EFFICIENT TECHNIQUES AND TOOLS FOR SOFTWARE TESTING BASED ON TRACES AND COVERAGE ANALYSIS

Doctoral Dissertation by
Faustin AHISHAKIYE

Thesis submitted for
the degree of Philosophiae Doctor (PhD)
in
Computer Science:
Software Engineering, Sensor Networks and Engineering Computing

Department of Computer Science,
Electrical Engineering and Mathematical Sciences

Faculty of Engineering and Science

Western Norway University of Applied Sciences

June 1, 2022

Series of dissertation submitted to
the Faculty of Engineering and Science,
Western Norway University of Applied Sciences.

ISBN: 978-82-93677-91-8
ISSN: 2535-8146

Author: Faustin AHISHAKIYE
Title: Efficient Techniques and Tools for Software Testing based on Traces and Coverage Analysis

Printed production:
Molvik Grafisk / Western Norway University of Applied Science

Bergen, Norway, 2022

TO MY WIFE, ALICE TUYISENGE,
TO MY SON, JAYDEN AHISON UHIRIWE,
TO MY FAMILY,

for endlessly loving, caring, supporting, and encouraging me.

PREFACE

The author of this thesis has been employed as a Ph.D. research fellow at the *Department of Computer Science, Electrical Engineering and Mathematical Science* in the *Faculty of Engineering and Science* at *Western Norway University of Applied Sciences, Norway (HVL)*.

The author has been enrolled into the PhD programme in the software engineering group of the Computer Science: Software Engineering, Sensor Networks and Engineering Computing with specialization in Software Engineering under supervision of Prof. Volker Stolz and Prof. Lars Michael Kristensen.

The research presented in this thesis has been accomplished in cooperation with the Institute for Software Engineering and Programming Languages at University of Lübeck, Germany through the European Horizon 2020 project COEMS[1].

This thesis is organized in two parts. Part I presents an overview and introduction to the research field of efficient techniques and tools for software testing based on traces and coverage analysis. It gives details on Modified Condition Decision Coverage (MC/DC) and its application on design level models. We describe our approaches for data races detection and test cases generation satisfying MC/DC. In addition, it provides a discussion of the research methodology used, a summary of the results obtained, the state-of-the-art in the form of related work, and a discussion of the research contributions and possible extensions of our work in the form of future work. Part II consists of a collection of published and peer-reviewed research articles and submitted papers.

Paper A Faustin Ahishakiye, Svetlana Jakšić, Felix Dino Lange, Volker Stolz, Malte Schmitz, Daniel Thoma: Non-intrusive MC/DC measurement based on traces. In Proceedings of the 13th International Symposium on Theoretical Aspects of Software Engineering (TASE 2019), Guilin, China, pages 86-92, Jul. 2019, https://doi.org/10.1109/TASE.2019.00-15.

Paper B Faustin Ahishakiye, José Ignacio Requeno Jarabo, Violet Ka I Pun, Volker Stolz: Hardware-Assisted Online Data Race Detection, In Proceedings of the Formal Methods in Outer Space. LNCS, volume 13065, Springer, Cham, Rhodes, Greece, October 2021, https://doi.org/10.1007/978-3-030-87348-6_6.

Paper C Faustin Ahishakiye, José Ignacio Requeno Jarabo, Lars Michael Kristensen, Volker Stolz: MC/DC Test Cases Generation based on BDDs, In Proceedings of the Symposium on Dependable Software Engineering Theories, Tools and Applications, **(SETTA 2021)**, LNCS, volume 13071, Beijing, China, November 2021, https://doi.org/10.1007/978-3-030-91265-9_10

Paper D Faustin Ahishakiye, José Ignacio Requeno Jarabo, Volker Stolz, Lars Michael Kristensen: Coverage Visualization and Analysis of Net Inscriptions in Coloured Petri Net Models. Journal of Innovations in Systems and Software Engineering (ISSE), March, 2022 (Submitted, under review).

[1]https://www.coems.eu/

ACKNOWLEDGMENTS

The completion of this research work and the whole doctoral life have been supported by several individuals in various ways. Therefore, I would like to give my wholehearted thanks to all.

First and foremost, I would like to express my heartfelt gratitude to my supervisors Prof. **Volker Stolz** and Prof. **Lars Michael Kristensen** for directing and supervising this research. I am very grateful for their expert guidance, encouragement, visionary ideas, and responsibility towards me as a student. I appreciate their priceless supports, precious comments, constructive discussions, constant patience and great kindness.

I am very grateful to my main supervisor Prof. Volker for taking care of me from day one in Bergen and guiding my research work since the writing of my research proposal to the completion of this thesis. I cannot find words to thank him enough for his guidance and support during the whole PhD period. Without his excellent guidance, pushing, and encouragement when I needed to improve my research work, this dissertation would not have been possible. He helped me to learn the basics of software testing, runtime verification, and programming skills. I appreciate that whenever I got stuck during my research activities, I was welcomed into his office to ask questions and I always got satisfactory support. I want to thank Volker also for introducing me to the COEMS project, BECHONG project and AURORA project. I got the opportunity to participate and to present my work in those projects and I also learned a lot from other researchers. I thank him for introducing me to the world of researchers through various conferences and meetings that I attended. I appreciate that in most of the trips, he made sure that I have all the necessities to travel and from all these trips, I got strong connections that were valuable to my research work. In addition, we had great discussions on different scientific work we published and I learned a lot from him on how to disseminate my research work. I am grateful for his useful inputs, format, careful reading and constructive comments on this dissertation. Moreover, I had chance to get feedback from him on the courses that I was teaching and the courses that were apart of my course work. For the social life, I am very thankful for his kindness in different cheering events we had together and I thank him for various personal advises he gave me and wonderful caring to my family. I learned a lot of lessons from him that will always guide me in the future.

I would like to give my special thanks to my co-supervisor, Prof. Lars whose guidance has been very valuable to the success of this dissertation. He introduced the knowledge of modeling with Coloured Petri Nets (CPNs) and I got an opportunity to apply the coverage analysis to CPNs models which resulted in good publications. I appreciate all the constructive reviews and suggestions he gave me that helped me to learn and understand different concepts in my research work. I thank him for being there at every phase of my doctoral study, for his encouragements and careful reading as well as precious comments on this dissertation. I appreciate our discussions in different meetings and his various advises that helped to improve my work. I enjoyed our conversations during lunch and coffee breaks or at HVL events and parties. Collaborating with him was an enriching opportunity.

My sincere gratitude goes to Dr. José Ignacio Requeno Jarabo for his full support and guidance. I consider him as my "unofficial co-supervisor" since his arrival in Bergen and to the completion of this thesis. I learned a lot from him especially when we worked together on coverage analysis for CPNs models, test cases generation satisfying MC/DC and online data races detection. It was a blessing to know him and an opportunity to me to improve my research work. I thank him for the productive meetings and discussions about my project and other scientific work. I always got help from him whenever I needed a support. From him, I learned how to overcome different challenges and how to implement different solutions practically. In addition, I thank him for the proof of reading and useful comments on this dissertation. I appreciate the time we spent together in Bergen, and I enjoyed different lunches, dinners and parties we had together. I am really thankful for his flexibility and kindness as well as his encouragements to me. I always appreciate that he was always caring on how my family is doing, and it was our pleasure to host him to our family.

Special thanks to Dr. Svetlana Jakšić for her guidance and discussions on my research work. Before she left HVL, she was my co-supervisor and she helped me to have a good start and to shape my research proposal. We had several productive meetings and discussions that enriched my understanding on software testing and her suggestions were always brilliant. I am grateful for introducing me to data races detection based on lock instrumentation and TeSSla specification language. Thanks a lot for the work we carried out together especially for our published papers on MC/DC measurements and coverage analysis of CPN models. I cannot forget that we always traveled together when attending COEMS project meetings and I am very thankful to her feedback and comments on my presentations. I also want to mention that Svetlana and Volker were always by my side both academically and socially and I will always remember your kindness! I am grateful to Dr. Dan Li who also provided useful opinions, advice, and encouragements on my research work. I still remember the lovely barbecue we had the four of us together, it was a warm welcome and I enjoyed the time with you.

I want also to thank Associate Professor Violet Ka I Pun for her support and encouragements for my research work. She taught me many basics of Linux system and we collaborated on different projects including the COEMS project, hardware assisted online data race detection, and we taught together the Database and Unix System Administration course. I learned a lot from different discussions, meetings and conferences we had together. Again, I want to thank Violet and Volker together for inviting me and my family to their home several times. We had great times together and thanks a lot for that friendly welcome.

I also want to give my special thanks to Felix Dino Lange for our strong collaboration on MC/DC measurement based on traces through the COEMS project. We had good discussions during my research trip to Lübeck and his research stay in Bergen. For our non-intrusive MC/DC measurement paper, Dino contributed in various ways both for the implementation and paper writing. I equally want to thank Dr. Malte Schmitz and Dr. Daniel Thoma whose contribution has been very valuable to the success of MC/DC measurement paper.

I would like to thank Prof. Martin Leuker and COEMS project team members for introducing to me the knowledge of continuous observation of embedded systems and run time verification in general. I gained and learned a lot from different experienced researchers from the COEMS project. My special thanks to Associate Professor Guillaume Hiet for their warm welcome to Rennes. We had great discussions and nice presentations during our research trip to Rennes.

In regards to the teaching duties, I express my gratitude to Associate Prof. Bjarte Kileng whom we taught together the course of "Database and Unix System Administration". I equally thank Maksim Melnik Storetvedt and Haakon André Reme-Ness who as well participated in the same course, for their immense contribution and knowledge sharing. In addition, I thank Bjarte for helping me to learn and understand the distributed systems and big data management as well as providing to me a working environment with a small cluster to improve my knowledge. My gratitude goes to Prof. Yngve Lamo, Dr. Slobodan Drazic and Qasim Ali whom we taught together the course of Discrete Mathematics. It was wonderful to work with them as a team.

I gratefully acknowledge the Western Norway University of Applied Sciences (HVL) and COEMS project as funding sources that made my Ph.D. work possible. I was honored to have a research fellow position at HVL and I am thankful to everyone who made it happen. I am especially thankful to the staffs at the department of Computer Science, Electrical Engineering and Mathematical Sciences for their friendly support. In particular, I want to first mention Kristin Fanebust Hetland, Håvard Helstrup, and Pål Ellingsen. They always kindly provided me with help and made sure that I was doing well if I met any problem at work. I was also very welcomed by them whenever I dropped by their offices. Thanks a lot for taking care of all administrative concerns. I am truly thankful for the time and effort they used to support my work in the department. Here, I again want to thank Volker together with Lars who both helped and contributed to the Norwegian abstract of the thesis. Special thanks to the PhD programme committee that I was a member of for two years, as a representative of other PhD students. I am proud that I have been apart of the team where I learned a lot from my seniors and thanks a lot for the work we carried out together. I thank Prof. Antoine Tambue for Various advises he provided to me.

I would like to thank Prof. Yngve Lamo and Ass. Prof. Joao Lourenco who were my mid-term evaluators for giving me many ideas and suggestions about my research directions that helped me to solve technical problems. I would also like to thank Prof. Adrian Rutle who gave me the bed for my first day in Bergen and who was always nice and friendly to me.

I am grateful to Reidun Johanne Benes Lyslo Lohiniva for helping my family about the visa application process and other related issues. I also thank Kristian August Mowinckel and the IT staffs for providing support to whenever it was needed. I am thankful to the HVL library for useful resources both in terms of books and seminars that helped me to carry out my research activities. Special thanks to Dr. Rui Wang and Dr. Alejandro Rodriguez Tena for their discussion on CPN models and providing their models to for coverage analysis.

I also want to give my thanks to all other former and new Ph.D. fellows colleagues for both the great working and social environment. To Dr. Fernando, Ass. Prof. Rabbi, Espen, Simon and Andreas, thanks a lot for helping me a lot about different information as my seniors at the beginning of my work at HVL. To Alejandro, Justus, Ole, it was good to travel together in different conferences. To Frikk it was always fun to try some of your electronics circuits and to chat with you about Norwegian culture, and all other interesting things. To Suresh, Angela, Patrick, Amin, Samaneh, Rizwan, Håkon, Fatemeh, Salah, Maksim, Anton, Mahmood, Michele, Tim and Gerard, I appreciate the time we spent together as colleagues and friends. To Diana, Eunice, Theogene, Esther, John's family and Joseph's family, it was always great to have some dinners and discussions together. To Qasim, thank you for inviting my family to your home for dinner; that was great with nice food. Also, to Daudel, Tsafag, Gizem, Haja, Remco, Rahul, Mojtaba, Nguyen, Keila, Lena, Laurenz, Vegard, Hunter, Johannes, Daniel, and all other Ph.D. researchers, thanks to you, there were countless funny and interesting topics we have discussed during lunch and coffee breaks or at HVL events and parties.

In addition, I want to express my deep gratitude to Rwandan community and our friends in Bergen who always were close to my family and we appreciate all the time we spent together. I would like to give special thanks to the Embassy of Rwanda in Sweden for all support and good service provided to us whenever it was needed. I would like to thank all Seventh Day Adventist (SDA) church members in Bergen for their hospitality and love. I enjoyed the good time we had with you. Special thanks to all my fiends who always gave me their encouragement in various ways.

My gratitude goes to the fellow colleagues from my former University of Rwanda especially from the School of Engineering, Department of Electrical and Electronics Engineering (EEE), your support will always be remembered. Special thanks to Dr. Kizito Nkurikiyeyezu for his encouragement and good advises to me.

I express my eternal gratitude to my mother who raised me with love, my sisters and brother far away for their blessings, love and support. My gratitude goes to my parents in law, sisters in law, and brother in law who were always there whenever I needed them. Special thanks to Epaphrodite NIZEYIMANA for helping me in different back home activities that I couldn't manage to do remotely. I thank him for being there whenever I needed him and for visiting me in Norway. It was such a huge joy and pleasure to host him.

I would also like to thank my son Jayden Ahison UHIRIWE for his love and happiness he brought to our family. Whatever the challenges, stresses and difficult moment we encountered, with him I was feeling relieved and it was always fun to be out with him. He is such lovely and cute boy and I love him.

Last and most of all, I am grateful to my patient wife Alice Tuyisenge who always loved, supported, motivated, and encouraged me in this endeavor. Your full care is countless and priceless. It would require a separate book to write about your faithful support during all stages of this Ph.D. I am proud that I have chose you as my wife!

ABSTRACT

To ensure ultra-high dependability and ultra-low defect rates, certification standards such as DO-178C requires safety-critical software with the highest safety level (Level A) in avionics systems to conform to the modified condition decision coverage (MC/DC) criterion. MC/DC is a strong coverage criterion that subsumes existing coverage criteria and it requires a small number of test inputs compared to the combinatorially exhaustive multiple condition coverage (MCC). MC/DC has also proven to reveal many program defects. However, both MC/DC measurement and generating test cases satisfying MC/DC remain a challenging task. In addition, related properties such as data races detection can be monitored using some methods used to check MC/DC, as good concurrency coverage increases a likelihood of catching concurrent-related bugs. To address the above challenges, existing strategies rely on intrusive instrumentation which is not recommended for safety critical software since it consumes valuable resources and can alter the behaviour of the system under test (SUT) if it remains in the released code.

To overcome the above challenges, this thesis introduces novel paradigms and tools for software testing based on traces and coverage analysis. Our aim is to analyse the MC/DC without instrumentation and to monitor data races with a lightweight instrumentation. In addition, we explore the applicability of MC/DC criterion on the design level models. Furthermore, we investigate new techniques for test cases generation satisfying MC/DC with the aim to increase the coverage.

The scientific contribution of this thesis is fourfold:

First, we propose an approach for measuring MC/DC without instrumentation. This has resulted in a tooling for MC/DC measurement and analysis based on the trace of an executing program. A static analysis is used to find conditional jumps in object code that correspond to conditions in the source code. With that information the assignments of the conditions during the execution of the code can be reconstructed by analyzing the trace. MC/DC is then evaluated and the covered/uncovered conditionals in the program can be identified. This approach is evaluated on C programs.

Secondly, we provide a non-intrusive tooling for data races detection using the continuous observation of embedded multicore systems (COEMS) technology through continuous online monitoring with lightweight instrumentation on a novel FPGA-based external platform for embedded multicore systems. It is used in combination with formal specifications in the high-level temporal stream-based specification language (TeSSLa), in which we encode a lockset-based algorithm to indicate potential race conditions. We show how to instantiate a TeSSLa template that is based on the Eraser algorithm, and present a corresponding light-weight instrumentation mechanism that emits the required observations to the FPGA with low overhead.

Thirdly, we investigated the applicability of MC/DC criterion on design level models, where specifically, we conducted a coverage analysis to Coloured Petri Nets (CPNs) models. We implement a library for CPN Tools and a post-processing tool for MC/DC coverage analysis of net inscriptions on a set of model executions and evaluate our approach on eleven larger publicly available CPN models.

In the fourth contribution, we propose a new and alternative strategy for test case generation satisfying MC/DC. We have implemented an algorithm for MC/DC test cases based on binary decision diagrams (BDDs) and evaluated on Traffic Alert and Collision Avoidance System (TCAS II) benchmarks. A performance evaluation with respect to the state-of-the art in the form of related work has been conducted.

SAMMENDRAG

For å sikre høy pålitelighet og lav feilrate krever sertifiseringsstandarder som DO-178C at sikkerhetskritisk programvare som oppfyller det høyeste sikkerhetsnivå (nivå A) innen flykontrollsystemer tilfredsstiller det modifiserte betingelses-beslutningsdekning (MC/DC) kriterium. MC/DC er et sterkt dekningskriterium som inkluderer eksisterende dekningskriterier og krever et lite antall test input sammenlignet med kombinatoriske utfyllende fler-betingelses dekning (MCC).

MC/DC har vist seg å kunne detektere mange programvare feil. Samtidig er MC/DC målinger og generering av testtilfeller som oppfyller MC/DC fortsatt en utfordring. I tillegg kan relaterte egenskaper som detektering av data-inkonsistens overvåkes ved å bruke metoder for å sjekke MC/DC. Årsaken til dette er at god samtidighetsdekning øker sannsynligheten for å oppfange feil i programvare relatert til samtidighet.

For å adressere disse utfordringer bygger eksisterende strategier på instrumentering som ikke er anbefalt for sikkerhetskritisk programvare siden dette forbruker ressurser og kan endre oppførselen av systemer som er under test (SUT) når dette forblir i koden som settes i produksjon. For å eliminere disse utfordringer introduserer denne avhandlingen nye paradigmer og verktøy for programvaretesting basert på spor og dekningsanalyse. Målet vårt er å analysere MC/DC uten instrumentering og overvåke data-inkonsistens med lettvekt-instrumentering. I tillegg undersøker vi bruken av MC/DC kriterier på design-nivå modeller. Videre ser vi på nye teknikker for test-tilfeller generering som oppfyller MC/DC med mål om å øke dekning.

Det vitenskapelig bidra av denne avhandling er innen fire områder:

Først foreslår vi en tilnærming for å måle MC/DC uten instrumentering. Dette har resultert i verktøy for MC/DC måling og analyse basert på spor fra et program som kjøres. Statisk analyse brukes til å identifisere betingede-hop i objekt-koden som svarer til betingelser i kildekoden. Basert på denne informasjon kan tildelinger til betingelser under utførselen av koden rekonstrueres ved å analysere sporet. MC/DC evalueres og dekkede/ikke-dekkede betingelser i programmet kan identifiseres. Tilnærmingen evalueres på C programmer.

Dernest utvikler vi et ikke-intrusivt verktøy for data-inkonsistens deteksjon ved å bruke teknologi for kontinuerlig observasjon av innebygde fler-kjerne system (COEMS). Dette realiseres via kontinuerlig overvåking med lettvekts-instrumentering på en ny FGPA-basert ekstern plattform for innbygde fler-kjerne system. Dette brukes i kombinasjon med formell spesifikasjon i det høy-nivå strøm-basert temporal spesifikasjonsspråket TeSSLa, der vi innkoder en låsemengde-baserte algoritme for å indikere potensielle data-inkonsistenser. Vi viser hvordan TesSSLa maler kan brukes basert på Eraser-algoritmen, og presenterer en korresponderende mekaniske for lettvekts-instrumenterings som gir de observasjoner som kreves til en FPGA med liten reduksjon i ytelse.

Som det tredje bidrag undersøker vi bruken av MC/DC på design-nivå modeller, der vi spesifikt utfører dekningsanalyse på Fargede Petri Net (CPN) modeller. Vi har implementert et bibliotek for CPN verktøyet og et etter-prosesseringsverktøy for

MC/DC dekningsanalyse av net-inskripsjoner på en mengde av modellutførsler og evaluerer vår tilnærming på elleve større offentlig tilgjengelig CPN modeller.

Som det fjerde bidrag forslår vi en ny og alternativ strategi for generering av testtilfeller som oppfyller MC/DC. Vi har implementert en algoritme for MC/DC testfilfelle-generering basert på binære beslutningsdiagrammer (BDDs) og utført evaluering på en samling av trafikale alarm og kollisjonssystemer. En ytelsesevaluering med hensyn til relatert arbeid har vært gjennomført.

Contents

Part I

OVERVIEW

Computers themselves, and
software yet to be developed,
will revolutionize the way we learn.

—Steve Jobs [30]

CHAPTER 1

INTRODUCTION

Today's life is dependent on software based systems in multiple domains such as medicine, aviation, automation, communication and other safety critical systems. Typical examples have shown that software failures in most critical systems have led to the loss of lives and tremendous damage. A therapy planning software misinterpreted the holes drawn by doctors for specifying the placement of metal shields to protect healthy tissue from radiation and eight patients died and 20 received overdoses [119]. The Boeing Maneuvering Characteristics Augmentation System (MCAS) has proven susceptible to erroneous activation, as in the cases of Lion Air Flight 610 and Ethiopian Airlines Flight 302 and this led to not survivable accidents [65]. Thus, it is important to ensure that software validation, software verification and software testing are well implemented as an integral part of the software development life-cycle. Software *validation* aims at building the right product with respect to the user requirements, whereas software *verification* checks whether one is building the product right with respect to the specifications. Software *testing* checks the actual software system rather than a model, on sampling executions according to some coverage criteria. Coverage defines the extent to which a given verification or/and testing activity has satisfied its objectives.

