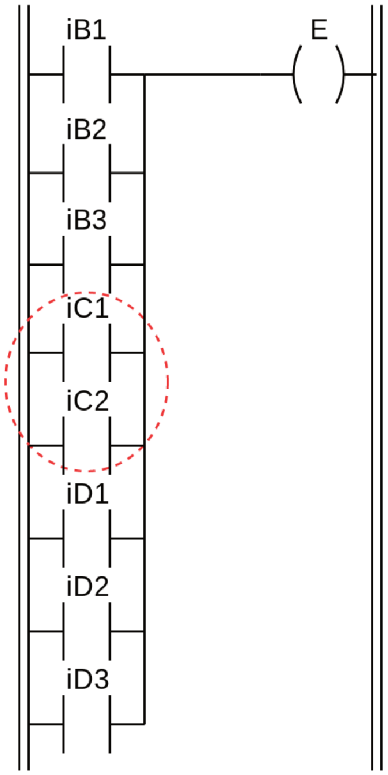
The Ladder diagrams containing the faulty code can be seen in Figures 16, 17 and 18:

Figure 16 – Ladder diagram for voting logic of test case 12x4. This code is repeated for inputs *B*, *C* and *D*.



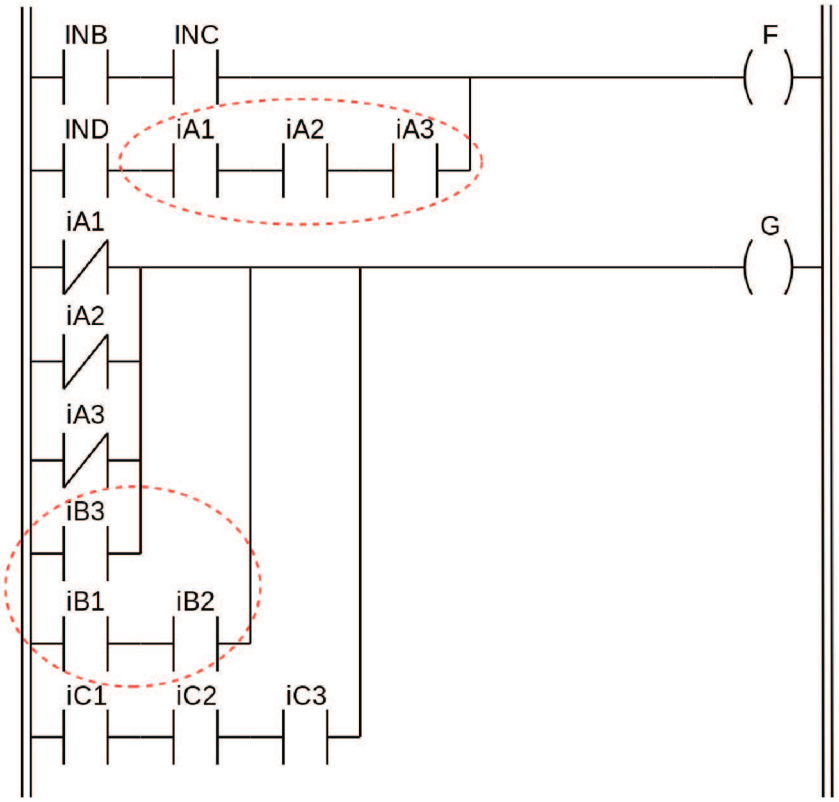
Source: Author.

Figure 17 – Ladder diagram for faulty combinatorial logic of output E in test case 12x4. Intentional mistakes in the logic are circled red.



Source: Author.

Figure 18 – Ladder diagram for faulty combinatorial logic of outputs F and G in test case 12x4. Intentional mistakes in the logic are circled red.



Source: Author.

In the implementation, a total of 3 errors were introduced, two of which are expected to cause Failures on Demand (FD_E and FD_F) and one of which should cause a Spurious Trip (ST_G). The results of the verification can be observed in Table 6:

Table 6 – Comparison of solving times for the 12x4 test case. A verdict of valid implies that the implementation is safe in regards to the corresponding failure type.

Model parameters		Solving Time		Analysis
Output Signal	**Safety Property**	**NuSMV**	**Kind2**	**Verdict**
E	FD	3ms	120ms	falsifiable
E	ST	3ms	123ms	valid
F	FD	1ms	118ms	falsifiable
F	ST	8ms	122ms	valid
G	FD	3ms	138ms	valid
G	ST	3ms	142ms	falsifiable
H	FD	8ms	124ms	valid
H	ST	4ms	132ms	valid

Source: Author.