During the software testing process, there are two types of analysis: static analysis and dynamic analysis [106]. Static analysis checks a program without executing it whereas dynamic analysis executes the program and makes analysis during the execution or after run time. These techniques yield good performance for sequential programs with the introduction of testing criteria and the implementation of supporting tools. Concurrent programs are attracting attention due to their performance in implementing parallel executions. However, the challenges and complexity increase when testing concurrent programs due to features such as communication, synchronization, nondeterminism, and concurrency defects. During execution, computations in a concurrent program can involve a high number of interactions among processes and the number of possible execution paths in the program can be extremely large. Therefore, some of the coverage criteria defined for structured testing of sequential programs need to be extended with other properties to be applicable for concurrent program testing.

In order to produce fail-safe and low risk systems, certification standards, for example the DO-178C [124] in the domain of avionic software systems, are used by certification authorities, like the Federal Aviation Administration (FAA) and the

European Aviation Safety Agency (EASA), to approve safety-critical software and ensure that the software used in the systems follows certain software engineering standards. DO-178C requires that structural coverage analysis is performed during the verification process mainly as a completion criterion for the testing effort and to identify design faults as well as finding dead code. Safety-critical software with the highest safety level (Level A) in avionics systems is required to conform to the modified condition decision coverage (MC/DC) criteria [42]. For MC/DC, each condition in a decision has to show an independent effect on that decision's outcome by: (1) varying just that condition while holding fixed all other possible conditions, or (2) varying just that condition while holding fixed all other possible conditions that could affect the outcome. MC/DC has attracted much attention both in academia and industry due to its benefits. Unlike weaker coverage criteria, MC/DC is sensitive to the complexity of decisions, it requires a small number of test cases, it is sensitive to the program structure, and it is unique due to the independence effect for each condition.

As part of this thesis, we investigate efficient techniques and tools for software testing based on traces and coverage analysis. We use MC/DC as coverage criterion since it subsumes the existing coverage criteria and is recommended for safety critical software systems [58, 124].

First, we analyse MC/DC measurement non-intrusively based on object code with traceability to source code using modern processor-based tracing facilities [15, 38, 129]. Our motivation is to measure MC/DC based on object code without instrumentation and our approach is based on traces [2, 9] recorded using the Intel Processor Trace (IntelPT) [15, 129], a facility present on modern Intel CPUs.

Secondly, we investigate concurrent programs by analysing the data race detection using the continuous observation of embedded multicore systems (COEMS) infrastructure with a lightweight instrumentation. The COEMS infrastructure refers to a novel observer platform for online monitoring of multicore systems. It offers a non-intrusive way to gain insights of the system behaviour, which are crucial for detecting non-deterministic failures caused by, for example, accessing inconsistent data as a result of race conditions.

The COEMS project[1] developed a novel platform for online monitoring of multicore systems. It offers the possibility to observe the system's behaviour without affecting it. This insight enables the detection of non-deterministic failures which are caused for example by race conditions and access to inconsistent data. To observe system-on-chip (SoC), the COEMS platform uses the tracing capabilities that are available on many modern multicore processors. Such capabilities provide highly compressed tracing information over a separate tracing port. This information allows the COEMS system to reconstruct the sequence of instructions executed by the processor [125]. The instruction sequence- and data trace can then be analysed online using TeSSLa [100], and by a reconfigurable monitoring unit. To cope with the potentially massive amount of tracing data generated by the processors, the COEMS system is implemented in hardware using an FPGA-based event processing system.

Thirdly, we explore the applicability of our MC/DC measurement to coverage analysis on design level models which combine both the modeling and the functional programming language. We implement our approach to coloured petri-nets (CPNs),

[1]https://www.coems.eu/

a language for modeling and validation of systems which involve concurrency, communication, and synchronisation.

Last but not the least, MC/DC measurement both on models and programs requires test inputs. Therefore, we investigate the MC/DC test cases generation where, we guide our tooling and models can be guided to improve coverage.

1.1 Software Testing

The demand for high-quality software is increasing both on the industrial level, users level, and academic level in order to cope with new advancements in technology and user needs. In addition, the software development industry is becoming more complex and testing has to be undertaken before the deployment of software as the truly effective means to assure the quality of a software system. Testing is defined by ANSI/IEEE 1059 standard [77] as the process of analyzing a software item to detect the differences between existing and required conditions (that is, bugs or defects) and to evaluate the features of the software artifact. Therefore, software testing is an important means of assessing software, a program or an application to ensure its quality [115]. Depending on the level of required software reliability and safety, software testing consumes considerable effort especially for systems that require higher levels of reliability as well as for safety critical systems. In this section, we provide background on different software testing concepts that are used as foundation for coverage measurement, test cases generation, data race detection and cache simulation.

1.1.1 *Testing levels*

Every stage of the software life cycle involves testing that is different in nature and has different objectives for each level of software development. Tests can be derived from requirements and specifications, design artifacts, or the source code. The following are the levels of testing with respect to software development activities [115]:

- *Unit Testing* is done at the lowest level. It tests the basic unit of software, which is the smallest testable piece of software, and is often called "unit", "module", or "component" interchangeably. It checks whether the individual modules of the source code are working properly. That is testing each and every unit of the application separately.

- *Integration Testing* is performed when two or more tested units are combined into a larger structure. It assesses software with respect to subsystem design. The test is often done on both the interfaces between the components and the larger structure being constructed, if its quality property cannot be assessed from its components.

- *System Testing* tends to affirm the end-to-end quality of the entire system. System test is often based on the functional/requirement specification of the system. Non-functional quality attributes, such as reliability, security, and maintainability, are also checked.

– *Acceptance Testing* assesses software with respect to customers'requirements to ensure that all objectives components are correctly included in a customer package. It is done when the completed system is handed over from the developers to the users. The purpose of acceptance testing is to provide confidence that the system is working as intended rather than to find errors. That is to determine whether or not a system satisfies its acceptance criteria.

The correspondence between testing levels and design phases of software development [13, 122] is shown in Fig. 1.1. The four phases of definition (requirements analysis, requirements specification, preliminary design and detailed design) correspond directly to the four levels of testing. The *requirements analysis* phase captures the user's needs. The *requirements specification* phase defines what components are required to implement the software system that meet the previously identified requirements. The *preliminary design* phase specifies the structure and behavior of subsystems, each of which is intended to satisfy some function in the overall architecture. The *detailed design* phase determines the structure and behavior of individual units. The *coding* phase produces the actual code of the software under development.

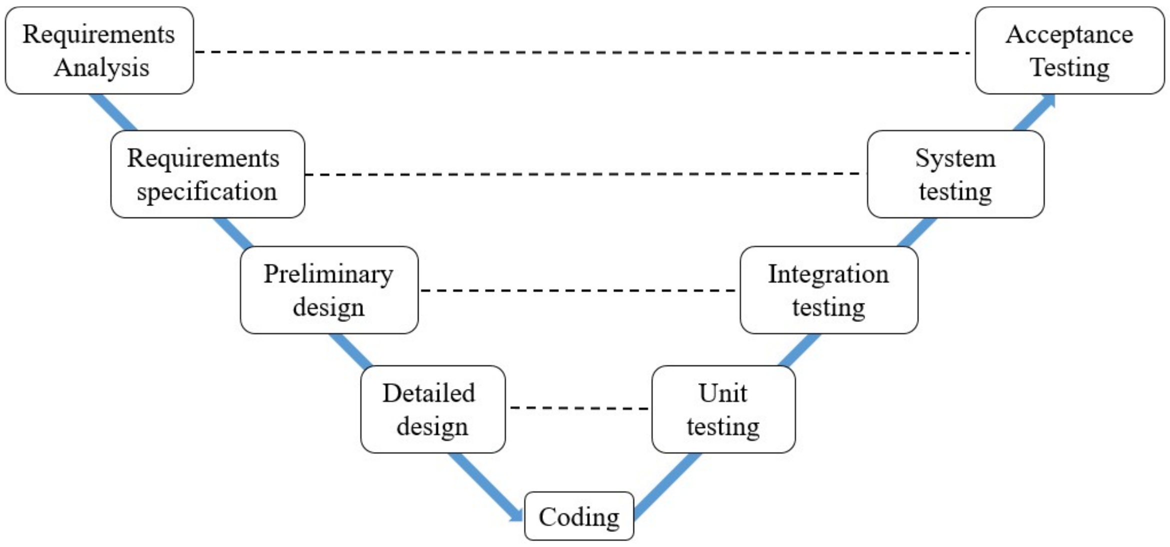


Fig. 1.1: Testing levels and design phases in "V-model" [122]

1.1.2 *Software testing terminologies, artifacts and limitations*

Below we discuss the related software testing terms and artifacts that will be used throughout this thesis as defined by the IEEE Standard Glossary of Software Engineering Terminology and the International Software Testing Qualifications Board (ISTQB) [77, 110]. The glossary explains an *error* as a mistake a human being can make, which produces a *defect (fault, bug)* in the program code, or in a document. If a defect in code is executed, the system may fail to do what it should do (or do something else it shouldn't), causing a *failure*. A *software failure* is an external, incorrect behaviour with respect to the requirements or other description of the expected behavior. Defects in software, systems or documents may result in failures, but not all defects do so. Another useful term in software testing is *debugging* which refers to the process in which developers identify, diagnose, and fix errors found. That is the process of finding a fault given a failure.

In addition, it is important to clarify the distinction between *validation* and *verification* [13, 77]. Validation is the process to check that the completed end product (software, a system or components) complies with the intended usage or the specified user requirements. Verification is the process of determining whether the products of each software life cycle phase comply with previous life cycle phase requirements and products, satisfy the specifications, and establish the proper basis for initiating the next life cycle phase activities.

During the testing activities, tests normally include more than just input values to the software system under test. There are actually multiple software artifacts. A *test case* specifies actual input values known as *test inputs* and expected results. The latter is know as a *test oracle*, which is the source to compare an expected result with the actual result of the system under test. A finite set of test case to be executed in a specific test run is called a *test-suite*. Test cases can be written manually or generated automatically. In order to limit the size of a test, *test selection criteria* are used to guide the *generation of test cases* or to select test cases. To perform testing, a test-suite can be executed using a *test script* which provide the sequence of instructions for the execution of a test. In structural testing of a software or a program it is necessary to define a *test condition* which refers to testable aspect of a component or system identified as a basis for testing. Another important artifact to note is the *concurrency testing* which intend to evaluate if a component or software system involving concurrency behaves as specified.

Although software testing is an important part of the whole software development life cycle, it possesses some limitations.The most important limitation is that software testing can only reveal the presence of failures, but not their absence [53]. The problem of finding all bugs in a program is undecidable and a test is called effective if an error is detected. Software testing does not provide prediction of the proper functioning of the product under all conditions, but it may prove to be helpful in delivering the information with respect to incorrect or improper functioning of the product under specific conditions. The earlier the prevention measures are taken, the better. Software testing does not help in finding root causes which resulted in injection of defects in the first place. Identifying the root causes of failures helps in preventing injection of defects for future purposes. Testing cannot detect any errors in requirements and ambiguous requirements lead the complete testing process to inadequate testing.

1.1.3 Static analysis and dynamic analysis

To evaluate the software product, there are three main techniques that are used: *static analysis, dynamic analysis* and *formal analysis* [77, 106]. The later is not discussed in detail inhere because it is not directly a part of software testing but rather is based on mathematical proofs. In the context of software testing, static analysis and dynamic analysis are used for software quality assurance activities. They differ on whether the actual execution of software under test is needed or not.

– *Static analysis* checks a software product without executing it. It focuses on determining or estimating software quality without reference to actual executions. For example, static analysis can be applied for code inspection, specification reviews, structure analysis, symbolic analysis, and model checking. Static analysis may be applied in all phases of development, and to all documented deliverables.

– *Dynamic analysis* executes the program and makes analysis during run time or after the execution to determine the validity of some of its attributes. It deals with specific methods for ascertaining and/or approximating software quality through actual executions. That is for example with real data and under real (or simulated) circumstances. Its techniques include synthesis of inputs, the use of structurally dictated testing procedures, and the automation of testing environment generation.

1.1.4 *Testing approaches*

There are three basic types of software testing approaches: *white-box, black-box,* and *gray-box* testing.

– *White-box testing* [13, 24, 122, 165]: also called glass-box, clear-box or structural testing. It is based on the application's internal code structure, and an internal perspective of the system as well as programming skills, are used. It analyzes the program structure to generate test cases. This testing is usually done at the unit level.

– *Black-box testing* [13, 122, 165]: also known as *behavioral, specification-based, input-output or functional testing.* It validates whether the functions specified in the system requirement specifications were correctly implemented or not. That is it evaluates the functionality of the software under test without looking at the internal code structure. Test cases are delivered from external descriptions of the software, including specifications, requirements, and design.

– *Grey-box testing* [165]: is the combination of both white-box and black-box testing. It extends the logical coverage criteria of white box method and intends to find possible paths from the design model which describes the expected behavior of an operation. Then it generates test cases which can satisfy the path conditions by black box method.

1.1.5 *Model based software testing*

From the software testing perspective, it is important to be concerned with the quantitative and qualitative analysis of the extent to which all the instances of the model have been covered. One part of this thesis investigates the applicability of MC/DC on design level model where we considered specifically the coverage analysis of net inscriptions in Coloured Petri Net models (CPNs).

Model based testing (MBT) [123] is a software testing technique which relies on the construction of an abstracted model of the system under test (SUT) and its environment such that the run time behavior of the SUT is checked against predictions made by a model. The system's behavior can be described in terms of input sequences, actions, conditions, output and flow of data from input to output. The basic idea of MBT is that test cases are generated automatically from a model and are executed manually or automatically on the SUT. In other words, it eliminates creating test cases manually.

The advantages of MBT are that it allows tests to be linked directly to the SUT requirements, which renders readability, understandability, and maintainability of tests easier. It helps ensure a repeatable and scientific basis for testing. Furthermore, MBT has been shown to provide good coverage of all the behaviors of the SUT [109] and to reduce the effort and cost for testing [108].

As shown in Fig. 1.2, the process of MBT proceeds as follows [109]:

Step 1: Based on the requirements or existing specification, a test model of the SUT is built. The model needs to reside at a certain level of abstraction and it encodes the intended behaviour of the SUT in a simpler way such that it can be modified, validated and maintained.

Step 2: Test selection criteria are defined and testing strategies and techniques may be used to guide the automatic test cases generation. These criteria can relate to a given functionality of the system (requirements coverage), to the structure of the model (structure coverage) or to stochastic characterizations.

Step 3: Test selection criteria are then transformed into test case specifications which formalize the notion of the criteria to be used and render them operational. These specifications describe the desired test cases such that an automatic test case generator is capable of deriving a test suite.

Step 4: A test suite is generated. It enumerates all the test cases satisfying the test case specifications. However, there may exist many test cases that satisfy the

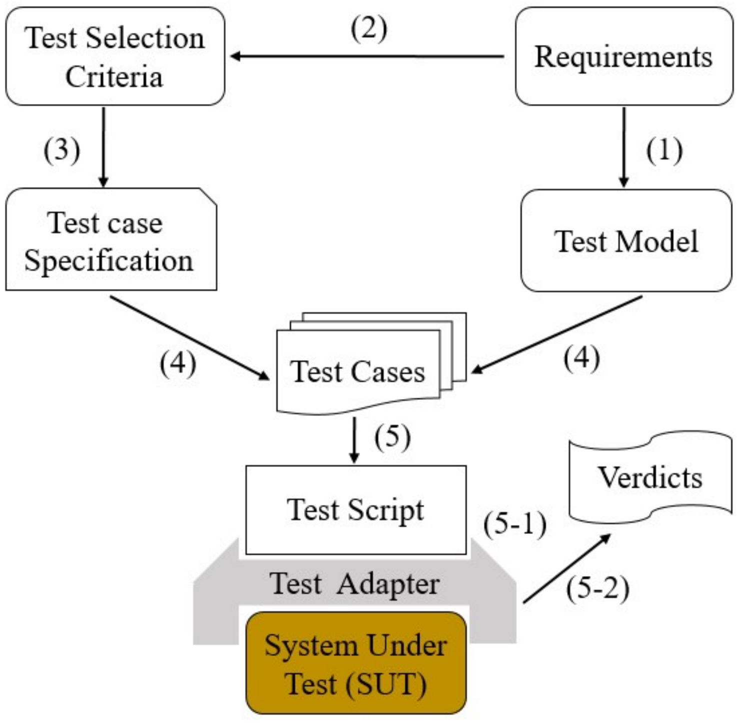


Fig. 1.2: The process of model based testing [108, 109]

specification and the test case generators that tend to pick a smaller set randomly that cover a large number of specifications.

Step 5: The generated test cases are then executed either manually or automatically. Running a test case includes two stages.

Step 5.1: The model and SUT reside at different levels of abstraction, and they must be bridged by an *adapter*. The adapter enables the concretisation of the test inputs into the SUT and collects the test outputs of the SUT.

Step 5.2: Finally the test adapter compares the test outputs against the expected outputs which results into the so called *verdict* which is also known as *test oracles* output.

A test adapter is a concept and not necessarily a separate software component. It may be integrated within a test script or can be implemented as a separate software component. Therefore, a test script is some executable code that executes a test case, abstracts the output of the SUT, and then builds the verdict.

1.1.6 *Software safety-levels*

Safety-critical software is defined as: "software whose use in a system can result in unacceptable risk [59]. It includes software whose operation or failure to operate can lead to a hazardous state, software intended to recover from hazardous states, and software intended to mitigate the severity of an accident. Software in the context of safety critical systems such as avionics are classified in safety-levels, depending on the severity of a defect [132].

– *Level A-Catastrophic*: represents the level where failure conditions result in multiple fatalities: death, permanent total disability, loss exceeding one million US dollars, severe environmental damage violating law or regulation. Level A software is the most rigorous and applies to software functionality that could cause or contribute to a catastrophic aircraft-level event.

– *Level B-Hazardous/ Severe major*: consists of software whose anomalous behaviour, would contribute to failure conditions resulting in a hazardous/severe-major event for the aircraft such as permanent partial disability, injuries or illness.

– *Level C-Major* : refers to failure conditions that would reduce the capability of the airplane or the ability of the crew to cope with adverse operating conditions to the extent that there would be a significant reduction in safety margins or functional capabilities.

– *Level D-Minor*: are failure conditions that would not significantly reduce airplane safety and involve crew actions that are within their capabilities. It may include a slight reduction in safety margins or functional capabilities, a slight increase in crew workload (such as routine flight plan changes), or some physical discomfort to passengers or cabin crew.

– *Level E-No effect*: are failures conditions that would have no effect on safety (that is, failure conditions that would not affect the operational capability of the airplane or increase crew workload).

During the testing of software of level E and level D, there are no required coverage criteria, rules or guidelines that have to be satisfied although they are not prohibited. For level A , level B and level C, different types of coverage criteria are strictly recommended. For example one item (coverage criteria, rules or guidelines) may be advisory for level C but mandatory for levels A and B.

1.1.7 *Testing coverage*

Test coverage is defined as a technique which determines whether our test cases are actually covering the application code and how much code is exercised when we run those test cases. *Coverage* is the measure of the degree to which testing activities have been exercised by a test suite expressed as a percentage. To measure coverage it is necessary to formulated coverage criterion to be satisfied. *Coverage criteria* define adequacy measures to qualify if a test objective is satisfied or reached when executing test cases on a system under test [51, 167]. Several testing coverage criteria have been proposed in the literature, including *statement coverage, branch coverage/decision coverage (BC/DC), path coverage, condition coverage (CC), condition/decision coverage (C/DC), multiple condition coverage (MCC)* and *modified condition/decision coverage (MC/DC)* [42, 58, 62, 94, 99, 124, 167]. We provide more details on these types of coverage in Section 2.1.

Certification authorities use the DO-178C as a certification standards to approve commercial software-based aerospace systems, which assures a certain quality of software. The software safety-level is assessed by examining the effects of a failure in the system as discussed in the previous subsection. Depending on the software safety-level different coverage criteria have to be fulfilled during software testing. Software level C (major effect) requires *statement coverage* and software level B (hazardous effect) requires *decision coverage*. Software level A (catastrophic effect) requires *modified condition/decision coverage* (MC/DC). Depending to which type of coverage, the coverage is measured in terms of ratio of covered entities (covered conditions, executed number of line, covered branch, ...) and total number of entities. The lower the percentage the higher the effort required to accomplish the testing activities.

1.2 Research Questions

This section presents our research questions and briefly provides an overview of how we address them in the PhD thesis.

RQ1: *Can we check modified condition decision coverage (MC/DC) without instrumentation?*

Usually MC/DC is measured by instrumenting the source code to gather information about the execution and assignment of conditions and decisions. Because it is required by the DO-178C that the structural coverage analysis has to be performed on the code that is released, the instrumentation has to be left inside the code. This is problematic because instrumentation consumes valuable resources. In case the instrumentation is removed from the released code, it is required to show that the behaviour of the program did not change. Therefore, MC/DC needs to be analyzed on object code level. Additionally, it is necessary to perform an object code to source code analysis to show that every line from the object code is traceable directly to the source code in order for compliance to DO-178C. If parts of the object code cannot be traced back to the source code, then additional analysis must be provided [38].

RQ1 aims at developing non-intrusive MC/DC measurement tooling based on traces. MC/DC needs to be checked on the object code level by analyzing program traces and investigate how conditionals in source code are reconstructed during the execution. This can be achieved by recording and decoding the trace of an executing program using Intel processor tracing (IntelPT) technology [15]. Then we can analyse the trace to see if the jumps corresponding to conditionals in the source code have been taken during the execution, and reconstruct how the conditions have been evaluated and the outcome of an entire decision. Finally, the above information is filled in a table and MC/DC is calculated as the ratio of covered conditions to the total number of conditions in each decision.