The memory usage was too low to be analyzed with conclusiveness (below 500kB for both checkers).

In this test, both methods once again accurately recognized the errors introduced. Both methods were able to solve the problem in less than 150ms for all properties, and thus it can be concluded that both methods can be applied to verification of SIS interlock logic without any timed constraints. While the NuSMV solver was once again faster in recognizing the errors, no conclusions can be drawn in terms of comparing the efficiency of both tools in a more general scope of timeless SIS logic, as the Kind2 consistently has solving times higher than 120ms, possibly due to a higher initialization overhead.

5 Conclusion

From the analysis of the test results, the following conclusions can be made in regards to the studied tools:

- The efficiency of the Bounded Model Checking methods need to be refined so that they may be applied to systems containing timers with higher values of time of activation. Possible ways to achieve this include detecting in which cases the complementary methods (k-induction, invariant generation, etc.) have a broader range of intervention, reducing overload on Bounded Model Checking in non-ideal cases. Another option is the development of techniques with explicit temporal constraints via SAT or SMT;
- The NuSMV tool showed good promise in its effectiveness for Model Checking Safety Instrumented Systems, and even though the manual construction of the SMV is an arduous repetitive task, the use of templates for semi-automatic of them facilitates the process of verification.

From this, the following steps could be executed with the goal of extending the understanding of the applicability of each of the tools for the verification of Safety Instrumented System logic:

- Implementing own set of solvers from scratch or low-level libraries, with the goal of evaluating the benefit of each auxiliary method more independently, especially surrounding Bounded Model Checking Techniques. The student began the development of said solvers, but due to lack of time in the internship a significant advance was not possible;
- Implementing a hybrid method with explicit time representation, such as the one employed by Fiacre/TINA, in which temporal constraints are represented via a set of equations. Even though it was more of an idea than a concrete goal, it was not possible to initiate the development of this tool.

Bibliography

AKERS, S. B. Binary decision diagrams. *IEEE Transactions on Computers*, C-27, n. 6, p. 509–516, 1978. Cited in page 32.

American Institute of Chemical Engineers. *Layer of Protection Analysis: Simplified Process Risk Analysis*. New York, USA: Center for Chemical Process Safety, 2001. Cited 2 times in pages 7 and 20.

BARRETT, C.; STUMP, A.; TINELLI, C. The SMT-LIB standard - version 2.0. *Proceedings of the 8th international workshop on satisfiability modulo theories*, Edinburgh, Scotland, v. 13, p. 14, 2010. Cited in page 32.

BIERE, A.; CIMATTI, A.; CLARKE, E. M.; STRICHMAN, O.; ZHU, Y. Bounded model checking. *Advances in Computers*, v. 58, p. 117–148, 2003. Cited 2 times in pages 7 and 32.

BIERE, A.; HEULE, M.; MAAREN, H. v.; WALSH, T. *Handbook of Satisfiability: Frontiers in Artificial Intelligence and Applications*. Amsterdam, NL: IOS Press, 2009. Cited in page 32.

CHAMPION, A.; MEBSOUT, A.; STICKSEL, C.; TINELLI, C. The kind 2 model checker. *Chaudhuri S., Farzan A. (eds) Computer Aided Verification. CAV 2016. Lecture Notes in Computer Science*, v. 9780, p. 510–517, 2016. Springer, Cham. Cited 3 times in pages 9, 32, and 34.

CIMATTI, A. et al. NuSMV 2: An opensource tool for symbolic model checking. *Brinksma, Ed , Larsen, K. G. (eds) Computer Aided Verification. Lecture Notes in Computer Science*, Springer Berlin Heidelberg, Berlin, Germany, v. 2404, p. 359–364, 2002. Cited 2 times in pages 9 and 33.

CIMATTI, A.; GRIGGIO, A.; MOVER, S.; TONETTA, S. Ic3 modulo theories via implicit predicate abstraction. 2013. Cited in page 32.

de MOURA, L.; BJØRNER, N. Z3: An efficient SMT solver. *Ramakrishnan, C. R., Rehof, Jakob (eds) Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science*, v. 4963, p. 337–340, 2008. Springer Berlin Heidelberg. Cited 2 times in pages 32 and 34.