RQ2: *How can we monitor data races with low overhead instrumentation?*

Just as high coverage (at least in theory) provides good chances of catching bugs in sequential programs, high concurrency coverage should increase the probability of catching concurrent-related bugs. Our aim is to monitor data races which are problematic in multi-threaded programs. A data race occurs whenever two or more threads access same memory location concurrently without using any synchronization mechanism and at least one of the accesses is a write. However, dynamic data race detection techniques usually involve invasive instrumentation.

In relation to RQ1, high coverage measured via the traces analysis can show the uncovered part of the program. The question is whether the non-intrusive method used in RQ1 can also be used to check data races. In case avoiding the instrumentation is challenging, can data races be checked with lightweight instrumentation? We use continues observation of embedded systems (COEMS) technology as a solution through continuous online monitoring with low-impact instrumentation on FPGA-based external platform for embedded multicore systems. Based on RQ2, we expect to be able to identify potential data races (if they exist) from the program under observation.

RQ3: *Does MC/DC have applicability on the design level models?*

Coverage analysis is important for programs and models in relation to fault detection. Low coverage indicates that the software product has not been extensively tested [94, 132]. We investigate how the MC/DC criterion can be applied to design level models. Our intention in RQ3 is to conduct a coverage analysis of Net inscriptions in Coloured Petri Nets (CPNs) models. CPNs combine both Petri nets and a high-level programming language. *Petri nets* provide a formal foundation for modelling concurrency and synchronization constructs for defining colour sets and declaring variables concept of multisets and associated functions and operators. The *high-level programming language* in CPN model provides the primitives for modeling data manipulation by creating compact and parameterizeble models for concurrent systems using standard meta-language (SML). In a CPN model, the net inscriptions (e.g., arc expressions and guards) are specified using SML. The simulation and state space exploration for validating CPN models traditionally focuses on behavioural properties related to net structure, i.e., places and transitions. This means that the net inscriptions are only implicitly validated, and the extent to which these inscriptions have been covered is not made explicit.

Hence, we focus on how MC/DC (normally used for programming languages) can be applied for coverage analysis on the design level models such as CPN models. This provides an evidence that each condition contained in guards and arcs contributed as intended or not. In line with RQ1, it is possible to record the trace of net inscriptions evaluation in a CPN model and conduct a coverage analysis. In RQ3, we intend to collect MC/DC and branch coverage (BC) statistics in publicly available CPN models and conduct performance analysis on how well are SML conditions are covered in the existing CPN models under-test.

RQ4: *How can test cases satisfying MC/DC be efficiently generated?*

For analyzing coverage, it is required to have some test cases to be able to evaluate the behavior of a given program or model. One of the advantages of MC/DC [42] is that for a decision with n conditions, it may be satisfied with a low number of test cases: between a lower-bound of $n+1$ and upper-bound of $2n$ test cases, compared to MCC which requires 2^n test cases. MCC aims on trying all possible combinations which is exhaustive and requires tremendous resources [73], as well as becoming impracticable for a high number of conditions [74, 89]. While MC/DC is recommended for high level safety assurance, finding a test set equal or closer to $n+1$ with MC/DC assurance is a non-trivial task [67, 90]. Therefore, it is important to investigate new strategies for generating good test suites both in terms of number of test cases and coverage adequacy [66, 157] and with reasonable resources. In RQ4, we investigate a novel and alternative approach to test case generation satisfying MC/DC.

1.3 Research Methodology

Our research methodology is presented in Figure 1.3 which shows associated activities underlying our research on efficient techniques and tools for software testing based on traces and coverage analysis. The research methodology focuses on four main areas: theoretical foundations and approaches, software testing tools and techniques, test cases and testing scripts, and SUT case studies and experiments.

Theoretical foundations and approaches in our research methodology consist of conducting a literature review in the form of related work. That state of art helped us to develop and propose our approaches for efficient techniques and tools for software testing based on traces and coverage analysis, which can be used as the theoretical foundations.

Then, based on theoretical foundations, we implement software testing tools and techniques for MC/DC coverage analysis and data race detection. Our tools are

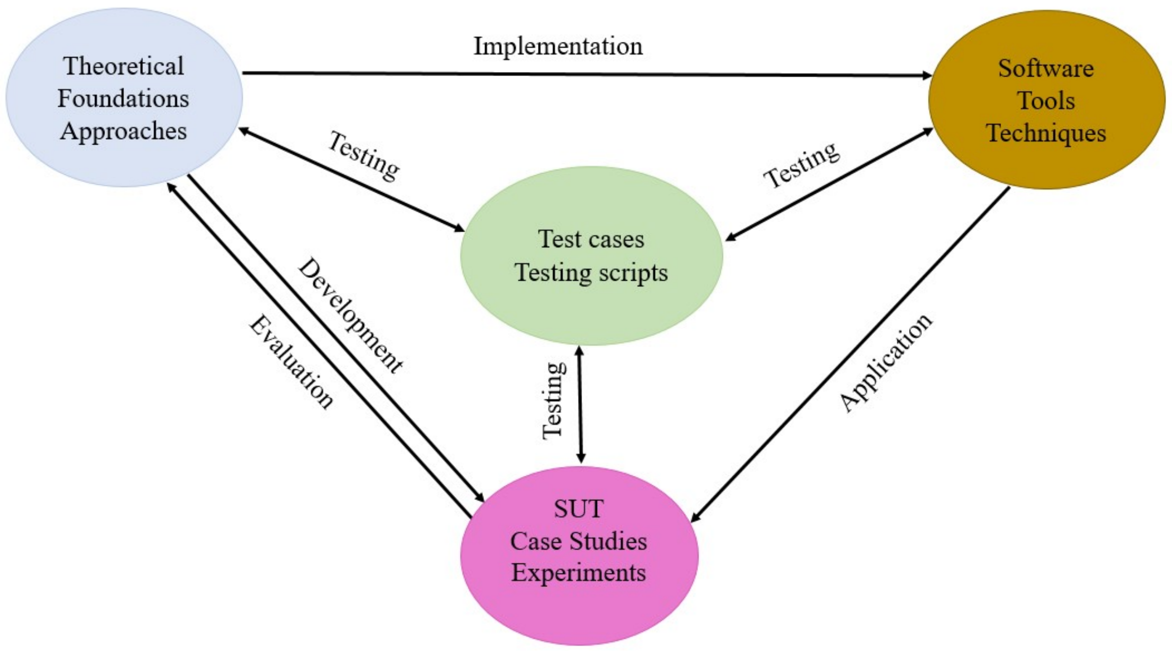


Fig. 1.3: Research method and underlying activities.

non-intrusive in the sense that they are either without instrumentation or with a light weight instrumentation.

We apply our approaches and techniques for coverage analysis and data race detection on case studies including models, benchmarks and other real examples. In this context, we investigated the applicability of MC/DC on the design level and how to monitor data races with low overhead of instrumentation.

As testing and coverage analysis require test cases and test scripts, we proposed and implemented the test cases selection/generation method that helps to test the systems under test from the case studies. Our test cases refer to the sample data that allow to check the actual behavior of the system under test based on the MC/DC criterion. The test scripts are the test drivers that help us to verify that both our tooling and methods work as expected. Moreover, we conduct a performance evaluation from the experiments of the case studies against the theoretical approaches proposed. We compare our proposed heuristics and provide recommendation to the heuristic that perform better than others. We provide open access to our tooling and guidance on how to use them.

1.4 Goals and Contributions

The contribution of this thesis is fourfold:

1. We provide an approach of how MC/DC can be measured non-intrusively by analyzing program traces. Our approach is based on the idea that every condition in the source code is translated into a conditional jump on the object code level and we can reconstruct how the conditions have been evaluated and the outcome of the decision from the trace. We first record the trace of an executing program and then analyze it offline. Program traces contain information about jumps taken during the execution and make it possible to reconstruct the evaluation of

each condition without instrumentation. We provide a non-intrusive MC/DC measuring tool and measured MC/DC for C programs [2, 9].

2. We present a *non-intrusive* approach to monitoring applications for data races detection on embedded system-on-chips (SoCs) using the COEMS platform [49]. This work eliminate the overhead of dynamic checking by offloading it to external hardware[2]. The platform offers control-flow reconstruction from processor-traces and data-traces through explicit instrumentation [125].

   This approach for data races detection is not directly related MC/DC analysis but it serves as a starting point of addressing data races in concurrent programs such that we could in the future explore how MC/DC information can be used for the analysis of concurrent programs.

3. We investigated the applicability of MC/DC measurement on the design level models where we considered specifically CPN models as case study. Our contribution to coverage analysis of net inscriptions in CPN models includes: 1) implementation of a CPN Tools library and annotation mechanism that intercept evaluation of Boolean conditions in guards and arcs in SML decisions in CPN models, and record how they were evaluated; 2) a post-processing tool that computes the conditions' truth assignment and checks whether or not particular decisions are MC/DC-covered or branch covered in the recorded executions of the model; 3) we provide visualization of coverage information in the CPN graphical user interface (GUI) such that it is easy to explore which part of the models is not covered; 4) we collect coverage data using our library from eleven publicly available CPN models and report whether they are MC/DC and BC covered; and 5) we visualize the coverage information in the CPN model such that the covered and uncovered transitions and arcs are revealed [10].

4. We propose a novel heuristics-based approach for generating test cases for a Boolean decision that satisfy the MC/DC criterion based on reduced ordered binary decision diagrams (roBDDs). For a decision of n conditions, we generate n pairs that contain between $n+1$ to $2n$ test cases altogether. We propose and compare heuristics with different preferences with respect to three-valued truth-values and the length of paths in the roBDD. We present an algorithm which is implemented in Python with the PyEDA library [56]. Our algorithm is tested on the Traffic Alert and Collision Avoidance System (TCAS II) benchmarks [156] which are widely used in the literature [71, 74, 86, 87, 161].

1.5 Thesis Outline

This dissertation is structured into two main parts. Part I presents a general introduction to software testing together with the research directions, research methodology, obtained results and contributions of this thesis. Part II consists of a collection of three published and peer-reviewed articles [2, 4, 7], and one submitted international journal paper [5].

[2]The EU Horizon 2020 project: "COEMS–Continuous Observation of Embedded Multicore Systems", https://www.coems.eu.

Part I contains of seven chapters and after the introduction, the remaining chapters are structured as follows:

Chapter 2: Background. This chapter presents a detailed background on coverage analysis and Modified Condition Decision Coverage (MC/DC) as the main criterion used for coverage analysis. It describes different techniques and theoretical foundations that were used for MC/DC measurement and MC/DC test case selection. It provides an overview on concurrent programs and data race detection. Furthermore, this chapter gives a background on different techniques that are used to explore the applicability of MC/DC to our case studies including programs and models.

Chapter 3: MC/DC Analysis and Measurement. This chapter describes how MC/DC can be measured non-intrusively by analyzing traces. It summarizes our findings presented in papers [2, 9], and puts our work into a state-of-the-art context through the discussion of related work.

Chapter 4: Data Race Monitoring in Concurrent Programs This chapter details the COEMS framework for hardware-assisted data race detection in concurrent programs. It illustrates the feasibility of our approach for data race detection in a Linux `pthreads`-based case study. It briefly summarizes our experimental results and gathers the related work and future work.

Chapter 5: Coverage Analysis on the Design Level Models This chapter discusses the application of our MC/DC measurement approach and techniques on design level models. We considered specifically the coverage analysis on net inscriptions in CPN models and obtained results are presented in our papers [5, 10] in Part II.

Chapter 6: Generating Test Cases Satisfying MC/DC. This chapter describes our approach and algorithm for generating test cases satisfying MC/DC based on binary decision diagrams (BDDs). It explains the implementation of our algorithm and summarizes the results together with related work. The detailed description is presented in the paper [4] in Part II.

Chapter 7: Conclusion and Future Work. This chapter provides the concluding remarks and discusses the future outlook of this thesis. It discusses the limitations of our approaches and proposes a way forward to addresses those limitations and challenges as well as the possible extensions of our work.

1.6 Supplementary Material

In addition to the papers [2, 4, 5, 7] presented in Part II of this thesis, one full paper, three conferences and workshop short papers and one pre-print have been published presenting initial research results:

[10] F. Ahishakiye, J. I. R. Jarabo, V. Stolz, L. M. Kristensen: Coverage Analysis of Net Inscriptions in Coloured Petri Net Models. In Proceedings of the International Conference on Verification and Evaluation of Computer and Communication Systems(VECoS), volume 12519 of Lecture Notes in Computer Science(LNCS), pages 68-83, Springer International Publishing, 2020, https://doi.org/10.1007/978-3-030-65955-4_6.

[9] F. Ahishakiye, F.D. Lange: Non-intrusive MC/DC measurement based on traces. In: Proceedings of the PhD Symposium at iFM'18 on Formal Methods: Algorithms, Tools and Applications (PhD-iFM'18), Maynooth, Ireland (Sept 2018), https://ifm2018.cs.nuim.ie/PhDSymposium.

[12] F. Ahishakiye, V. Stolz, L. M. Kristensen: Generating Test-cases Satisfying MC/DC from BDDs, The 31st Nordic Workshop on Programming Theory-NWPT'2019, https://doi.org/10.23658/taltech.nwpt/2019

[11] F. Ahishakiye, V. Stolz, L. M. Kristensen: Coverage Analysis of SML Expressions in CPN Models, In Proc. of the PhD Symposium at iFM'19 on Formal Methods: Algorithms, Tools and Applications (PhD-iFM'19), Bergen, Norway, 2-6, Dec. 2019. https://ifm2019.hvl.no/phd-symposium/.

[3] F. Ahishakiye, J. I. R. Jarabo, V. Stolz, L. M. Kristensen: Coverage Analysis of Net Inscriptions in Coloured Petri Net Models (2020), pages 1-20, https://arxiv.org/abs/2005.09806v1

All the tools developed and implemented for the publications included in this thesis are available for academic evaluation on the COEMS website[3], Github[4],[5] and the open repository [8].

[3]https://www.coems.eu/mc-dc/
[4]https://github.com/selabhvl/cpnmcdctesting
[5]https://github.com/selabhvl/py-mcdc/

CHAPTER 2

BACKGROUND

In this chapter we provide a detailed background on coverage analysis and the MC/DC criterion that was used for coverage analysis and the main techniques that were directly involved in our research. These include program tracing, data race detection, Coloured Petri nets (CPNs) models, and Binary decision diagrams (BDDs).

We present the state-of-the art on MC/DC and its different variants. In addition, we give an overview on MC/DC with short-circuiting logic and three-valued truth values as well as the background on MC/DC measurement on the object code level. Then, we look into concurrent programs specifically on data race detection as the property that we are concerned with. Next, we envisage different tracing facilities and source of traces which include the program traces and model traces. In this context, we provide an overview on CPN models as the case study considered for coverage analysis. We are using BDDs as a technique for generating test cases satisfying MC/DC criterion.

2.1 Coverage Analysis

While testing is meant to provide quality assurance for software products, there is still a need for better support to determine the effectiveness of the tests. Without coverage analysis, inadequate testing of software is likely to remain a major problem. Coverage analysis has been envisaged as a criterion for when to stop software testing activities for both sequential and concurrent programs [57, 94, 124]

There are two main measures of test coverage: requirements coverage and structural coverage [94]. *Requirements coverage* considers how well requirement- and specification-based test cases verified the implementation, and establishes a relationship between requirements and test cases. *Structural coverage* determines how much of the program or code structure was executed by the requirements-based test cases, and establishes traceability between the code structure and the test cases. Normally, requirements coverage analysis precedes structural coverage analysis. However, requirements may not have a complete specification of all behaviours present in the executable code. In addition, requirements may not be specified at a sufficient level of granularity to assure full testing of all functional behaviours of the code. Hence, requirements-based testing alone cannot confirm that the code does not include bugs.

Different structure testing coverage criteria have been proposed in the literature. It includes *statement coverage, branch coverage/decision coverage (BC/DC), path coverage, condition coverage (CC), condition/decision coverage (C/DC), multiple condition coverage*

(MCC) and *modified condition/decision coverage (MC/DC)* [42, 58, 62, 94, 99, 124, 167].

Statement coverage is a white box testing approach which checks if each statement in the source code is executed at least once [94, 105, 167]. It is calculated as the number of executed statements over the total number of statements in the source code. Statement coverage is considered inadequate because it is insensitive to some control structures. That is, if there is no test case that causes a conditional statement to evaluate as false, statement coverage rates the code fully covered, but the code may fail, if a condition ever evaluates false [48]. In addition, it does not report whether loops reach their termination condition, as it only checks that the loop body was executed.

Branch coverage is a coverage criterion intended to ensure that each decision from every branch is executed at least once [105, 167]. It allows to validate all the branches in the code and in addition it ensures that no branch lead to any abnormality of the program's operation. Both statement- and branch coverage are completely insensitive to the logical operators ($\vee$/|| and $\wedge$/&&).

Path Coverage which is also called *predicate coverage* checks whether each of the possible paths in each function entry to the exit have been followed [48]. It considers a sequence of branches or statements and evaluate combinations of branch decisions with other branch decisions which may not have been tested based on statement or branch coverage [105]. Path coverage requires very thorough testing and it is difficult to achieve 100% path coverage as the number of paths are exponential to the number of branches and that many paths are impossible to exercise due to data dependencies [48, 99].

The coverage criteria taking logical expressions into consideration have been defined and proposed [94, 105, 163]. A logical expression is a list of Boolean expressions all of which are required to evaluate to true or false. We refer to such Boolean expressions as *decisions*. They can be presented in a form of conditionals in if-then-else expressions, where a decision determines whether the then- or the else-branch will be taken. The following are the definitions for a condition and a decision [58].

Definition 1 (Condition)**.** *A **condition** is a Boolean expression containing no Boolean operators except for the unary operator NOT.*

Definition 2 (Decision)**.** *A **decision** is a Boolean expression composed of conditions and zero or more Boolean operators* (OR, AND, *or* XOR)*. It is denoted by D*$(c_1, c_2, c_i, \cdots, c_n)$*, where* c_i, $1 \leqslant i \leqslant n$ *are Boolean conditions.*

A Boolean expression is a predicate (which refers as well to a decision) that returns a Boolean value. We denote the truth values *true* and *false* by **1** and **0**, respectively.

As an example, we may have a Boolean expression in the if-then-else expression containing a decision of the form D = $((x \geqslant 6)\mathrm{AND}(y < 9))\mathrm{OR}(\neg(x \geqslant 6)\mathrm{AND}(z < 3))$. It can be abstracted using literals represented by lowercase letters such as a, b and c where a literal can be positive or negative (a or $\neg a$). The resulting decision is D = $(a \wedge b) \vee (\neg a \wedge c)$, where conditions a, b, and c represent $(x \geqslant 6)$, $(y < 9)$ and $(z < 3)$.

Condition coverage (CC) checks individual outcomes for each logical condition. CC offers better sensitivity to the control flow than decision coverage. The following is the definition of condition coverage [13, 94, 163]

Definition 3 (***Condition Coverage***)**.** *Each condition in a decision takes on each possible outcome at least once true and once false.*

Decision coverage (DC) requires each decision to be evaluated once true and once false [13, 94, 163]. It is commonly equated with branch or path coverage.

Definition 4 (***Decision Coverage***)**.** *Every point of entry and exit in the program has been invoked at least once and every decision in the program has taken on all possible outcomes at least once.*

Definition 5 (***Condition/Decision Coverage (C/DC)*** [163])**.** *Every condition in a decision has taken on all possible outcomes at least once, and every decision in the program has taken on all possible outcomes at least once.*

Multiple condition coverage (MCC) is an exhaustive testing of all possible combinations of conditions' inputs.

Definition 6 (***Multiple condition coverage*** [163])**.** *All possible combinations of the outcomes of the conditions within each decision have been executed at least once.*

Condition coverage (CC), Decision coverage (DC), Condition/Decision coverage)(C/DC) and Multiple condition coverage (MCC) have different limitations that need to be addressed. CC and DC are considered inadequate due to ignorance of the independence effect of conditions on the decision outcome. MCC requires 2^n tests for a decision with n inputs. This results in exponential growth in the number of test cases, and is therefore time-consuming and impractical for many test cases.

DC has another disadvantage that it ignores branches within Boolean expressions which occur due to short-circuit operators [48]. Short-circuit means that the right operand of the *and*-operator (&&/$\wedge$) is not evaluated if the left operand is false, and the right operand of the *or*-operator (||/$\vee$) is not evaluated if the left operand is true. Consider an example in Listing 2.1, the decision is evaluated to true when condition1 and condition2 are true whereas function1 is short-circuited in that case. When condition1 is false, the decision evaluates to false, condition2 is not evaluated and there is no call to function1.

```
if (condition1 && (condition2 || function1()))
  statement1;
else
  statement2;
```

Listing 2.1: Illustration of short-circuit evaluation

To address the limitations of the structure coverage criteria discussed above, *modified condition/decision coverage* (MC/DC) was proposed [42, 58]. In safety critical systems such as in the avionics industry, software certification requires a vendor to demonstrate that the test-suite provides MC/DC coverage of the source code. The MC/DC coverage criterion has been chosen as the coverage criterion for the highest safety level software because it is sensitive to the complexity of the decision structure [42]. Compared to even stronger criteria like multiple condition coverage (MCC), that requires every possible combination of all conditions, MC/DC may be satisfied with only $n + 1$ test cases for a decision with n conditions [13, 41, 94]. In addition, MC/DC coverage criterion is suggested as a good candidate for model-based development (MBD) using tools such as Simulink and SCADE [75]. Therefore, our model coverage analysis is based on MC/DC as a coverage criterion subsuming the other coverage criteria. The following MC/DC coverage definition is based on DO-178C [132]:

Definition 7 (Modified condition/decision coverage)**.** *A program is MC/DC covered and satisfies the MC/DC criterion if the following holds:*

- *every point of entry and exit in the program has been invoked at least once,*
- *every condition in a decision in the program has taken all possible outcomes at least once,*
- *every decision in the program has taken all possible outcomes at least once,*
- *each condition in a decision has shown to independently affect that decision's outcome by: (1) varying just that condition while holding fixed all other possible conditions, or (2) varying just that condition while holding fixed all other possible conditions that could affect the outcome.*

MC/DC is also known as Restricted Active Clause Coverage (RACC), General Active Clause Coverage (GACC) or Correlated Active Clause Coverage (CACC) [13]. To demonstrate MC/DC, a structural coverage analysis tool should monitor statements, entry and exit points, decision and branching statements as well as Boolean conditions [94]. However, the first item in the definition of MC/DC, is traditionally added to all control-flow criteria and is not directly connected with the main elements of MC/DC definition [152]. The most challenging and most interesting part is showing the independence effect of conditions [42, 58, 132]. By showing the independent effect of each condition, MC/DC demonstrates that each condition of the decision has a defined purpose.