FERNANDEZ, B. et al. Cause-and-effect matrix specifications for safety critical systems at CERN. *ICALEPCS: International Conference on Accelerator and Large Experimental Physics Control Systems*, oct 2019. Cited in page 25.

IEC61131. *IEC 61131: Programmable controllers - Part 3: Programming languages*. Geneva, CH, 2013. Cited in page 25.

IEC61508. *IEC61508: Functional safety of electrical/electronic/programmable electronic safety-related systems*. Geneva, CH, 2010. Cited 5 times in pages 7, 8, 20, 21, and 22.

IEC61511. *IEC61511: Functional safety - Safety instrumented systems for the process industry sector*. Geneva, CH, 2018. Cited 2 times in pages 7 and 20.

IEC62881. *IEC62881: Cause and effect matrix*. Geneva, CH, 2018. Cited in page 25.

LAZARO, F. da S. Metodologia para desenvolvimento de sistemas de controle e monitoração de navios assistidos por model checking. 2018. Cited 2 times in pages 8 and 25.

NUSMV. *NuSMV 2.6 User Manual*. 2010. Available at <http://nusmv.fbk.eu/NuSMV/userman/v26/nusmv.pdf>. Accessed on November 14th 2019. Cited in page 33.

SHEERAN, M.; SINGH, S.; STåLMARCK, G. Checking safety properties using induction and a sat-solver. *Johnson, S.D., Hunt Jr., W.A. (eds.) FMCAD. Lecture Notes on Computer Science*, v. 1954, p. 108–125, 2000. Cited in page 32.

VEIGA, H. W.; de QUEIROZ, M. H.; FARINES, J.-M.; de LIMA, M. L. Automatic conformance testing of safety instrumented systems for offshore oil platforms. *Petrucci L., Seceleanu C., Cavalcanti A. (eds) Critical Systems: Formal Methods and Automated Verification. AVoCS 2017, FMICS 2017. Lecture Notes in Computer Science*, v. 10471, p. 51–65, aug 2017. Springer, Cham. Cited in page 25.

APPENDIX A – Template for batch execution of test case 3x2 with NuSMV

```
-- ------------------------------------------
-- Template for batch execution of test case 3x2
-- Developed by Mateus Giovani Ewert Bonet
-- LAAS, Toulouse, 2019
-- ------------------------------------------

MODULE scancycle
VAR
    step : 0..5;
    iA : boolean;
    iB : boolean;
    iC : boolean;
    oD : boolean;
    oE : boolean;
    IN1 : boolean;
    IN2 : boolean;
    mB : boolean;
    mC : boolean;
    tB : TON(IN1, 300/SCANTIME_MS, Scan_CLK);
    tC : TON(IN2, {}/SCANTIME_MS, Scan_CLK);  -- Implementation Error
                                              -- Failure on Demand of oD

DEFINE
   Scan_CLK := step = 1;
   SCANTIME_MS := 150;        -- Representation of PLC cycle duration

ASSIGN
   init(step) := 0;
   init(iA) := FALSE;
   init(iB) := FALSE;
   init(iC) := FALSE;
   init(oD) := FALSE;
   init(oE) := FALSE;
```

```
    init(mB) := FALSE;
    init(mC) := FALSE;
    init(IN1) := FALSE;
    init(IN2) := FALSE;

     -- Synchronous cyclical operation
     next(step) := ((step + 1) mod 6);

-- ----------------------------------
-- Scan Cyle (Update Memories)
-- ----------------------------------
     next(IN1) :=
         case
             (step = 0) : iB;
             TRUE     : IN1;
         esac;
         next(IN2) :=
         case
             (step = 0) : iC;
             TRUE     : IN2;
         esac;
     next(mB) :=
         case
             step = 2 : tB.Q;
             TRUE     : mB;
         esac;
     next(mC) :=
         case
             step = 2 : tC.Q;
             TRUE     : mC;
         esac;

-- ----------------------------------
-- Scan Cyle (Calculate outputs)
-- ----------------------------------
     next(oD) :=
         case
             step = 3 : (iA & mB) | mC;
             TRUE     : oD;
```

```
        esac;
    next(oE) :=
        case
            step = 3 : (iA | !iC) | iB;        -- Implementation Error
            TRUE     : oE;                     -- Spurious Trip of oE
        esac;

-- ------------------------------------
-- Scan cycle (Read Inputs)
-- ------------------------------------
    next(iA) :=
        case
            step = 5 : {{FALSE, TRUE}};
            TRUE     : iA;
        esac;
    next(iB) :=
        case
            step = 5 : {{FALSE, TRUE}};
            TRUE     : iB;
        esac;
    next(iC) :=
        case
            step = 5 : {{FALSE, TRUE}};
            TRUE     : iC;
        esac;

-- ---------------------------------------------------------------------
-- Timer Template
-- ---------------------------------------------------------------------
MODULE TON(TIN, PT, CLK)
    VAR
        ET : 0..PT;
        state : {{idle, running, elapsed}};
        CLKp : boolean;
    ASSIGN
        init(state) := idle;
        init(ET) := 0;
        next(state) :=
            case
```

```
                !TIN & CLK & !CLKp : idle;
                state = idle & TIN & CLK & !CLKp : running;
                state = running & (ET >= PT) & CLK & !CLKp : elapsed;
                TRUE : state;
            esac;
        next(ET) :=
            case
                state = idle & CLK & !CLKp : 0;
                state = running & CLK & ! CLKp : (ET < PT ? ET + 1 : ET);
                TRUE : ET;
            esac;
        next(CLKp) := CLK;
    DEFINE
        Q := state = elapsed;

-- ---------------------------------------------------------------------
-- Properties Definition
-- ---------------------------------------------------------------------

MODULE main
    VAR
        plc : scancycle;
        D_TON_B : TON(plc.iB, 300/plc.SCANTIME_MS, plc.Scan_CLK);
        D_TON_C : TON(plc.iC, {}/plc.SCANTIME_MS, plc.Scan_CLK);
    DEFINE
        write := plc.step = 4;
        cause_D := (plc.iA & D_TON_B.Q) | D_TON_C.Q;
        FD_D := write & cause_D & !plc.oD;
        ST_D := write & !cause_D & plc.oD;

    DEFINE
        cause_E := (plc.iA & !plc.iC) | plc.iB;
        FD_E := write & cause_E & !plc.oE;
        ST_E := write & !cause_E & plc.oE;

-- Property to be evaluated
    SPEC AG !{}
```