The term decision is also used as the term *branch point* and the CAST-10 [37] position paper clarifies the meaning of decision in the context of the DO-178C [124]. It explicitly states that MC/DC should apply to all decisions, not just those within a branch point. That means that in addition to the decision within a branch point all Boolean operations that appear (i.e. in assignment statements) have to be considered. This avoid cheating MC/DC criterion.

As an example , the following decision:

```
if (A && (B || C)) then ...
```

can transformed as:

```
D = B || C
E = A && D
if E then ...
```

To show MC/DC for the first decision in the if-statement at least four test cases are required whereas the second decision in he if-statement can be covered by just assigning E to both True and False if a decision would be equal to a branch point. Therefore for MC/DC analysis, all logic structures need to be taken into account, not just the branch points and logical structure, that cannot be detected or are hardware-based, have to be evaluated externally.

2.2 Modified Condition Decision Coverage (MC/DC)

Certification standards for safety assurance such as DO-178C [124] in the domain of avionic software systems require software with the highest safety level (Level A) to satisfy MC/DC [42]. This criterion requires each condition to show an independent affect on the decision's outcome as defined in Definition 7 by DO-178C [124] and CAST-10 [37]. By showing the independence effect of each condition, MC/DC demonstrates that each condition of the decision has a defined purpose. The most challenging part in the Definition 7 of MC/DC is showing this independence effect: item (2) in the definition has been introduced in DO-178C to clarify that so-called *Masked MC/DC* is allowed [36, 124]. Masked MC/DC means that it is sufficient to show the independence effect of a condition by holding fixed only those conditions that could actually influence the outcome. MC/DC was developed to ease the testing of complex Boolean expressions in safety-critical applications such as traffic collision avoidance systems in aircrafts, patient monitoring systems in hospitals and nuclear power control systems [42]. MC/DC subsumes the existing logical coverage criteria such as condition coverage (CC), decision coverage (DC), and multiple condition coverage (MCC). Unlike other types of structural coverage, MC/DC can be applied to any representation (graphical or textual, mathematical or not) where logic is expressed.

There are three main forms of MC/DC according to [41], Unique-Cause (UC) MC/DC, Masking MC/DC and Unique-Cause + Masking MC/DC. UC-MC/DC is the original MC/DC which requires strictly an independence effect of each condition where only one condition changes at a time and cannot be achieved for a decision with coupled condition. Masking MC/DC is designed to handle coupled conditions and dealing with conditions whose changes do not affect the outcome when checking the independence of their peer condition. Coupled conditions are defined as follow:

Definition 8. ***(Coupled conditions).*** *Two or more conditions are said to be coupled if changing one condition can cause the other condition(s) to change (see Table 2.1a,* b *and* $\neg b$ *are coupled conditions). Conditions are said to be* ***strongly coupled*** *if changing one always changes the others. At the contrary, they are said to be* ***weakly coupled*** *if changing one sometimes (but not always) changes the others.*

2.2.1 *Unique cause MC/DC (UC-MC/DC)*

Unique-Cause (UC) MC/DC is defined by DO-178C [124] as the original MC/DC with the same original interpretation of *"independent effect"* requirement for each condition in a decision. A condition is toggled once true and false showing the independence effect of that condition on the outcome while holding all other possible conditions fixed. It requires a strict selection of UC pairs for every condition in the decision and it cannot be satisfied if a decision contains strongly coupled conditions.

In the context of selecting UC-MC/DC independence pairs, two variant definitions of UC-MC/DC were identified [42, 97, 121, 163].

1. *Weak UC-MC/DC*: This variant consists of selecting one MC/DC independence pair for any occurrence of a condition. It is explicitly required that one pair is selected for any occurrence of a condition in the form of the black box approach

tc	a	b	c	D	MC/DC pairs
1	0	0	0	0	
2	0	0	1	0	a(2,6)
3	0	1	0	0	a(3,7)
4	0	1	1	0	
5	1	0	0	0	a(4,8)
6	1	0	1	1	c(5,6)
7	1	1	0	1	b(5,7)
8	1	1	1	1	

(a) MCC & All UC-MC/DC pairs

π	a	b	c	D	MC/DC pairs
1	0	0	1	0	
2	1	0	0	0	a(1,3)
3	1	0	1	1	c(2,3)
4	1	1	0	1	b(2,4)

(b) Weak UC-MC/DC pairs

tc	a_1	b	a_2	c	D	pairs
1	0	1	1	0	0	a_1(1,5)
2	1	0	0	1	0	
3	1	0	1	1	1	a_2(2,3)
4	1	0	1	0	0	c(3,4)
5	1	1	1	0	1	b(4,5)

(c) Strong UC-MC/DC pairs

Table 2.1: MCC & MC/DC pairs for $D = (a \wedge b) \vee (a \wedge c)$

of module testing [163]. For example, the decision $D = (a \wedge b) \vee (a \wedge c)$ is considered to have only three conditions, a, b and c, hence, requiring only one independence pair for each of the conditions.

2. *Strong UC-MC/DC*: In this case one MC/DC independence pair is selected for every occurrence of a condition. According to [58], if a condition appears more than once in a decision, each occurrence is a distinct condition. For instance the decision $D = (a \wedge b) \vee (a \wedge c)$, contains four conditions where the occurrence of condition a in the first and third position are treated as different conditions. The expression would be written as $D = (a_1 \wedge b) \vee (a_2 \wedge c)$.

Example 1. The truth table representing all eight possible test cases (combinations) for MCC for the decision $D = (a \wedge b) \vee (a \wedge c)$ is given in Table 2.1. In this table, the MC/DC column lists conditions (here a,b, and c) together with a pair of test cases that demonstrate the independence effect of the particular condition. For an example, the MC/DC pair $a(2,6)$ specifies that from test cases 2 and 6 we can observe that changing the truth value of a while keeping the values of b and c, we can affect the outcome of the decision. Comparing MCC to MC/DC in terms of the number of test cases, there are seven possible MC/DC test cases (test cases 1 through 7) that are part of an MC/DC pair, where condition a is represented by 3 pairs of test cases showing the independence effect of condition a, and one pair of test cases for conditions a and b. However, all seven test cases provided in Table 2.1a are not necessary to ensure MC/DC coverage. For weak UC-MC/DC, only four test cases (1,2,3, and 4), i.e., $n+1$ test cases for a decision with three conditions are required to achieve MC/DC coverage as shown in Table 2.1b. For Strong UC-MC/DC, we have five test cases represented in Table 2.1c because the two occurrence of condition a are treated as two distinct conditions.

Definition 9 (Independence effect of a condition, independence pair, $\oplus_c$)**.** *Given two test cases* tc, tc' *for a decision* D, *we call* tc independent from tc' on condition c, *iff i)* $D(tc) = \neg D(tc')$ *(they evaluate to opposite truth values), and ii)* $tc \oplus_c tc'$, *where* $\oplus_c$ *means they differ exactly only in the input position corresponding to condition* c. *We then say that* tc *and* tc' *form an independence pair (for some condition* c*), written uc*(tc, tc').

For example, test cases 2 and 6 in Table 2.1a form an MC/DC pair and they show an independence effect of condition *a* in the decision. Similarly for the rest of the conditions. Note that it is possible for a condition to have more than one MC/DC pair which is the case for condition *a* in Table 2.1a. In our context, we can reformulate the general definition of MC/DC from Definition 7 for our purposes [4]:

Definition 10 (MC/DC-cover). *Given a decision* D *and set of test cases* ψ*, we say that* ψ MC/DC-covers D*, iff* $\forall c \in D, \exists tc, tc' \in \psi : tc \oplus_c tc' \wedge uc(tc, tc')$ (tc *is independent from* tc' *for every condition* c*).*

In other words, a set is an MC/DC-cover for a decision D, if for every condition, there exists a pair of test cases in that set which shows the independence effect of that condition by evaluating to opposing truth values.

2.2.2 *Unique cause MC/DC + masking MC/DC (UCM-MC/DC)*

This Unique cause MC/DC + Masking MC/DC (UCM-MC/DC) extends UC-MC/DC so that masking is allowed for only strongly coupled conditions. For this type of MC/DC, the uncoupled conditions are required to show UC-MC/DC. This means that except for the condition under-test all other possible uncoupled conditions must be fixed excluding these that are strongly coupled [42]. Similar to the original MC/DC, there are two possible ways of selecting independent pairs that satisfy UCM-MC/DC [34, 41, 92, 141]:

1. *Weak UCM-MC/DC*: For any occurrence of an uncoupled condition, one UC pair is selected, and one masking pair for any occurrence of each strongly coupled condition.

2. *Strong UCM-MC/DC*: It consists of selecting one UC pair for every occurrence of an uncoupled condition, and one masking pair for every occurrence of the strongly coupled condition.

The main difference between weak UCM-MC/DC and strong UCM-MC/DC is the resulting number of test cases where the later consider every occurrence of a strongly coupled condition as a new condition and hence resulting in a higher number of test cases.

2.2.3 *Masking MC/DC*

As its name implies, Masking MC/DC allows masking in all cases not only for strongly coupled conditions. It is the weakest form of MC/DC. A condition is considered masked, if varying that condition cannot affect the outcome of a decision [34, 41, 141].

For example, it is sufficient to show the independence effect of a in $D = a \vee (b \wedge c)$ by holding the sub-expression $b \wedge c$ fixed to *False* even if the values of b and c are changing [41]. The position paper CAST-6 [36] the Certification Authorities Software Team (CAST) compared UC-MC/DC and Masking MC/DC and concluded that Masking MC/DC meets the intent of the MC/DC objective and is therefore an acceptable method for meeting MC/DC with applicants striving to the objectives of DO-178B, level A. Table 2.2 contains Masking MC/DC pairs with independence effect of each condition.

TC	conditions			Decision	evaluation		
	a	b	c	a ∥ (b && c)	a	b	c
1	0	0	0	0	0	0	?
2	0	0	1	0	0	0	?
3	0	1	0	0	0	1	0
4	0	1	1	1	?	1	1
5	1	0	0	1	1	?	?
6	1	0	1	1	1	?	?
7	1	1	0	1	1	?	?
8	1	1	1	1	1	?	?

(a) Short-circuit evaluation for D = a ∥ (b && c).

TC	a	b	c	D	Masking MC/DC pairs
1	0	0	?	0	
2	0	1	0	0	c(2,3)
3	?	1	1	1	b(1,3)
4	1	?	?	1	a(1,4)

(b) Independent Masking MC/DC pairs

Table 2.2: Short-circuit evaluation and Masking MC/DC pairs for D = a ∥ (b && c)

2.2.4 *MC/DC with short-circuit logic and three-valued truth values*

The short circuit logic known also as "don't care" corresponds to a software optimization which consists of skipping the evaluation of some conditions or Boolean expressions which do not influence the decision outcome. In most modern programming languages, Boolean expressions are evaluated in strict order (left to right) and by using short circuit logic. The left operand can always be evaluated first and the right operand is only evaluated if its value is needed to determine the result of the decision. For example, the right operand of the *and*-operator is not evaluated if the left operand is False and right operand of the *or*-operator is not evaluated if the left operand is True. Some programming languages also provide short circuit control forms.

Short circuit logic also occurs when compiler optimizations are selected which do not require all of the operands within an expression to be evaluated once the output has been determined. This is important for programming languages that use short-circuit evaluation, because certain executions of decisions are not distinguishable, if the outcome of the decision is determined before every condition has been evaluated. Short circuit logic, whether by language construct or compiler optimization, is similar to the masking evaluation discussed in section 2.2.3.

In [94] examples of MC/DC with short circuit logic are presented where short circuit expressions can be treated in the same manner as conventional and and or gates. In Table 2.2 we present the short-circuit evaluation for D = a ∥ (b && c) with all possible inputs and select independent Masking MC/DC pairs. The "don't care" value "?" means that the condition has not been evaluated at all due to short-circuit evaluation. It can be seen that the test cases 5, 6, 7 and 8 are not distinguishable by looking at the evaluated conditions because of the $|| - \mathtt{operator}$. Test cases 1 and 2 show the same behavior with respect to the $\&\& - \mathtt{operator}$. This is because the left-hand operand alone determines the outcome of the decision, the right-hand operand can be considered as masked in sense of Masked MC/DC [47]. Binary decision diagrams (BDDs) also are used to illustrate the short-circuit evaluation in Section 2.7. Figure 2.3b show a reduced ordered BDD (roBDD) for the decisions $D = a \mid\mid (b \; \&\& \; c)$ with short-circuit evaluation.

Therefore, Masking MC/DC complies with the second part of DO-178C definition 7: "each condition in a decision has shown to independently affect that decision's outcome by ... (2) varying just that condition while holding fixed all other possible conditions that could affect the outcome.", because a condition that is not evaluated cannot affect a decision's outcome. Table 2.2 presents Masking MC/DC pairs that show the independence effect of each condition. We define the set of test cases with short-circuit, as the three valued test cases.

Definition 11 (Two/Three-valued test case). *Given a decision* D, *a* test case *is a truth vector* $tc = (I_1, I_2, I_3, \cdots, I_n)$ *where* $I_i \in \{0, 1\}$ *(respectively,* $\{0, 1, ?\}$*) are the inputs assigned to each conditions. ? is known as "don't care" meaning that a condition does not need to be evaluated due to short-circuiting. A* set of test cases *for a given decision is called a* test suite*. We denote the projection onto the truth-value at the position corresponding to some condition* c *in the test case* tc *as* $tc[c]$.

2.2.5 MC/DC measurement on object code

MC/DC measurement can be performed either at the object code or the source code level. Achieving MC/DC at the object code level is not necessarily equivalent to achieving MC/DC at the source code level [41]. MC/DC may be demonstrated at the object code level [35, 38], however, the analysis must show that coverage at the object code level is equivalent to coverage at the source code level. That is, it requires coverage traceability from object code to source code. Therefore, a static analysis of object code and back to source code needs to be conducted. Then MC/DC can be measured based on the sequence of jumps performed by the CPU by finding out which conditions in the source code correspond to which conditional jumps in the object code. This mapping is necessary in order to establish the correctness of such generated code sequences.

We presents in Listing 2.2 the source code for decision $(a \mid\mid (b == 5 \; \&\& \; c > 3))$ and its object code representation is shown in 2.3. The translation to object code is performed using the Clang compiler.

```c
int decision(int a, int b, int c){
 if (a || (b==5 && c>3)){
  return 1;
 }
 else{
  return 0;
 }
}
```

Listing 2.2: Source code for decision $(a \mid\mid (b == 5 \; \&\& \; c > 3))$

It is possible to reconstruct how the conditions were assigned during the execution based on this mapping and the program trace evaluation. In order to perform a reconstruction of the condition assignments it is required that every condition on the source code level translates to one specific conditional jump on the object code level. Listing 2.3 shows conditional jumps in lines 7, 9, 11 that correspond to the decision in line 2 of Listing 2.2 and 13 for the decision evaluation. However, this assumption does not hold if the compiler uses any optimization level that influences conditional jumps because even for the first optimization level (for example for `gcc` compiler: the options `-fif-conversion`) conditional jumps are translated into branch-less equivalents.

```
1  0x0000000000400480 <+0>: push   %rbp
2  0x0000000000400481 <+1>: mov    %rsp,%rbp
3  0x0000000000400484 <+4>: mov    %edi,-0x8(%rbp)
4  0x0000000000400487 <+7>: mov    %esi,-0xc(%rbp)
5  0x000000000040048a <+10>: mov    %edx,-0x10(%rbp)
6  0x000000000040048d <+13>: cmpl   $0x0,-0x8(%rbp)
7  0x0000000000400491 <+17>: jne    0x4004ab <decision+43>
8  0x0000000000400497 <+23>: cmpl   $0x5,-0xc(%rbp)
9  0x000000000040049b <+27>: jne    0x4004b7 <decision+55>
10 0x00000000004004a1 <+33>: cmpl   $0x3,-0x10(%rbp)
11 0x00000000004004a5 <+37>: jle    0x4004b7 <decision+55>
12 0x00000000004004ab <+43>: movl   $0x1,-0x4(%rbp)
13 0x00000000004004b2 <+50>: jmpq   0x4004be <decision+62>
14 0x00000000004004b7 <+55>: movl   $0x0,-0x4(%rbp)
15 0x00000000004004be <+62>: mov    -0x4(%rbp),%eax
16 0x00000000004004c1 <+65>: pop    %rbp
17 0x00000000004004c2 <+66>: retq
```

Listing 2.3: Object code for decision $(a||(b == 5\&\&c > 3))$

The binary operators can be omitted in case symbolic conditions are used (as in in case of condition *a*), which are commonly used for single Boolean expressions. The above expression would be the same as $(a \;||\; (b \;\&\&\; c))$ which is used in Figure 2.3 and Table 2.2. Note how every condition is translated into a conditional jump and short-circuit logic is used, if the target of a jump skips the evaluation of other conditions. For example, if the jump in line 7 is taken, the other conditional jumps in lines 9 and 11 are not evaluated at all.

Depending on the relational operator in the condition (for example: <, <=, and ==), two different possible conditional jumps can be generated by the compiler because conditions can be translated to their negation (it is up to the compiler to choose "jump-if-equal" or "jump-if-not-equal"). If a condition is translated as its negation, this has implications for the reconstruction of the assignments by analyzing the trace as a taken jump shows that the condition has been evaluated as **false**. The possible combinations for the Intel x86-64 instruction set and its ARM counterpart are shown in Table 2.3, which have to be taken into account when the reconstruction is performed.

2.3 Concurrent Programs

Today multi-core/multi-processor hardware is in main stream use with the main capability of implementing parallelism and concurrency to increase processing speed and optimize the resource sharing. However, available resources and the processing speed may not be used efficiently if different program components execute in sequence on all processors. That is processes are executed in order where a process is started when the preceding process has finished. There is always only one process per processor being executed concurrently. Thus, it is necessary to develop concurrent programs that utilize available resources where multiple processes are executed to perform a job together [16, 111].

Figure 2.1 compares process execution for sequential versus concurrent programs. Figure 2.1a shows the sequential processing where one process executes after another and Figure 2.1b presents the concurrent processing where two or more actions are executing at the same time. A process is defined as a unit of program execution as seen by an operating system. A process has its own address space, file handles, and threads. A unit of control within a process is known as a thread and it will execute a function in the program whenever it runs. Threads have their own program counter and register values, but they share the memory space and other resources of the process. Contrary to a sequential program that has a *single thread of control*, there are *multiple threads of controls* for a concurrent program.

Due to the non-deterministic behavior of concurrent programs, their testing involves complexity which makes them prone to faults, difficulties in sharing global resources, management of allocation of resources and difficulties in locating programming errors. In addition, there are other problems associated with debugging concurrent programs such as the "*probe effect*", non-repeatability, and the lack of a synchronized global

Relational Operator:	Possible Conditional Jumps:		Condition Value of Detected Jump:
	x86-64	ARM	
no operator	`jne`	`bne`	**True**
	`je`	`beq`	**False**
==	`je`	`beq`	**True**
	`jne`	`bne`	**False**
<	`jl`	`blt`	**True**
	`jge`	`bge`	**False**
<=	`jle`	`ble`	**True**
	`jg`	`bgt`	**False**
>	`jg`	`bgt`	**True**
	`jle`	`ble`	**False**
>=	`jge`	`bge`	**True**
	`jl`	`blt`	**False**

Table 2.3: Multiple interpretations of jumps in the x86-64 and ARM instructions sets compiled with clang version 5.0 [17]

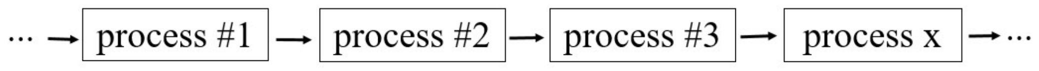
clock [111]. Probe effect denotes the behavioral changes in the frequency of run-time computational errors caused by delays introduced into concurrent programs due to the insertion or removal of code instrumentation [64]. It refers to the unintended behavior of the system when attempting to observe its behavior. Even without any attempt to observe the program behavior, there may be non-repeatable behavior for some concurrent programs where different executions with the same data yields different results. Even when the behavior can be observed, it may be difficult to interpret the results of the observation due to the lack of a synchronized global clock [111]. Testing concurrent software exhaustively is not practicable because of the huge interleaving space (the total number of execution orders between processes). Thus, there is a need of synchronization between processes [57, 64, 104]. Section 2.3 provides a background on the main aspects of concurrent programs such as thread creation, thread synchronization and communication. We provide an overview on the detection of concurrent programming errors such as deadlocks, livelocks, starvation, and data races [28]. In addition, we discuss different methods proposed to deal with non-deterministic behavior in concurrent programs, such as locking, serialization, and time stamp.

2.3.1 *Thread creation*

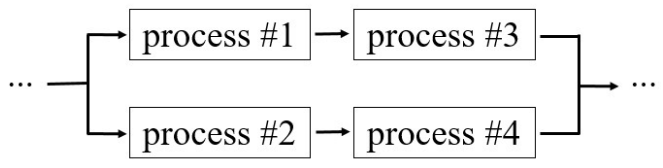
A thread is defined as a single sequence of executable statements within a program. It is also known as a thread of execution or a thread of control. Within a single thread, the sequential flow of execution from one statement to the next can be traced.

Note that the main difference between threads and processes is that threads within the same process run in a shared memory space, while processes run in separate memory spaces. Both processes and threads have their own program counter (PC), register set, and stack space.

It is possible to implement *multithreaded* programs where multiple threads can run concurrently by alternating them. To create threads, our illustration is based on C programs as shown in Listing 2.4. The first step is to include the file *"pthread.h."* and then a thread is created and started using the function *pthread_create()*. Each thread takes four parameters: the pointer to the thread ID with a specific object of type *pthread_t* associated with it, the attributes of a thread, the function that the thread starts to execute, and the argument that the function takes.



(a) Sequential processes



(b) Concurrent processes

Fig. 2.1: Comparison of processes for sequential versus concurrent programs

```c
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <inttypes.h>
pthread_mutex_t m,l; //two locks
int x=0, y=0;  //global variable

void* f(void *arg) { //The function that the thread starts to execute.
 for ( int i = 0; i < 25; i++ ) {
  pthread_mutex_lock(&m);//acquire the lock m
  x++;
  pthread_mutex_unlock(&m);//release the lock m
 }
 return NULL;
}
void* g(void *arg) { //The function that the thread starts to execute.
 for ( int i = 0; i < 25; i++ ) {
  x++;
 }
 return NULL;
}
void* f(void *arg) { //The function that the thread starts to execute.
 for ( int i = 0; i < 25; i++ ) {
  pthread_mutex_lock(&l);//acquire the lock l
  y++;
  pthread_mutex_unlock(&l);//release the lock l
 }
 return NULL;
}

int main() {  //Entry point of program
 pthread_t p1; //First thread ID
 pthread_t p2; //Second thread ID
 pthread_mutex_init( &m, NULL );
 pthread_create( &p1, NULL, f, NULL );// Create thread for function f
 pthread_create( &p2, NULL, g, NULL );// Create thread for function g
 pthread_join(p1, NULL); //wait for peer thread
 pthread_join(p2, NULL);  //wait for peer thread
 printf( "x= %d\n", x );
 return 0;
}
```

Listing 2.4: Threads creation and locking

Listing 2.4 shows two threads which execute the functions f and g and contains two locks m and l. Using *pthread_join()*, a thread can be made wait for another thread where its ID is passed as an argument. In addition, the value which will be returned by exiting a thread can be passed as an argument. As the local variables are destroyed when a thread exits, only references to global or dynamic variables should be returned.

2.3.2 *Communication between threads*

Communication between threads refers to control mechanisms where threads are able to correctly transmit data among them. Threads frequently have to interact with each other to accomplish a work together and forming a dependency between them. With such dependency, a dependent thread will typically have some knowledge about the states of the partner thread. Normally, a thread requires that another thread is in a specific state before proceeding with an operation. A thread is said to be "*causally dependent*" on another thread, if it is affected by its state changes (such as by reading memory that it has written).

In concurrent programs, there exist two main methods of communication between threads [16]:

– **Shared memory**: All the threads have access to the same memory but they are working on different chunks of data. However, some threads can use the results from others. Thus, threads cooperate with each other to perform a given task together by by communicating via shared memory. For example one thread can write into a variable which is read by another thread.

– **Message passing**: one thread sends a message that is received by another thread. With this means of communication, a queue is used to store the message until it is processed. Communication channels provide a one way path from a sending to a receiving process and channels are FIFO queues of pending messages.

Critical section. A critical section in concurrent programs refers to a region of the program where a shared resource is accessed [16]. This part of the program needs to be protected so that conflicting access is avoided. In other words, if the region is executed by more than one process (or thread) at a time, it yields wrong results and this is referred to us as *mutual exclusion*. We will discuss later the possible solutions to the *critical section problem* [54, 93].

For example in the Listing 2.4, the global x (line 7) can be accessed by both function f (line 12) and function g (line 21). The access can be either a write or read to x. If p1 executing function f needs to read the right value of x, executing p1 and p2 at the same time may give wrong results for x, especially when there is no protection of the global variable (see for example in function g where access on x is not protected). To avoid the conflict, the variable x need to be protected. First, p1 gets the access to the critical section. Once p1 finishes writing the value, p2 gets the access to the critical section and variable x can be read and be written. To prevent conflicting access to the shared variable, it is necessary to control which variables are modified inside and outside the critical section. The shared variables need also to be synchronized to maintain consistency of data variables. It is assumed that a process that enters its critical section will eventually exit, where for instance it can terminate outside the critical section [16].

A race condition. In concurrent programs a race condition occurs when more than one process are accessing the same memory at the same time and at least one of them writes to that memory location [21]. If several processes access and perform the manipulations over the same data concurrently, then the outcome depends on the particular order in which the access takes place. A race condition occurs inside a critical section when the result of multiple thread executions differs according to the order in which the threads execute.

The outcome of the data race is determined by the interleaving which is also depends to different factors such as processor load, network traffic, non-determinism in the communication protocol and timing of events [136].

It is in general impossible to determine in advance the outcome of race conditions as concurrent programs are not predictable with respect to its interleaving space. Changing one of the factors mentioned above, for example, the processor load, is enough in order to get a different outcome of such a race. Moreover, a different outcome of a race condition may give different behavior. The non-deterministic behavior of concurrent software tends to make them more difficult to understand, write and debug, compared to sequential software [111].

2.3.3 *Thread synchronization*

To avoid data races in concurrent programs, there is a need to control execution order (interleaving) of instructions of concurrent tasks. Synchronization refers to the interaction between processes that controls the order in which the processes execute [16]. Different threads compete for time on the same processor or may execute in parallel on separate processors. Therefore, thread synchronization ensures timing relationships among threads. There are two basic types of threads synchronization in concurrent programs: *mutual exclusive* and *condition synchronization*.

- *Mutual exclusive* synchronization ensures that only one process enter the critical section at a time for the resource. One of the requirements for the mutual exclusive is that a process must not be delayed to access to a critical section when there is no other process using it and a process remains inside its critical section for a finite amount of time only. The other thread waits until the current thread has reached a certain point in its code.

- *Condition synchronization*: ensures that the state of a program satisfies a particular condition, therefore a process is delayed until a condition is *true*.

An example of these types of synchronization is the communication between producer and consumer processes [16]. Their communication can be implemented using shared memory buffer. The producer writes to the buffer while the consumer reads from the buffer. It is required to have mutual exclusion between the two processes to avoid that the producer and consumer access the buffer at the same time. Therefore, the partially written message is not read before it is ready to be read. In addition, the condition synchronization will ensure that the message is not read by the consumer before it has been written by the producer.

To ensure concurrent program control and data synchronization, other synchronization primitives that are defined includes:

- *Semaphores*: They were first introduced by Dijkstra [54] to define the critical section problem. They allow certain patterns of data synchronization in which a fixed number of threads are permitted to be inside the critical region simultaneously.

- *Locks and barriers*: One of the ways to implement mutual exclusion is to use locks to protect critical sections. Exclusive locks are used to control threads accesses to the same location. Listing 2.4 shows the usage of `pthread_mutex_lock` to control access to global variable x (line 10-12). There will be no conflicting access to x as long as the first thread is holding the lock. We observe the incorrect locking in Listing 2.4 from line 16-21, where the thread p2 accesses x without any lock used. Hence this will result in a data race.

- *Compare and swap*: This an an atomic instruction that compares the contents of a memory location with a given value and, only if they are the same, modifies the contents of that memory location to a new given value.

2.4 Data Race Detection

As we introduced in Section 2.3, a data race in a multi-threaded program may rise when two or more threads access the same memory location concurrently, and at least one of the accesses is a write while the threads are not using any synchronization mechanism to control their accesses to that memory [21, 135, 137].

There are two techniques to detect data races: *static data race detection* and *dynamic data race detection*. Sometimes a combination of both approaches can used as a *hybrid technique*. Static data race detection techniques perform a compile-time analysis of the code. While static techniques can warn about all possible bugs in all possible executions, they may result in an excessive number of false alarms (false negatives) that hide the real data races due to over-approximations of the behaviour of the program. Dynamic data race techniques analyze programs at run-time and use a tracing mechanism to detect whether a particular execution of a program actually exhibited data races. They detect only those apparent data races that occur during execution and they locally consider only one specific execution path of the program each time. Dynamic data races are hard to detect because they depend on the particular order in which the accesses take place. That is, dynamic bugs may be related to interaction of two different processes, and their manifestation may change over time where re-running the program may not always produce the same results.

There are two main methods for dynamic data race detection: *trace based post-mortem* and *on-the-fly*.

Post-mortem methods: After the execution terminates, the traced information is analyzed.If there are possible data-races found, a warning is raised.

On-the-fly methods: They report data races as they occur during an execution of a program. These methods contain three different analyses: *lockset analysis, happens-before analysis, and hybrid analysis*

- *Lockset-based approaches* [72, 135] detect data races based on the set of locks protecting shared-variable accesses. That is checking violations of a locking discipline. The limitation of lockset-based approaches is that they often report many false data races (false positive), because the related shared variables are not actually concurrently accessed.One of the most frequently used lockset-based approaches is proposed and implemented in Eraser [135]. The proposed *lockset algorithm* is based on the assumption that access to a shared variable is always protected by a set of locks and any thread accessing the variable must hold a lock.

It is necessary that the lockset algorithm knows about relevant events with respect to *locking* and *memory accesses*. For example it needs to know which thread is taking which lock and which thread is reading from or writing to a given memory location. To address this, Jakšić et al. [80] proposed a basic version of the lockset algorithm where for each shared variable:

1. the accesses to the shared variable is first identified;
2. on each access to the variable, the set of locks held by the thread accessing the variable is identified;
3. the set of locks guarding the variable with the set of all locks is initialized;
4. on each access to the variable, the set of locks guarding the variable is updated by intersecting it with the set of locks held by the thread which is accessing the variable.

They present a stream-based dynamic data race detection based on the TeSSLa specification language [100] with the help of a dynamic data structure monitoring platform that records lock operations and memory accesses. The proposed lockset algorithm is able to detect the error in the example presented in Listing 2.4 in a single run, regardless of the final value of x. In [7], we extend this approach for detecting data races using the COEMS technology through continuous online monitoring with low-impact instrumentation on a FPGA-based external platform for embedded multicore systems. The summary of our findings about this approach is provided in Chapter 4 and detailed results are in Part II.

– *Happens-before analysis* [40, 72] reports data races between a current access and maintains previous accesses by comparing their happens-before relation based on the usage of a logical time stamp, such as vector clocks. For example, when two conflicting memory accesses a_1 and a_2 are on the same memory location, and neither a_1 happens before a_2 nor a_2 happens before a_1, a data race may occur. An advantage of happens-before analysis is its precision, since it does not report false positives and can be applied to all synchronization primitives. However, it involves complexity to be efficiently implemented due to the performance overhead.

– *Hybrid analysis* [72] reduces the main drawback of pure lockset analysis and provide a high performance than pure happens-before analysis.

2.5 Source of Traces and Tracing Mechanisms

In this section we provide an overview on the sources of traces and tracing facilities that are used in this thesis. Our coverage analysis is conducted without instrumentation but rather it is based on traces. In other cases such as data race detection and model tracing we use a lightweight instrumentation. There exists many tracing facilities and sources of tracing, however, we put much emphasis on the main tracing mechanisms used in this thesis.

2.5.1 *Source of traces*

Traces can come from a model or a program as a record of its behavior during the execution.

The *model-based traces* refer to the behavior of the model from a system's design during its execution, allowing to combine model-driven engineering with dynamic analysis. Specifically, we obtain traces from CPN models that are collected during the state space exploration and simulation in the form of logs. For more detail on CPN models we refer the reader to Section 2.6. Within CPN Tools, there is no coverage analysis of the SML expressions in a CPN model. This means that to record coverage data for a CPN model under test, it is necessary to instrument the Boolean expressions such that the truth-values of individual conditions are logged in addition to the overall outcome of the decision. Our approach to instrumentation makes use of side-effects by outputting intermediate results of conditions and decisions, which we then process to obtain the coverage verdict. No modifications to the net structure of the CPN model are necessary. Furthermore, the instrumentation has little impact on model execution so that it does not delay the simulation and state space exploration (SSE).

Program tracing is an important mechanism for developers for gathering useful information for debugging, monitoring and performance analysis of an executing program. It enables post-analysis of a software execution through the minimal recorded information necessary to reconstruct complete program control-flow [68]. The program trace consists of a sequence of addresses of instructions executed, and different types of data referenced while a program runs. These include the program flow information such as branch targets, branch taken/not taken indications and program-induced mode related information such as state transitions.

To enable a trace tool to reconstruct the instruction execution sequence and jumps executed by the processors efficiently, there exists tracing facilities and technologies for different processors such as: 1) the *Intel Processor Trace (PT)* used by Intel to trace program execution [15, 68], and 2) the *ARM CoreSight* used by ARM processors for debugging and tracing multicore SoCs [17].

2.5.2 *Intel processor tracing (IntelPT) facility*

Intel Processor Tracing (Intel PT) is an extension of the Intel Architecture that traces program execution with low overhead [15]. Intel CPUs have an older mechanism called Branch Trace Store (BTS). However, the BTS is estimated to incur a significant performance slow-down in a range of factor of twenty to forty [68]. Intel PT can be used by modern Intel CPUs such as Intel Broadwell (5th generation), Skylake (6th generation Core, Xeon v5), and Goldmont (Apollo Lake, Denverton) CPU. Intel PT was introduced to provide an accurate and detailed trace with triggering and filtering capabilities [147]. Intel PT works by capturing information about software execution on each hardware thread using dedicated hardware facilities so that after execution completes, software can do processing of the captured trace data and reconstruct the exact program flow.

Intel PT uses an extremely compact format that makes it possible to overcome the small bandwidth and limited buffer space by only storing information about taken and not taken branches, indirect branches, function returns and interrupts. Based on these, the complete program execution flow can be reconstructed.

The traces contain instructions executed by the processor, but there are no data values. For example, for the C-level instruction `x = y + z`, as the trace essentially only consists of instruction pointers, we can only reconstruct the assembly instructions for loading the values, summation and storing the result in memory, and maybe even map them onto the source-code, but we have no information about the actual values of x, y and z or their location in memory during execution.

To record and decode Intel PT, *perf record*[1] and *perf script* are used on Linux, respectively with additional options that limit the amount of trace data captured to the specified interest of the users. In addition to the *perf record* tool, three tools can be used to deal with the recorded trace: *perf script, perf report and perf inject* [95].

- *perf script*[2] decodes the trace and reads the recorded data file and display the trace output. There exists several variants of *perf script* allowing one to obtain a detailed trace of the workload that was recorded.

- *perf report* reads the recorded data file and displays the resulting information. Its main role is to display the information recorded through *perf record*.

- *perf inject* reads a perf-record event stream and repipes it to stdout. It allows also to inject other events into the event stream at any point.

More details on how to use *perf* tools are given on its manual pages [15, 95]. For example, to trace a C program called readers-Writers.c, *perf* tool is used on the executable of the compiled program (let say*"bin/readers-Writers.out"*).

```
perf record -e intel_pt -o pt_ls/rdWr.perf bin/readers-Writers.out
```

which will create a directory named pt_ls and add in there the rdWr.perf file.

The recorded trace can be displayed with *perf report* (e.g. *"perf report -i pt_ls/rdWr.perf"*). It can be decoded and converted to a text file using the following command:

```
perf script --itrace=i0ns --ns -F ip -i rdWr.perf >decoded-rdWr.txt
```

Where the option *"itrace=ions"* defines "instructions" events and in this case it allows to see every possible instructions-per-cycle (IPC) value. The decoded trace is a sequences of instruction addresses and it contains no data as shown in the Listing 2.5. Further analysis can be done over the decoded trace and the reconstruction with respect to the defined watch-points as well as traceability to source code can be done.

[1]https://perf.wiki.kernel.org
[2]https://man7.org/linux/man-pages/man1/perf-script.1.html

```
400990
400992
400995
400996
400999
40099d
40099e
40099f
...
```

Listing 2.5: Decoded perf trace

A limitation of Intel PT is that it produces huge amounts of trace data (hundreds of megabytes per second per core [68]) which takes a long time to decode. Another limitation is the performance impact of tracing, something that will vary depending on the use-case and architecture.

2.5.3 *ARM core sight tracing facility*

To debug and trace software that runs on ARM-based SoCs for multicore processors, a set of ARM CoreSight technology [17] tools is used. To observe or modify the state of parts of the design different features may be used. For instance, the tracing feature is used to continuously collect the system information for later off-line analysis. It involves a separate trace generation component, and in case of multicores, a component for generating trace information can be assigned to each core.

The basic ARM CoreSight tracing functionality consists of different components such as Embedded Trace Macrocell (ETM), Program Trace Macrocell (PTM), System trace Macrocell (STM), Embedded Trace Buffer (ETB), Instrumentation Trace Macrocell (ITM), Trace Port Interface Unit (TPIU), Trace Memory Controller, configured as Embedded Trace Router (TMC-ETR), Trace Memory Controller, configured as Embedded Trace FIFO (TMC-ETF), and Cross Trigger Interface (CTI) [19].

The Program Flow Trace (PFT) is used to acquire trace data of the operations executed by the ARM processors. As presented in [103], the "PFT identifies concerned instructions in the program, and events as waypoints. A waypoint is a point where instruction execution by the processor might involve a change in the program flow." The PFT waypoints include: *all indirect branches, conditional and unconditional direct branches, all exceptions, any instruction that changes the instruction set state or security state of the processor, and synchronization primitives.* A trace macrocell that implements the PFT architecture is called a Program Trace Macrocell (PTM). The PTM trace module provides instruction tracing of ARM processors including the current ARM processor (Cortex M, R and A) [103]. The main features of PTM include trace generation, and triggering/filtering facilities that can be used to control tracing.

The tracer modules (PTM/ETM) of the Coresight framework[3] can be used in two ways [17–19]: 1) using the *perf* framework's Performance Monitoring Unit (PMU) abstraction (the *perf* command line tools); 2)interacting directly with the Coresight

[3]https://01.org/linuxgraphics/gfx-docs/drm/trace/coresight/coresight.html#acronyms-and-classification

devices using the sysFS interface (on Linux). The *perf* command line tools are used as described in the previous Subsection 2.5.2. Using the sysFS interface requires different commands to control the trace capture functionality on ARM-based processors.

Similarly to Intel PT, there is a problem in capturing traces in the sense that few seconds of operation yield trillions of cycles of execution. Therefore, to make sense of this volume of information would be extremely difficult. In addition, current cores can potentially perform one or more 64-bit cache accesses per cycle, and consequently, to record both the data address and data values can require a large bandwidth. Therefore, it makes more sense to only record data addresses, and PFT only traces execution at waypoints in order to filter events with a certain reconstructed addresses.

2.6 Coloured Petri Nets (CPNs) Models

Coverage analysis is normally conducted on programming language. To convey the same analysis on a model, it is necessary to consider models which have a given programming language for defining conditionals and functions, declaring variables, and writing inscriptions in the model. We considered Coloured Petri Nets (CPNs) models [83] and used CPN Tools [23, 82] for coverage analysis. CPNs is a graphical language for modeling and validating concurrent systems where concurrency, communication, and synchronization plays a major role. The main advantage of CPNs model is to combines Petri Nets [131] and the functional programming language CPN Meta-Language(ML) which is based on Standard ML [149]. Petri Nets provides the foundation of the graphical notation and the primitives for modeling concurrency and communication while Standard ML is used for modeling data. A CPN model of a system represents both the states of the system and the transitions causing state changes of the system.

CPN models are constructed and analysed using CPN Tools which has been widely used for modeling and verifying models of complex distributed systems. CPNs has been widely used in different areas such as model-based testing [102, 153], data networks [25], distributed algorithms [130], communication protocols [26, 133], coverage analysis [101], test cases generation [154] and embedded systems [1].

The structure of a CPN model consists of places (in the form of ellipses or circles), transitions (drawn as rectangular boxes), a number of directed arcs connecting places and transitions. In addition it contains nets inscriptions (text) written in Standard ML and can be located next to the places, transitions, and arcs. It is possible also to write auxiliary text in the net or in an associated function. The CPN model may be organised into a set of modules and sub-modules. A module of a CPN model can have substitution transitions (drawn as rectangular boxes with double lines). The substitution transition is called a sub-module [84] and hierarchically sub-modules are associated to a module in a CPN model. The name of the sub-module is indicated in a name-tag next to its associated substitution transitions.

One of the sub-modules contained in the Paxos CPN model is the initial Proposer which is shown in Figure 2.2 and it is associated to the *InitProposer* substitution transition. It initializes Proposers to obtain a new leader, and receive a client request for consensus. Then, the value of the current round number of the leader and the value of the received client request will be presented on the port places as tokens, respectively [154].