APPENDIX B – Template for batch execution of test case 3x2 with Kind2

```
-- ------------------------------------------
-- Template for batch execution of test case 3x2
-- Developed by Mateus Giovani Ewert Bonet
-- LAAS, Toulouse, 2019
-- ------------------------------------------

-- ------------------------------------------------------------
-- PLC Scancycle (logic implementation)
-- ------------------------------------------------------------

const SCANTIME_MS = 150;
const TIMER_B_LIMIT_SPEC = 300;
const TIMER_B_LIMIT = 300;
const TIMER_C_LIMIT_SPEC = {0};
const TIMER_C_LIMIT = {1};

node scancycle(iA, iB, iC : bool) returns (oD, oE: bool);
var tB, tC: bool;
let
    tB =  TON(iB, TIMER_B_LIMIT/SCANTIME_MS);
    tC =  TON(iC, TIMER_C_LIMIT/SCANTIME_MS);     -- Timer Mistake
    oD = (iA and tB) or tC;
    oE = (iA or not iC) or iB;                    -- Logic Mistake
tel

-- ------------------------------------------------------------
-- Timer template
-- ------------------------------------------------------------

node TON(TIN : bool; PT : int;) returns (Q : bool);
var ET, pET : int;
let
    pET = (0 -> pre ET);
```

```
    ET = if TIN then
            if (pET < PT) then (pET + 1) else pET
         else 0;
    Q = ET >= PT;
tel

-- ------------------------------------------------------------
-- Temporal properties (specification)
-- ------------------------------------------------------------

node ReqPLC(iA, iB, iC : bool) returns (FD_D, ST_D, FD_E, ST_E: bool );
var tB_req, tC_req, causeD, causeE, oD, oE: bool;
let
    tB_req = TON(iB, TIMER_B_LIMIT_SPEC/SCANTIME_MS);
    tC_req = TON(iC, TIMER_C_LIMIT_SPEC/SCANTIME_MS);
    (oD, oE) = scancycle(iA, iB, iC);
    causeD = (iA and tB_req) or tC_req;
    causeE = (iA and not iC) or iB;
    FD_D = causeD => oD;
    ST_D = oD => causeD;
    FD_E = causeE => oE;
    ST_E = oE => causeE;

--%MAIN;
--%PROPERTY {2};
tel
```