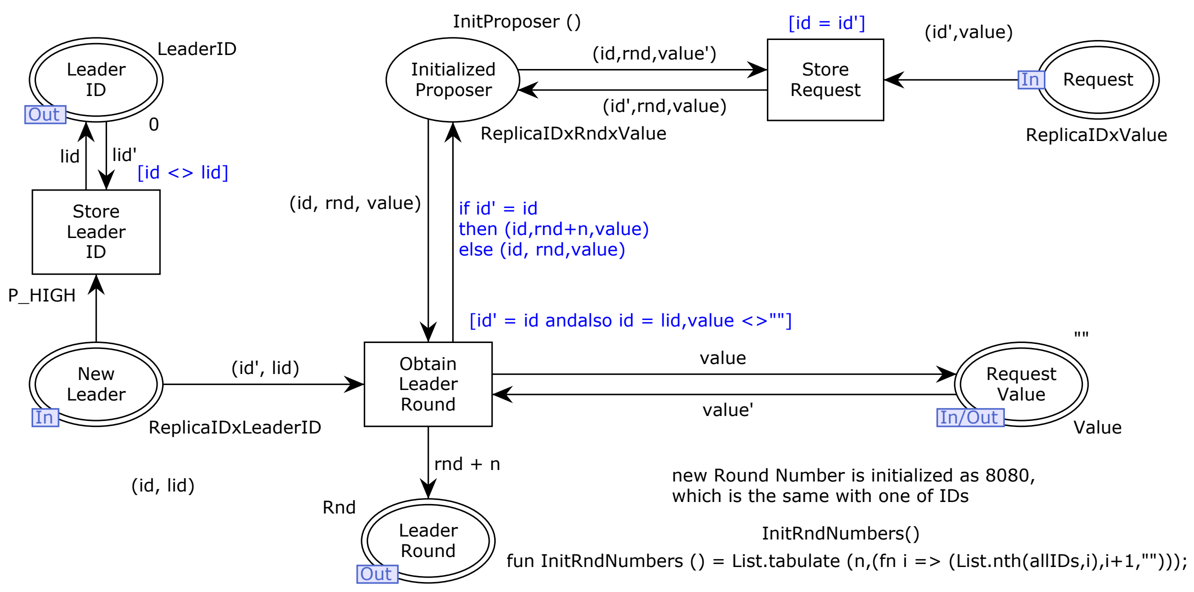


Fig. 2.2: Initial Proposer sub-module for Paxos CPN model

When considering CPN models under test, we do not generally know the requirements underlying the construction of the model. Furthermore, we cannot assume the explicit presence of test cases as they will only be given implicitly via the behaviour of the model and its initial marking (state). In the terminology of coverage analysis, we will therefore be concerned with structural coverage analysis of guard- and arc expressions. For these expressions, the test cases will arise as the transition bindings (occurrence modes) in which these expressions happens to be evaluated during a simulation or a state space exploration of the model.

A guard expression is a list of Boolean expressions all of which are required to evaluate to true in a given transition binding for the transition to be enabled. We refer to such Boolean expressions as *decisions*. Similarly, an if-then-else expression on an arc will have a decision determining whether the then- or the else-branch will be taken. Decision are in turn constructed from *conditions* and Boolean operators according to the definitions 1 and 2. For example, in Figure 2.2 we have decisions both in guards and arcs marked in blue and we are interested in their coverage based on MC/DC criterion.

Two main techniques can be used for the dynamic analysis of CPNs and CPN Tools. It is possible to explore behaviors of the modeled system using simulation-based analysis or verify the behavioral properties using state space exploration (SSE) methods and model checking. Simulation-based analysis with CPNs and CPN Tools aims at debugging and investigating the system design. The simulation of a CPN model can be done in two ways using CPN Tools: 1) An interactive simulation where a CPN model is executed step by step (similar to single-step debugging) and this allows to run single instances of the different parts of the model; 2)An automatic simulation where a CPN model is run once in the same way as a program execution. Through the state-space exploration methods the system model properties such as boundedness, reachability, liveness, and fairness can be verified. This allows to investigate verification questions related to the behaviors of the system model. The SSE can be performed fully automatically and it computes all reachable states and states changes (caused by firing transitions) of the CPN model and represent them as a directed graph. In the resulting directed graph, the nodes represent states and the arcs represent occurring transitions. This means that the main focus is on structural elements such as places, tokens, markings (states), transitions and transition bindings. Arc expressions and guards are only implicitly considered via the evaluation of these net inscriptions taking place as part of the computation of transition enabling and occurrence during model execution. This means that design errors in net inscriptions may not be detected as we do not obtain explicit information on for instance whether both branches of an if-then-else expression on an arc have been covered. In general, CPN models may suffer from the state space explosion problem, especially if the CPN model is too complex.

2.7 Binary Decision Diagrams (BDDs)

Binary Decision Diagram (BDD) is a representation of a Boolean function defined over binary values 0 (*false*) and 1 (*true*) [31]. BDDs are often substantially more compact than traditional normal forms such as disjunctive and conjunctive normal forms (DNF/CNF) and they can be manipulated very efficiently. BDDs are widely used in different application areas such as symbolic simulation, verification of combinational logic, and verification of finite-state concurrent systems. BDDs find their primary usage in CAD applications for digital hardware [112]. Combining a huge number of chips had led to a combinatorial explosion problems which makes it even harder to represent and manipulate functional behaviors of these chips within a computer. BDDs have provided a better connection in the context of compactness of the data structures and efficient algorithms to solve this problem. This has led to more performance improvements and breakthroughs in many CAD projects. More applications of BDDs include symbolic model checking [44, 112], verification of combinational logic [76, 128], verification of finite-state concurrent systems [43, 45], sequential machine equivalence [76, 81], test pattern generation [114], graph reachability [52], timing verification [76], symbolic simulation [76] and logic synthesis and optimization [143]. BDDs also have application in other domains such as combinatorics and manipulating classes of combined Boolean algebraic expressions.

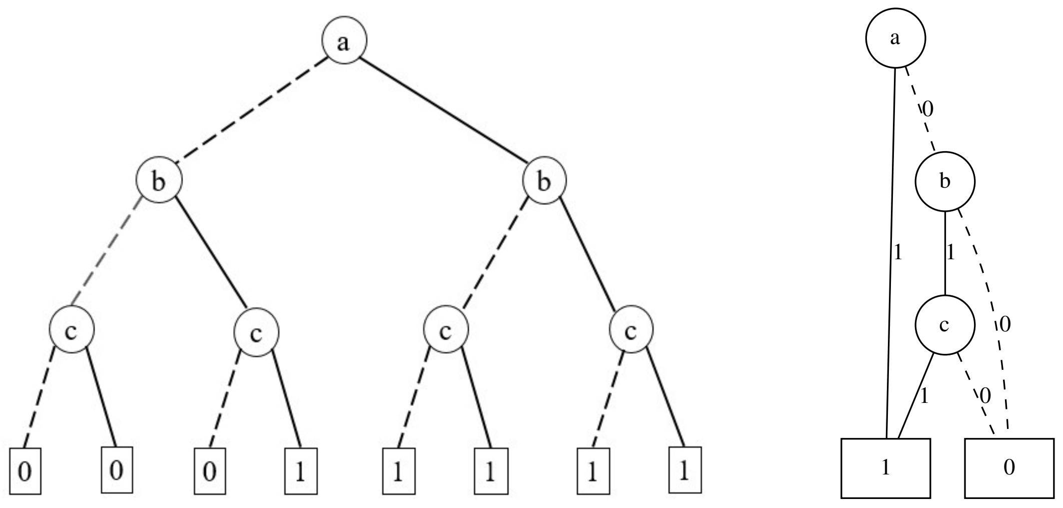
The structure of a BDD can be explained first from a *binary decision tree* (BDT) which is a rooted directed acyclic graph, with two types of nodes, terminal nodes and non-terminal nodes. Each terminal node is labeled by either 0 or 1, and every non-terminal node is labeled by a variable name *var(v)* and has two successors: a 0-successor and a 1-successor. A BDT is build based on a Boolean function consisting of Boolean variables concatenated by Boolean operators. It is equivalent to a decision or simply a formula made by variables combined in a disjunctive and/or conjunctive normal form.

Example 2. Consider an example of the BDT and BDD for the decision $D = a \mid\mid (b \; \&\& \; c)$ in Figure 2.3. The *a, b,* and *c* represent Boolean variables (conditions), which can be assigned 0 or 1. A dashed line represents false (0) input while a solid line represents true(1) input. The corresponding BDT is shown in Figure 2.3a which represents all possible combinations. One can decide whether a particular truth assignment to the variables makes the formula true or not by traversing the tree from the root node to the terminal node. For example, if a string "100" is assigned to the above variables, then f(1,0,0) will lead to a leaf vertex labeled 1. This is shown in Figure 2.3b and it makes a BDD to be more compact compared to BDT.

BDTs are essentially the same size as truth tables and they both do not provide a concise representation for Boolean function. For n variables, $2^n x(n+1)$ bits need to be stored in the memory and each entry of truth table need to be visited [44]. In addition, BDTs contain a lot of redundancy, isomorphic sub-trees can be merged to yield a concise representation called a BDD [31].

To obtain a canonical representation, two restrictions are placed on BDDs: 1) Variables must appear in the same order along each path from the root to a terminal node. That is a total ordering on the variables is required so that if any node u has a non-terminal successor v, then $var(u) < var(v)$. Different variable ordering methods have been explored and since they are not discussed herein in details, we refer the reader to the ones which give good results including *depth-first heuristic ordering* algorithm [63] (related variables are close together in the ordering) and *dynamic reordering* technique [134] (where the OBDD package internally reorders the variables periodically in order to reduce the total number of vertices in use). If we use the ordering $a < b < c$, for the function *f(a,b,c)*, we obtain the BDD in Figure 2.3b (a). 2) There should be no isomorphic sub-trees and redundant nodes in the diagram. This is achieved by applying three transformation rules repeatedly without altering the function represented by the diagram:

- *Remove duplicate terminals*: As shown in Figure 2.3b all duplicated terminals nodes are eliminated and a single 0-terminal node and a single 1-terminal node remain. All the arcs to the eliminated nodes are redirected to the two remaining terminal nodes.
- *Remove redundant tests*: If both outgoing edges of a node point to the same node, remove one edge and keep one incoming arc to that node.
- *Remove duplicate non-terminals (isomorphic subtrees)*: For any two non-terminals with the same label, with the same assignment and same successor, eliminate one of them and redirect all the incoming arcs to the other node.



(a) Binary decision tree (BDT) **(b)** BDD

Fig. 2.3: BDT and BDD for the decisions $D = a \,||\, (b \,\&\&\, c)$

The above rules are applied repeatedly until the size of the diagram can no longer be reduced. Different algorithms can be used to obtain a canonical form of a BDT. The obtained graph is called a *Reduced Ordered Binary Decision Diagram (roBDD)*. Note that in the rest of this paper, the term BDD refers to roBDD.

2.7.1 BDD algorithms

Today there is a number of algorithms [31] for providing BDDs with efficient graph representation. These algorithms include *reduce, apply, restrict, compose* and *satisfy* (which consists of *satisfy-One, satisfy-All* and *satisfy-Count*).

The *Reduce* algorithm transforms an arbitrary function graph into a reduced graph representing the same function. This is the algorithm used in Figure 2.3. Another generic form of manipulating BDDs using the Reduce algorithm can be found in [113].

The *Apply* algorithm is another procedure for BDDs and is used to implement operations on Boolean functions such as $\wedge, \vee, \oplus$ and complementary $(f \oplus 1)$. It takes as input two graphs representing functions f_1 and f_2 and a binary operator $< op >$ and produces a reduced graph representing the function $f_1 < op > f_2$. Given two roBDDs representing f_1 and f_2 , *apply*$(< op >, f_1, f_2)$ is defined as $[f_1 < op > f_2](x_1, ..., x_n)$=$f_1(x_1, ..., x_n) < op > f_2(x_1, ..., x_n)$. If the result contains other redundant nodes, it can usually be reduced to make it into an BDD using *reduce* algorithm.

The *Restrict* algorithm finds a BDD for a Boolean function *f*$(x_1, x_2, ..., x_n)$ with a restriction to one of the variables to 0 or 1 wich results in *f*$(x_1, x_2, ..., 1, ..., x_n)$ or *f*$(x_1, x_2, ..., 0, ..., x_n)$. Calling *restrict* $(0, x, f)$ for each node *n* labelled with *x*, incoming edges are redirected to *low(n)* and *n* is removed. Then we call *reduce* on the resulting BDD. The call *restrict* $(1, x, f)$ proceeds similarly, only we now redirect incoming edges to $hi(n)$.

The *Compose* algorithm constructs a graph for the function obtained by composing two functions. For example $f(x_1, x_2, x_5)$ composed with $g(x_3, x_4)$ at position of x_2 results in the function $f(x_1, g(x_3, x_4), x_5)$. Composition can also be expressed in terms of restriction and Boolean operations, according to Shannon expansion [31].

The *Satisfy* algorithm consists of satisfying a set of elements S_f for a function *f*$(x_1, x_2, ..., x_n)$. S_f can include the number of elements, a listing of the elements, or for instance just a single element denoting in fact the set of all truth assignments for variables $\{x_1, x_2, ..., x_n\}$. In other words, we consider operations to examine the set of satisfying truth assignments of a node u. The procedure includes *satisfy-one*, *satisfy-count* and *satisfy-all* [14].

Satisfy-one is normally called with the root of the graph and an *array x* initialized to some arbitrary pattern of 0's and 1's. It returns the value *false* if the function is unsatisfiable ($S_f = \varnothing$), and the value *true* if it is satisfiable.

Satisfy-count determines the number of valid truth assignments. *Satisfy-all* returns all satisfying truth-assignments. For example if, the variables are assumed to be ordered as : $x_1, x_2, ..., x_n$, then *satisfy-all* finds all paths from a node u to the terminal 1. It is an exhaustive search of the graph which prints out the element corresponding to the current path every time we reach a terminal vertex with value 1.

2.7.2 *BDD libraries*

Different BDD libraries/packages [55, 56, 142] can be used to manipulate BDDs and provide interfaces to programming languages such as C, C++ and Python. Our analysis for test cases generation based on BDDs uses the Pyeda library [56] and implemented in Python [4]. PyEDA is a Python library primarily used for electronic design automation (EDA). PyEDA provides both a high level interface to the representation of Boolean functions, and fast C extensions for fundamental algorithms where performance is essential. There are several ways to represent a Boolean function with PyEDA and different data structures have different tradeoffs. PyEDA's API for logic expressions, truth tables, and binary decision diagrams is a mature implementation. The PyEDA's repository[4] contains detailed instructions for its installation and its documentation[5] provides guidance of how to use PyEDA [56].

Different commands are used via an interactive terminal for manipulating Boolean functions, creating and transforming logic expressions, truth tables, and binary decision diagrams. Conditions are the literals elements defined as symbolic variables and different algebraic operators such as *Not, Or, And, Xor, Equal, Implies,* and *ITE (if-then-else)* can be used for writing the functions. Different function's basic properties such *simplification, satisfiability,* and *equivalence* can be explored as shown in Listing 2.6.

```
>>> from pyeda.inter import *
>>> a, b, c = map(exprvar, 'abc')
>>> F = a | (b & c)
>>> F.simplify()
Or(a, And(b, c))
>>> F.satisfy_one()
{c: 1, b: 1, a: 0}
>>> list(F.satisfy_all())
[{c: 1, b: 1, a: 0}, {a: 1}]
```

Listing 2.6: Function properties from PyEDA

```
>>> from pyeda.inter import *
>>> from graphviz import Source
>>> a, b, c= map(bddvar, 'abc')
>>> F = a | (b & c)
>>> gv = Source(F.to_dot())
>>> gv.render('BDDfigure.pdf', view=True)
```

Listing 2.7: BDD construction using PyEDA

To construct and visualize a BDD, the graph structure is converted to DOT format for consumption by Graphviz[6]. For example, Figure 2.3b is obtained from the Graphviz output on the majority function in three variables as shown in Listing 2.7.

As far as MC/DC test case generation is concerned, the BDD representation gives us an advantage in building fresh pairs: by exploring the tree from the root, the ordered labels tell us when we can preempt a search because the condition of interest does not

[4]https://github.com/cjdrake/pyeda.git
[5]http://pyeda.rtfd.org
[6]https://graphviz.org/

exist in the remaining subtree, and we can continue our search in a sibling. Compared to an exploration of the corresponding truth-table, this effectively allows us to skip over irrelevant rows.

CHAPTER 3

MC/DC ANALYSIS AND MEASUREMENT

This chapter summarizes our approach for MC/DC analysis and measurement based on the program traces recorded using IntelPT. Our coverage analysis is without instrumentation since the instrumentation is intrusive and it is not recommended for testing and coverage analysis of safety critical systems [124]. Compared to the overhead of intrusive software instrumentation which has to be removed before release, with our low overhead approach, we measure coverage directly on the release code based on the object code level which also complies with the position of CAST-17 [38]. In this chapter, we present an overview on the static analysis, tracing, MC/DC measurement and state-of-the art through the related work to our approach of MC/DC measurement based on the traces. The details of our findings are presented in our paper [2] included in Part II of this thesis.

3.1 Non-intrusive MC/DC Measurement Based on the Traces

Figure 3.1 shows an overview of our approach and the implementation of MC/DC measurement. We start with the static analysis of the source code in order to detect the decisions and their conditions. In addition, we extract all information about their corresponding conditional jumps performed by the CPU from the object code. This

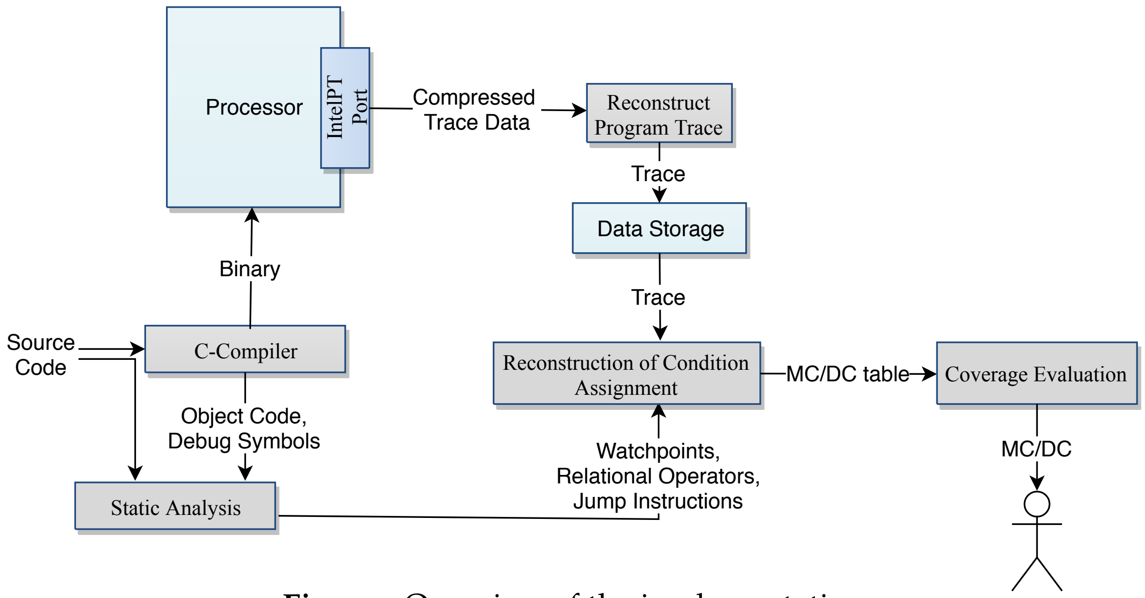

Processor
IntelPT Port
Compressed Trace Data
Reconstruct Program Trace
Trace
Data Storage
Trace
Binary
Source Code
C-Compiler
Reconstruction of Condition Assignment
MC/DC table
Coverage Evaluation
Object Code, Debug Symbols
Watchpoints, Relational Operators, Jump Instructions
MC/DC
Static Analysis

Fig. 3.1: Overview of the implementation.

mapping is necessary in order to demonstrate the MC/DC traceability from object code to source code [35, 38, 41]. We record and decode traces using the Intel PT facility. Then we combine this information and the program trace provided by Intel PT in order to reconstruct the conditions assignments. The resulting information is filled in a table where each row captures a single decision with the truth values of all conditions in a fixed order and the outcome. We conduct a coverage evaluation and measure MC/DC by checking the independence effect of each condition.

3.1.1 *Static analysis of source code and object code*

As defined in Section 2.1, decisions in the source code refer to all logic structures not only the control statements that result in branch points. These include:

– If statements

– Loops such as while-, do- and for-statements

– Switch-statements and

– Assignment statements containing Boolean expressions.

The main purpose of static analysis is to find these decisions and conditions in the source code and map them to their conditional jumps in the object code. This can be achieved by accessing the Abstract Syntax Tree (AST) provided by Clang [148], which contains information about the statements, their location in the source code, and how the statements are assembled.

```
-IfStmt <line:2:3, line:7:3> has_else
|-BinaryOperator <line:2:7, col:24> 'int' '||'
| |-ImplicitCastExpr <col:7> 'int' <LValueToRValue>
| | -DeclRefExpr <col:7> 'int' lvalue ParmVar 0x55be4094a400 'a' 'int'
|-ParenExpr <col:12, col:24> 'int'
|  -BinaryOperator <col:13, col:23> 'int' '&&'
|    |-BinaryOperator <col:13, col:16> 'int' '=='
|    | |-ImplicitCastExpr <col:13> 'int' <LValueToRValue>
|    | | -DeclRefExpr <col:13>'int' lvalue ParmVar 0x55be4094a480 'b' 'int'

|    | -IntegerLiteral <col:16> 'int' 5
|   -BinaryOperator <col:21, col:23> 'int' '>'
|      |-ImplicitCastExpr <col:21> 'int' <LValueToRValue>
|      | -DeclRefExpr <col:21>'int' lvalue ParmVar 0x55be4094a500 'c' 'int'

|      -IntegerLiteral <col:23> 'int' 3
|-CompoundStmt <col:26, line:4:3>
| -ReturnStmt <line:3:4, col:11>
|   -IntegerLiteral <col:11> 'int' 1
'-CompoundStmt <line:5:7, line:7:3>
-ReturnStmt <line:6:4, col:11>
-IntegerLiteral <col:11> 'int' 0
```

Listing 3.1: Abstract Syntax Tree generated with Clang-14.0 for Listing 2.2

One of the ways to access the AST is to use the *LibTooling* which is a C++ interface. *LibTooling* can be used along side with the *AST-matcher* [148] and simplifies the specification of patterns that the AST is supposed to match.

For example the if-statement (`"if (a || (b==5 && c>3))"`) in Listing 2.2 results in the `"IfStmt"` parent node in Listing 3.1 representing the AST, and condition names, binary operators and their locations (line and column number) are in its child-nodes. The binary operators help in the reconstruction of the assignments based on the executed program jumps. Based on *LibTooling* and the *AST-matcher* [148], our tooling detects all the decisions in the form of traditional branch points (`while-`, `if-` and `for`-statements).

In our object code analysis we use debug-symbols provided by the compiler to map the conditions detected from source code to conditional jumps in the object code. We use *clang* because this compiler provides rich debug symbols containing line and column information with the compiler option : `"-g -XClang -dwarf-column-info"`. Combined with the detected decisions from the LLVM tool, we then can detect all conditional jumps that are needed for measuring MC/DC based on traces. Because the outcome of the decision during the execution has to be reconstructed as well, it is necessary to find the *then-statement* which is the statement executed in case of a decision being evaluated as **true**. This statement is also mapped using debug-symbols to its corresponding instruction in the object code. Table 3.1 shows the interpretation of detected jumps with respect to Listing 2.3 with the if-statement (`"if (a || (b==5 && c>3))"`). The result are the decisions, conditions and then-statements in the source code and their translation in the object code. Each row represents which condition being considered, the corresponding relational operator and the jump instruction as well as its interpretation.

3.1.2 *Program traces and MC/DC evaluation*

As shown in Figure 3.1 the trace can be gathered through the trace port of the processor that is executing the program and can be saved in a data storage after the reconstruction process. We use IntelPT through perf [158] to record and decode the trace. With *perf*, the Linux-kernel provides an easy-to-use implementation of the recording and reconstruction of Intel PT traces. However, it contains a bottleneck, because the trace data has to be stored in some form of storage space. The data rate of the trace information is high (it fills up the hard drive quickly) that even very high speed storage can only record a few seconds of program execution. The reconstructed traces become quiet large even for short execution times. To reduce the size, we filter the trace against

Condition:	Relational Operator:	Jumps Instruction:	Jump Interpretation:
a	none	`jne`	**True**
$b == 5$	$==$	`jne`	**False**
$c > 3$	$>$	`jle`	**False**

Table 3.1: Interpretation of detected jumps from Listing 2.3: (`"if (a ||(b==5&&c>3)"`)

the watch-points and only store those parts of the trace that are relevant for measuring MC/DC.

After the assignments of the conditions are reconstructed from the trace, the MC/DC table can be filled and MC/DC can be measured iteratively. The tool chain of detecting all decisions, mapping conditions to conditional jumps, running and tracing the program and measuring MC/DC based on the trace can be accessed via a graphical user interface (GUI) or through the command line [2]. MC/DC is evaluated by calculating the ratio of the number of conditions with independence effect and the total number of conditions in decision. That is, we report the percentage of covered conditions to the total number of conditions. The uncovered condition are reported such that the developer can investigate the reason why they are not covered and possibly add additional test cases that may cover them.

In summary, we measure MC/DC based on program traces without any instrumentation. We show the possibility of measuring coverage directly on the release code as recommended for safety critical systems [124], by only using debug symbols that are not altering the behavior of the code and therefore are not considered intrusive. Our approach is tested on C programs and complies with the position of CAST-17, that provides certification authorities' concerns and position regarding the analysis of structural coverage at the object code level [38]. More about our finding for non-intrusive MC/DC measurement based on the traces can be found in [2] attached in Part II.

3.2 Related Work on MC/DC Measurement

The applicability of MC/DC to software testing for safety-critical systems have been introduced by Chilenski in [42]. Different comparisons for code coverage metrics have been investigated in the context of structure based metrics [118], data-flow metrics [45], decision coverage and MC/DC [91], comparison of multiple condition coverage (MCC) and MC/DC with short-circuit evaluation [89]. MC/DC and object branch coverage (OBC) criteria were compared in [32]. Even though the above papers give a basic foundation for MC/DC, none of them analyses MC/DC based on program traces. They are concerned with explaining and defining MC/DC, and the approach of measuring MC/DC without instrumentation is not explored.

A non-intrusive online monitoring for multi-core systems based on the embedded trace (ET) of the system under test is proposed in [50]. ET is a promising technology to observe the system under test and it allows to record a set of real-time streams of software execution data emitted by a processor non-intrusively. ET provides the means to investigate the system behavior for monitoring software in execution [126]. Online reconstruction and analysis of debug trace data are based on FPGA and TeSSLa [49]. The tracing approach used in these papers attracts our attention, and in our investigation for MC/DC measurement, our analysis is based on the traces.

Different testing tools for measuring coverage were developed for both industrial use and academic purpose. For instance, a survey conducted in [162] describes and compares 17 tools primarily focusing on, but not restricted to, coverage measurement. These tools are focusing on weaker coverage criteria for C, C++ and Java programs. Most of them are used only for code coverage, but some, such as Agitar, Dynamic, JCover, Jtest and Semantic Designs, provide debugging assistance as well.

Testing tools which focus on MC/DC measurement include: *Vector-CAST/MCDC* [151] which measures MC/DC coverage for C/C++. The tool supports both unique cause and masking MC/DC analysis. Beside reporting and documenting the results, the tool supports automatic test case generation to efficiently support the development of a full set of MC/DC test cases. Parasoft C++test [150] is a C/C++ testing tool that is capable of measuring MC/DC. MC/DC is measured as a percentage of the number covered conditions over the total number of conditions in all decisions. Testwell CTC++ tool [146] measures line, statement, function, decision, multiple condition, MC/DC and condition coverage for C, C++, Java, and C# on target and on host. The generated report is showing coverage percentage. CodeCover [46] is an open-source, white-box testing tool developed at the University of Stuttgart. It implements the Ludewig term coverage and they claim that it is similar to MC/DC (subsumes MC/DC). RapiCover [127] analyzes code coverage including MC/DC with low instrumentation overheads. The tool supports MC/DC analysis of decisions with up to 30 conditions. All these tools measure MC/DC intrusively by instrumenting the source code.

In [166], SmartUnit tool which supports statement, branch, boundary value and MC/DC coverage is described. They aim at the unit coverage-based testing and automatically generating MC/DC coverage test cases in an industrial environment. nditions and the total conditions in the source code. The commercial Lauterbach tool [98] uses a dedicated hardware-interface to transfer tracing data from the system-under-test into the developer's machine for analysis. They support a variety of trace sources, among others also Intel PT, and use it to measure MC/DC.

Another alternative, non-intrusive approach is running the system-under-test within an emulator. The QEMU emulator has been used to this end within the Adacore community [29], and in the RTEMS operating system [60]. Through the emulator it is easy to observe the execution of a program on the object code level, very much like through Intel PT. The obvious threat to validity is of course how closely the emulator setup can reflect the real system, especially when considering certification.

Lauterbach offers the t32cast command line tool for MC/DC coverage in TRACE32 based on a real- time trace recording, which analyzes the C/C++ source code [98]. Here, the user must ensure that the selected compiler translates each condition in the source code into a conditional jump at the object code level, e.g. by disabling optimizations. We provide a novel non-intrusive approach for MC/DC measurement based on traces recorded with IntelPT. Our tool does not require instrumentation and recompilation of software under-test. Similarly to Lauterbach, optimization needs to be off. To the best of our knowledge there are no other developed tool analyzing MC/DC coverage based on the trace generated with IntelPT with full source code to object code traceability.

CHAPTER 4

DATA RACE MONITORING IN CONCURRENT PROGRAMS

In this chapter, we summarize our approach for data race detection in concurrent programs. This is not directly related to MC/DC analysis, but we inherit our motivation through analyzing coverage without instrumentation, and our aim is to monitor data races with lower overhead instrumentation. This is in the context of COEMS project where we had to investigate different methods for data race detection in multi-core embedded systems. We summarize our hardware-assisted data race detection as presented in [7] and included in Part II of this thesis. Additionally, we discuss the state-of-the art on data race detection in the form of related work.

4.1 Hardware-assisted Data Race Detection

Referring to the definition of data races in Chapter 2, Section 2.3, a data race may occur when there is a conflicting access on a shared resource and one of the operations is a read/write or write/write. In a read/write race, one thread reads data, which is then subsequently overwritten by another thread, leading the first thread to proceed under the assumption that it has the current value. In a write/write race, another thread overwrites a previously written value, leading to a similar problem. We present our approach on how to detect data races using the COEMS tracing technology through continuous online monitoring with low-impact instrumentation on a novel FPGA-based external platform for embedded multi-core systems.

4.1.1 *Overview on data race detection in COEMS*

We have developed a *non-intrusive* approach to monitoring applications on embedded system-on-chips (SoCs) for data races using the COEMS platform [49] which aims to eliminate the overhead of dynamic checking by offloading it to external hardware[1]. The platform offers control-flow reconstruction from processor-traces (here: the Arm CoreSight control-flow trace), and data-traces through explicit instrumentation [125]. Race checking is executed on an FPGA on a separate hardware-platform to minimize impact on the system under observation.

[1]The EU Horizon 2020 project "COEMS–Continuous Observation of Embedded Multicore Systems", https://www.coems.eu

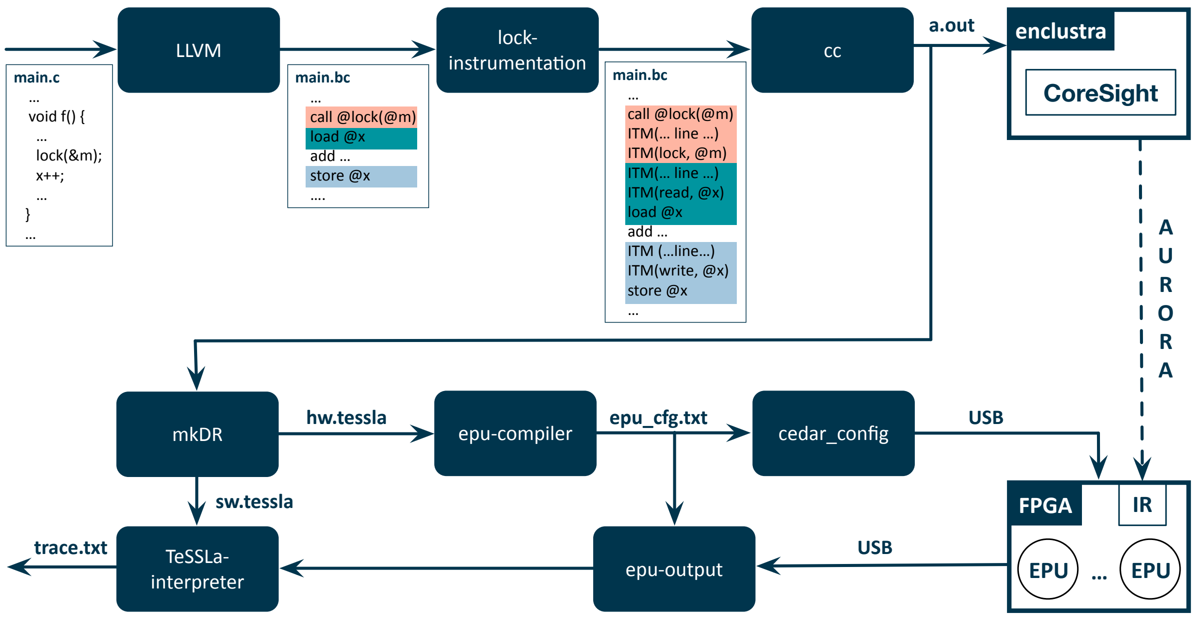

LLVM
lock-instrumentation
cc
a.out
enclustra
CoreSight
main.c
... void f() { ... lock(&m); x++; ... } ...
main.bc
... call @lock(@m) load @x add ... store @x ....
main.bc
... call @lock(@m) ITM(... line ...) ITM(lock, @m) ITM(... line ...) ITM(read, @x) load @x add ... ITM (...line...) ITM(write, @x) store @x ...
AURORA
mkDR
hw.tessla
epu-compiler
epu_cfg.txt
cedar_config
USB
sw.tessla
FPGA
IR
trace.txt
TeSSLa-interpreter
epu-output
USB
EPU ... EPU

Fig. 4.1: Lock instrumentation and race monitoring using the COEMS technology

The COEMS FPGA requires a compiled monitor-configuration. As this configuration needs to be generated for a specific binary under test, we present our approach where we instantiate a template that monitors a fixed number of memory locations for consistent access through a fixed number of locks. Although these numbers need to be determined before starting the monitoring, the flexibility of TeSSLa allows us to also deal with an unbounded number of threads, and limited monitoring of dynamically allocated memory and locks. Additionally, our instrumentation supports recording traces in files, and offline analysis of execution traces with the TeSSLa interpreter only. This aids in quick prototyping of new specifications on vanilla developer machines without replicating a full setup of SoC and COEMS hardware.

4.1.2 *Data race detection workflow with COEMS tools.*

The COEMS infrastructure is used for online monitoring of multicore systems behaviours crucial for detecting non-deterministic failures. The infrastructure consists of COEMS FPGA enclosure, the Arm-based Enclustra SoC that serves as the SUT, and the AURORA interface connecting both. As soon as the program starts running on the Enclustra board, control flow messages are generated via the Arm CoreSight module and transmitted, together with user-specified data trace messages from any instrumentation, through the AURORA interface to the COEMS trace box. Figure 4.1 summarizes the key steps for instrumenting and monitoring data race conditions using the COEMS technology.

The workflow of instrumentation and data race monitoring is as follow:

1. Using the COEMS lock instrumentation tool [8], we instrument the SUT during compilation so that the executable emits information to the COEMS trace box at runtime. We insert calls to instrumentation (i) after taking a lock, (ii) before releasing a lock, and (iii) on shared memory accesses with the help of LLVM.

Then, we compile and link the instrumented LLVM intermediate code (.bc) into a binary file (a.out).

2. We copy the binary to the system under observation (enclustra) where we will later run it. The mkDR-script instantiates a TeSSLa specification template with the memory addresses and mutexes to be observed, based on the names of global variables.

3. The instantiated specification is then split into two halves, as its size exceeds the currently available number of eight EPUs on the prototype hardware. The first half hw.tessla filters the high event rate stream of observations on the FPGA. It is translated by the epu-compiler into a configuration file (epu_cfg.txt), and then uploaded to the FPGA by cedar_config. The second half sw.tessla receives the output of the first stage, and does the final processing on a stream that now has a lower event rate in the TeSSLa interpreter.

4. Then, we run the binary file, which will automatically start sending trace data to the FPGA. The epu-output tool decodes the FPGA output into a TeSSLa event stream. Note that the behaviour of the application is independent of whether the COEMS FPGA is actually connected or not. If not, trace data is silently discarded, but does not affect the timing of the application.

5. Finally, we analyse this trace with the second part of the TeSSLa specification (sw.tessla) with the TeSSLa interpreter, which will emit race warnings if necessary.

Our data race specification uses mostly data trace events, since we require the addresses of memory and locks, except for a control flow event when pthread_create is called and to signal termination of program under test. COEMS supports also offline (software-based) analysis of execution traces where the user only needs the TeSSLa interpreter and the COEMS lock instrumentation tool. In this case, the software TeSSLa interpreter will run the entire TeSSLa specification for detecting data races on a locally generated software trace-file without compiling the TeSSLa specification for the EPUs.

The lockset-based algorithm in TeSSLa tracks which set of locks is held at every memory access by the current thread and update the current value if necessary. The set of all memory locations and all lock identifiers need to be known before we configure the FPGA. For each pair of memory location X and lock identifier L, we create a boolean stream `protecting_X_with_L` that is initialised to *true*. If all these streams for a given X carry *false*, we know that no common lock is protecting the current access, and we emit a race warning on the `error_X` stream for that memory location.

As detailed in [7], we have shown how to use the COEMS technology for online monitoring of multicore systems, and contextualized it to check for potential data races in applications that use locks for synchronisation. Through the COEMS platform, developers can observe the control-flow in a digital twin of their SUT on an embedded systems without affecting the behaviour. Additional instrumentation of the application can send more detailed data at negligible cost.

We provided an outline on how the lockset-based Eraser algorithm can be encoded in the TeSSLa-specification language for a given application. This specification is then compiled onto the external COEMS FPGA and uses the data- and control flow trace emitted from the system under test to observe a specified set of locks and memory locations. As the full specification exceeds the capabilities (in terms of size) of the available prototype, we combine a hardware- and a software stage to report on potential races.

Our use of the high-level stream-based temporal specification language, TeSSLa [100], means that the reconfiguration of the monitor is substantially faster than synthesising VHDL (few seconds vs. dozens of minutes), and allows end-users to customize the race checker specification to their needs without being FPGA-experts. This is not possible with other specification-based approaches that directly aim to use the integrated FPGA of a SoC. These approaches do not offer the quick reconfiguration possible with the COEMS platform but require full time-consuming reconfiguration, and do not support the use of control-flow tracing due to the limited capacity of the SoC.

Our proposed approach is more flexible than a dedicated race checker implemented on the FPGA: to the best of our knowledge, such a general solution does not exist, though it is of course in principle possible. Our *hardware*-based approach can be used in safety-critical systems such as the aerospace and railway-domains where certification is necessary. In these domains, using a software inline race-checker such as ThreadSanitizer [137] is not possible as the tooling for instrumentation and online race-checking is not certified for those systems, if it even exists.

4.2 Related Work on Data Race Detection

From the literature review, data races have been approached from two sides: *static analyses* that checks the code before it runs, and *dynamic analyses* which looks at individual executions of a program. Both techniques in general rely on the availability of the source code, and in the case of dynamic analysis, the possibility of recompilation with additional instrumentation.

As dynamic analysis for data race detection needs to record historic behaviour during execution, they often interfere in terms of computation time and memory consumption. For example, the popular dynamic ThreadSanitizer integrated with the LLVM compiler toolchain slows down executions by a factor of 10 to 100, depending on the workload [137, 164]. This is one reason why dynamic analysers are traditionally only employed during development and testing, but not included on the production system [155].

The COEMS project developed a hardware-based solution, in which a *field-programmable gate array* (*FPGA*) checks the execution trace in parallel to the running system with minimal interference. The hardware is adapted for analysing events described in the stream-based specification language TeSSLa [100]. Using an intermediate specification language that is executable on the FPGA, one can avoid the time-consuming re-synthesisation of the FPGA when changing specifications.

We have ported the gist of the Eraser algorithm [135] to the subset of the TeSSLa language that is supported on the hardware. An alternative approach already used the TeSSLa-interpreter, but was not suitable for compilation onto an FPGA due to the dynamic data structures (sets and maps) that only the interpreter offers [80]. We adapt the software-based analysis, which relies on dynamic data structures such as sets and maps in the TeSSLa interpreter, to the hardware-specific implementation of the COEMS trace box and we allow monitoring of dynamically allocated memory and locks.

For dynamic race detection, the Eraser algorithm is limited but it can be used for checking locking discipline [135] using sets and assuming that the number of used memory locations and locks is statically known. We can statically derive the necessary streams and such static encoding is also possible for the modern FastTrack-algorithm by Flanagan and Freund [61]. The FastTrack-algorithm uses lightweight vector clocks and the happens-before relation which can be used to avoid false positives. However, our approach requires focusing on a fixed number of memory locations and locks, but can deal with an arbitrary number of threads. An observation-based race checker that tracks memory accesses and lock operations can also be implemented through the help of virtualisation. Gem5 [27, 78] is such a framework. Virtualisation means on the one hand that observation cannot be done on a deployed system in the field but only in the lab and with a limited number of supported peripherals. On the other hand, control-flow events can easily be explicitly generated, no expensive reconstruction is necessary. However, given the high event rate of observations on memory accesses, we expect a similar performance impact like the one reported for ThreadSanitizer.

Another prominent example where a high-level specification is synthesised into an FPGA is RTLola [22]. This differs from our approach in the following: the specification language puts a stronger emphasis on periodic data than we do with our discrete TeSSLa events. Furthermore, RTLola is synthesized via VHDL onto the FPGA, and hence has a high turn-around time for reconfiguration. Communication between the system-under-test and the verification logic is left open to the user and requires knowledge of VHDL, though of course in principle data events can then be emitted through instrumentation. In contrast to our solution, an RTLola specification cannot benefit from control-flow tracing, since control-flow reconstruction is not available as specification and hence cannot be compiled onto the FPGA. Furthermore, it would exceed the capacity of current SoCs both in terms of space and execution speed [125]. We leave performance evaluation of RTLola execution for race checking purposes on the FPGA to future work, but note that providing an API for the instrumentation to the monitor requires VHLD-knowledge.

A similar direct approach via hardware-synthesis has been taken for Signal Temporal Logic (STL) [79]. It would certainly be feasible to encode a race checker in STL, but that would not be playing to STL's strength in terms of timing properties (which are not relevant for race checking) and observing signals on a wire (as opposed to a programmable interface to send values from the instrumented code to the monitor).

The R2U2 [116] monitoring system for unmanned aerial vehicles provides a generic observation component on an SoC. Again, events must be explicitly emitted, and no control-flow reconstruction is available. Similar to our approach, and unlike in RTLola, this component is generic and is parametrised by compiled specifications. R2U2 uses Metric Temporal Logic specifications (MTL), which are very suitable to describe, e.g. timing properties. While it is certainly possible to specify our race checker in MTL, we leave it to future experimental evaluation to determine how many instances of the race pattern (in terms of memory location/protecting lock) would be feasible, and how the communication bus would uphold under varying event rates.

A theory has only the alternative
of being wrong right or wrong.
A model has a third possibility:
it may be right but irrelevant

—Manfred Eigen [107]

CHAPTER 5 COVERAGE ANALYSIS ON DESIGN LEVEL MODELS

To evaluate our approach for MC/DC measurement, we explore its applicability on software design level models. We focus on models that contain inscriptions with conditionals such that we can check if they contributed as expected to the outcome of each decision.

We considered coloured Petri Nets (CPNs) models as they have the inscriptions (e.g., arc expressions and guards) specified using Standard ML (SML) which may contain conditions. Traditionally these conditions are not evaluated against any coverage criterion during the simulation and state space exploration (SSE). Simulation and SSE are only used for validating CPN models for checking the behavioral properties related to net structure, i.e., places and transitions. There is no coverage information shown in the SSE report generated by CPN Tools. Therefore, we apply our coverage analysis based MC/DC criterion and check how the conditions contained in CPN models are covered [5, 10].

5.1 MC/DC Measurement of Net Inscriptions in CPN Models

Coverage analysis is important for programs in relation to reveal faults in the program. Coverage analysis can be useful as well for models as a means to show if some parts of the models (net structure parts and net inscriptions) were not exercised. We apply MC/DC and BC on CPN models as a potential candidate that contains conditionals and was widely used in coverage analysis [101]. In CPN models [83] and CPN Tools [23, 82], the inscriptions (e.g., arc expressions and guards) are specified using Standard ML (SML). The application of simulation and state space exploration (SSE) for validating CPN models traditionally focuses on behavioural properties related to net structure, such as places, transitions and bindings. This means that the net inscriptions are only implicitly validated, and the extent to which these have been covered is not made explicit. Our goal is to establish a link between coverage analysis known from programming languages and net inscriptions of CPN models where we apply MC/DC which generalizes branch coverage of SML decisions.

Coverage analysis for software can be provided through dedicated instrumentation of the software under test, either by the compiler, or additional tooling, such as binary instrumentation. Transferring this to a CPN model under test, our aim is to combine the execution of a CPN model (by simulation or SSE) with coverage analysis of SML guard and arc expressions. To record coverage data for a CPN model under test, it is necessary to instrument the Boolean expressions such that the truth-values of individual conditions are logged in addition to the overall outcome of the decision. Our approach to instrumentation makes use of side-effects by outputting intermediate results of conditions and decisions, which we then process to obtain the coverage verdict. No modifications to the net structure of the CPN model are necessary. Furthermore, the instrumentation has little impact on model execution so that it does not delay the simulation and SSE.

5.1.1 *CPN models under test*

We apply MC/DC measurement of net inscriptions to eleven larger public-available CPN models. These models include: a model of the Paxos distributed-consensus algorithm [154], a model of the MQTT publish-subscribe protocol [133], three models for distributed constraint satisfaction problem (DisCSP): weak-commitment search (WCS), asynchronous backtracking (ABT) and synchronous backtracking (SBT) algorithms [120], a complex model of the runtime environment of an actor-based model (CPNABS) [70], a reactor control system for a nuclear power plant (RCS-NPP) model and Niki T34 Syringe driver model [33]. In addition, we have tested four CPN models for test case generation from natural language requirements (NatCPN) [138]: nuclear power plant (NPP) model, turn indicator system (TIS) model, priority command (PC) model and vending machine (VM) model. All models come with initial markings that allow state space generation, in the case of MQTT, T34PIM and DisCSP are complete, and are incomplete in the case of Paxos, NatCPN and CPNABS.

To choose the above CPN models we considered three main criteria: 1) the CPN models are freely public-available ; 2) the model contains at least one non-trivial decision (decision composed with more than one condition); 3) the model is correct in the sense that both simulation and state space exploration are feasible.

5.1.2 *Experimental setup and results*

Figure 5.1 gives an overview of our experimental setup. Initially, we have the original CPN model under test that is to be evaluated for coverage analysis. The first step is to instrument the original CPN model under test by transforming guard and arc expressions into a form that prints how conditions were evaluated and the overall outcome of the decision [10]. Our instrumentation is almost 99% automated and does not affect the functionality of the original CPN model.

Second, we run the SSE on the instrumented model. Next, we reconfigure the initial markings with any additional initial configurations if they are obvious from the model. As the side effect of SSE, we run the MC/DC generation which gives as output a log file containing the information of evaluations of conditions in arc expressions and guards and the decision outcome. Then, we run the MC/DC analyser that determines whether each decision is MC/DC-covered or not.

Furthermore, we visualize the coverage information in the CPN models taking as input on the original CPN model and the output of how conditions and decisions are MC/DC evaluated. This result in a coloured CPN model where the covered parts are

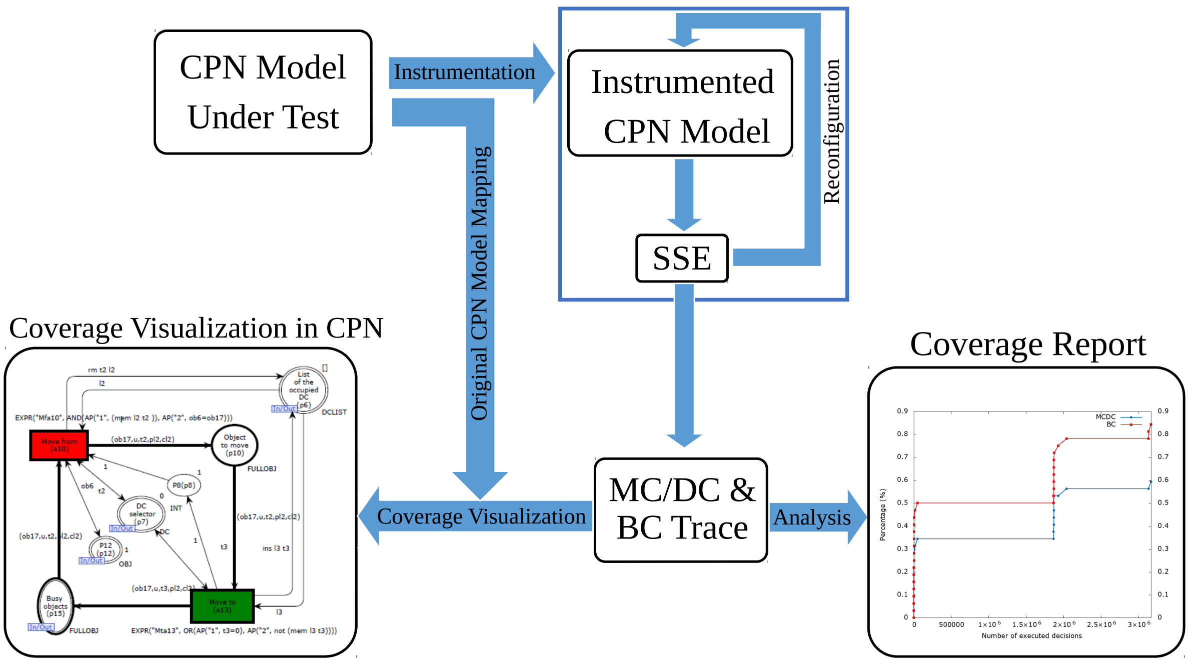


Fig. 5.1: Experimental setup for Coverage analysis for CPN models

annotated in green and the uncovered parts presented in red. Figure 5.2 shows a CPN model structure presented in Figure 2.2 after coverage analysis where covered and uncovered parts are highlighted. Table 5.1 presents the summary of the percentage of how much the tested CPN models are MC/DC and BC covered. For each model, we consider the number of executed decisions (second column) in arcs and guards. Column *"Model decisions"* refers to the number of decisions that have been instrumented in the model. The number of decisions observed in the model and in the log-file may deviate in case some of the decisions are never executed, in which case they will not appear in the log file. We indicate them in brackets if during our exploration we did not visit, and hence log, each decision at least once. In the case of DisCSP, there are

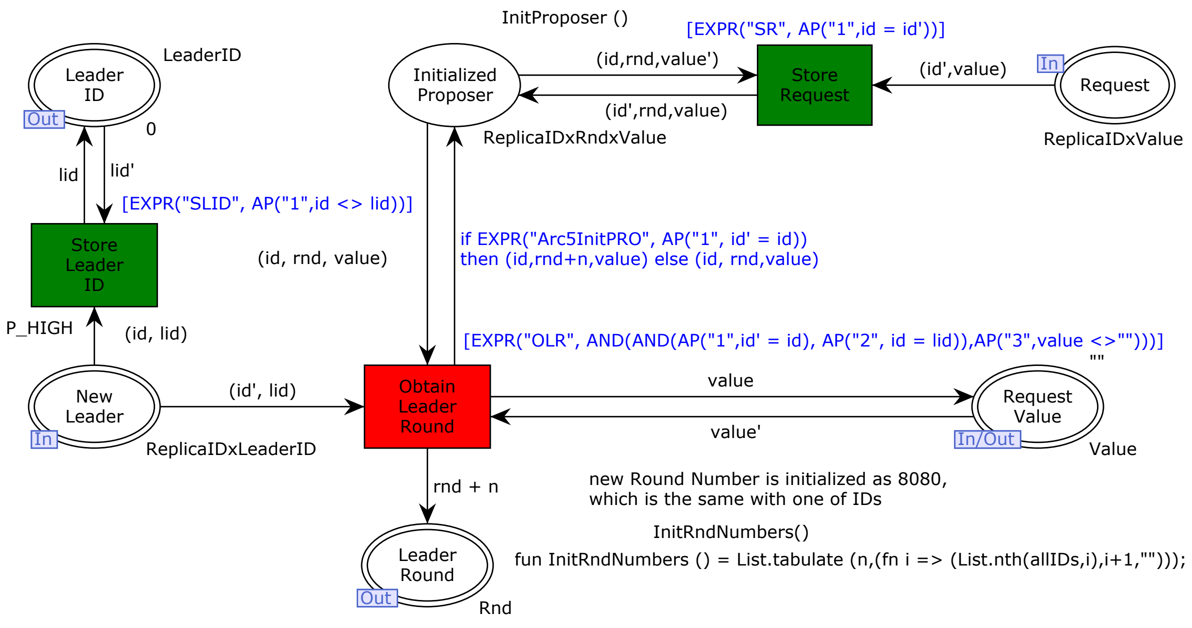

InitProposer ()
[EXPR("SR", AP("1",id = id'))]
Leader ID
LeaderID
Out
0
lid
lid'
[EXPR("SLID", AP("1",id <> lid))]
Store Leader ID
P_HIGH
(id, lid)
New Leader
In
(id', lid)
ReplicaIDxLeaderID
(id, rnd, value)
Initialized Proposer
(id,rnd,value')
(id',rnd,value)
ReplicaIDxRndxValue
Store Request
(id',value)
In
Request
ReplicaIDxValue
if EXPR("Arc5InitPRO", AP("1", id' = id))
then (id,rnd+n,value) else (id, rnd,value)
[EXPR("OLR", AND(AND(AP("1",id' = id), AP("2", id = lid)),AP("3",value <>"")))]
""
Obtain Leader Round
value
value'
Request Value
In/Out
Value
rnd + n
new Round Number is initialized as 8080,
which is the same with one of IDs
InitRndNumbers()
fun InitRndNumbers () = List.tabulate (n,(fn i => (List.nth(allIDs,i),i+1,"")));
Leader Round
Out
Rnd

Fig. 5.2: Coverage visualization in CPN model

Table 5.1: MC/DC coverage results for example CPN models

CPN Model	Executed decisions	Model decisions	Non-trivial decisions	MC/DC (%)	BC (%)	Simulation status
Paxos	2,281,466	27	11	37.03	40.74	incomplete
MQTT-timeout	3,654	18	14	11.11	22.22	incomplete
MQTT-notimeout	1,828,751	23	19	21.73	65.22	complete
CPNABS	1,386,642	32	13	59.37	88.88	incomplete
DisCSP WCS	140,680	9(2)	5	57.14	57.14	complete
DisCSP SBT	7,686	7	3	57.14	57.14	complete
DisCSP ABT	604,055	7	5	57.14	57.14	complete
NPP	194,481	13	13	53.84	92.3	incomplete
PC	8,677,800	10	9	90	90	incomplete
TIS	10,789,149	19	19	52.94	73.68	incomplete
VM	4,444	8	7	25	50	incomplete
T34PIM	3,644,768	23	8	69.56	82.6	complete

two guard decisions which were never executed.

The column *"Non-trivial decisions"* gives the number of the decisions (out of all decisions) that have at least two conditions in the model, as they are the interesting ones while checking independence effect. If a decision has only one condition, it is not possible to differentiate MC/DC from BC. Columns "MC/DC(%)" and "BC(%)" present the coverage percentage for the CPN models under test. The percentage is calculated as the number of covered conditions over the total number of conditions in case of MC/DC and the ratio of covered decisions/branches and the total number of decisions/branches. MC/DC is covered if all the conditions show the independence effect on the outcome. BC is covered if all the branches are taken at least once. This makes MC/DC a stronger coverage criterion compared to BC, and this can be seen from Table 5.1 (columns 5 and 6) where BC have a higher percentage in coverage compared to MC/DC. More detailed results are presented in our paper [5] included in Part II.

Our instrumentation does not have a significant impact on the execution time of the model. Considering the time taken for the full SSE of the finite state models, for instance DisCSP model, both without and with instrumentation, it takes 212.346 seconds versus 214.922 seconds, respectively. It is around 1% of overhead which is the cost for the instrumentation.

It is interesting to observe the quality differences of the coverage results for the tested models. Some of the tested models have less than half of their decisions covered. This should attract the attention of developers and they should assess whether they have tested their models enough, as these results indicate that there is something that might be considered doubtful and require to revisit their test-suite. Two factors affect the coverage percentage results presented for these models:

1. The tested models had no clear test suites; they might be lacking test cases to cover the remaining conditions. Depending on the purpose of each model, some of the test cases may not be relevant for the model or the model may not even have been intended for testing. This could be solved by using test case generation

for uncovered decisions.

2. The models might be erroneous in the sense that some parts (conditions) in the model are never or only partially executed due to a modelling issue, e.g. if the model evolved and a condition no longer serves any purpose or is subsumed by other conditions. For example in the DisCSP model, there are two decisions which were never executed, and we cannot tell if this was intentionally or not without knowing the goal of the developers.

A main result of our analysis of the example models is that none of the models (including those for which the state space could be fully explored) have full MC/DC or BC. This confirms our hypothesis that code coverage of net inscriptions of CPN models can be of interest to developers, such as revealing not taken branches of the if-then-else arc expressions, never executed guard decisions, conditions that do not independently affect the outcome and some model design errors. Our results show that even for full SSE, we may still find expressions that are not MC/DC covered. Assuming that the model is correct, improving coverage then requires improving the test suite. This confirms the relevance and added value of performing coverage analysis of net inscriptions of CPN models over the dead places/transitions report provided as part of the state space generation. A natural next step in a model development process would be for the developers to revisit the decisions that are not MC/DC covered and understand the underlying reason.

5.2 Related Work on MC/DC Analysis in CPN Models

Coverage analysis has attracted attention in both academic and industrial research. Especially the MC/DC criterion is highly recommended and commonly used in safety critical systems, including avionic systems [124]. However, there is a limited number of research addressing model-based coverage analysis. Ghosh [144] expresses test adequacy criteria in terms of model coverage and explicitly lists *condition coverage* and *full predicate coverage criterion* for OCL predicates on UML interaction diagrams, which are semantically related to CPNs in that they express (possible) interactions. Test cases were not automatically generated. In [160], the authors present an automated test generation technique, MISTA (Model-based Integration and System Test Automation) for integrated functional and security testing of software systems using high-level Petri nets as finite state test models. None of the above works addressed structural coverage analysis such as MC/DC or BC on CPN models.

Simulink [139] supports recording and visualising various coverage criteria including MC/DC from simulations via the Simulink Design Verifier. It also has two options for creating test cases to account for the missing coverage in the design. Test coverage criteria for autonomous mobile systems based on CPNs ware presented by Lill et al. in [102]. Their model-based testing approach is based on the use of CPNs to provide a compact and scalable representation of behavioural multiplicity to be covered by an appropriate selection of representative test scenarios fulfilling net-based coverage criteria. Simão et al. [140] provide definitions of structural coverage criteria family for CPNs, named CPN Coverage Criteria Family. These coverage criteria are based on checking if all-markings, all-transitions, all-bindings, and all-paths are tested at least once. Our work is different from the above presented work in that we are analysing the coverage of net inscriptions (conditionals in SML decisions) in CPN models based on structure coverage criteria defined by certification standards, such as DO-178C [132].

CHAPTER 6

GENERATING TEST CASES SATISFYING MC/DC

Test cases generation is an integral part of software testing and deals with providing verdicts that are used to test the behavior of the program. However, software testing techniques that achieve coverage effectiveness and provide test cases are cost intensive [145]. Different strategies are used for test cases generation and they differ in the criteria intended to satisfy, types of inputs (programs/models versus single decisions), and/or method of searching (exhaustive versus greedy). This chapter investigate specifically the generation of test cases supporting the MC/DC criterion. This chapter summarizes the article [4] which is in Part II of this thesis. We present our new and alternative method for MC/DC test cases generation based on binary decisions diagrams (BDDs), and we discuss the state of the art. In addition, we provide future directions for improving and extending our work.

6.1 Approach for MC/DC Test Cases Generation based on BDDs

Test cases are necessary for coverage analysis in order to examine the behavior of the system under test on a given sample set of tests. In addition, they are used to check if a given coverage criterion is satisfied. For MC/DC criterion, test case generation is a non trivial task [67, 90], due to the independence effect requirement while a user also need to comply with a considerable minimal number of test cases.

The main problems for generating test cases that show full MC/DC coverage include:

1. How to select an MC/DC cover test set with a size equal or closer to the lower bound $(n + 1)$ within reasonable effort and reasonable resources.
2. How to generate test cases that support all different forms of MC/DC (UC-MC/DC and Masking MC/DC).
3. It is not always possible to explore all combinations of inputs to conditions in a decision since it is exhaustive and impracticable for a high number of conditions [73, 74, 89]. It is necessary to investigate how to generate a set of solutions that gives a user a possibility to select interesting MC/DC pairs with respect to the specific purpose of MC/DC measurements.

The problem of test cases generation fulfilling MC/DC is NP complete and NP hard [74, 88]. *NP complete* means that there is no single existing approach that can generate optimal set of test cases without requiring exponential time and space, whereas *NP hard* means that there is no known efficient procedure for determining the test cases. We provide a novel and alternative approach to test case generation satisfying MC/DC based on reduced-ordered binary decision diagrams (roBDDs).

6.1.1 Terminologies and properties

In our approach for MC/DC test cases generation [4], we extract the pairs of paths with independence effect in our three valued logic from BDDs. Given an roBDD for some decision D over Boolean variables $x_0, \dots, x_1$. We denote a path from the root of the BDD to a terminal with π, and write $\pi[x] = 1$ if the path takes the true-branch in the node labelled with condition x (0/false respectively), and $\pi[x] =?$ if the path does not pass through a node labelled with condition x. That is, although paths through the roBDD can be of different lengths, for uniformity we always represent them as a vector with n elements.

From roBDD, we produce a set of three-valued test cases. Therefore, we extend general results from the standard two-valued Boolean logic (cfr. Definition 9) to a three-valued logic.

Definition 12 (Three-valued independence pair, $\oplus_c^3$)**.** *Given two* three-valued *test cases* tc, tc' *for a decision* D, *we write* $uc3(tc, tc')$ *iff i)* $D(tc) = \neg D(tc')$ *(they evaluate to opposite* concrete *truth values), and ii)* $tc \oplus_c^3 tc'$, *where* $\oplus_c^3$ *means at least one of the inputs for some condition* c *is a concrete truth value, and for every other condition the three-valued inputs coincide or one of them is "?".*

Example 3. Let $D(X, Y, Z) = X \wedge ((\neg Y \wedge \neg Z) \vee (Y \vee Z))$. Consider $tc = (0??)$ with $D(tc) = 0$ and $tc' = (11?)$ with $D(tc') = 1$ respectively, hence $uc3(tc, tc')$. Observe that hence also e.g. $uc3(011, 11?)$ and $uc(011, 111)$.

Each three-valued independence pair can be instantiated to some two-valued independence pair by suitable substitution of unknown values to fulfill the original definition of MC/DC.

Definition 13 ($merge(tc, tc')$)**.** *Given test cases* tc, tc', *we obtain* $\sigma = merge(tc, tc')$, *where* $\forall c \in C, (\sigma[c] = tc[c] \wedge tc'[c] = ?) \vee (\sigma[c] = tc'[c] \wedge tc[c] = ?)$.

In other words, *merge* substitutes some ? in a pair of paths, such that all conditions have equal values. The result is undefined if they disagree in one position where one has true and the other false. This can be understood as unifying both test cases with each other, taking ? as free variables.

We consider path that can be used for more than one condition. Therefore we define a reuse factor for each path in the ROBDD.

Definition 14 (Reuse factor $\alpha(\pi, \psi)$, $\alpha_{=_3}(\pi, \psi)$)**.** *Given the set of MC/DC pairs of paths* $(\pi^{\perp}, \pi^{\top}) \in \psi$ *with* $D(\pi^{\perp}) = 0$ *and* $D(\pi^{\top}) = 1$, *the reuse factor* $\alpha(\pi, \psi)$ *represents the number of pairs in* ψ *that use* π. *It is calculated as* $\alpha(\pi, \psi) := |\{(\pi, (\pi^{\perp}, \pi^{\top})) \mid \pi = \pi^{\perp} \vee \pi = \pi^{\top}, (\pi^{\perp}, \pi^{\top}) \in \psi\}|$.

We refine existing test cases such that we keep only one test case when two cases overlap. Then, suitable test cases that we might want to add to our set are identified. Therefore, for every uncovered condition our algorithm adds a new test case together with a complementary one (if such test case does not exist in our set) such that the pair shows the independence effect of the condition.

6.1.2 Proposed methods and algorithm for test case generation

We present an algorithm in [4] that takes as input the roBDD representing a Boolean expression and constructs a set of MC/DC pairs. For a decision of n conditions, we generate n pairs that contain between $n+1$ to $2n$ test cases altogether. We select paths based on their length in roBDDs and reuse factor ($\alpha()$). The reuse factor refers to the number of pairs that use a given path.

We propose and compare heuristics with different preferences with respect to three-valued truth-values (1, 0 and ?) and the length of paths in the roBDD. All of them maximize the reuse factor ($\alpha()$) together with a second criteria, namely: the longest paths in BDD ($\mathcal{H}_{LPN}$, $\mathcal{H}_{LPB}$), the longest paths which may merge ($\mathcal{H}_{LMMN}$,$\mathcal{H}_{LMMB}$), and the longest paths with better size ($\mathcal{H}_{LPBS}$). Each type of heuristic implements two different flavors which sort the BDD paths depending on the interpretation of the reuse factor as a natural number ($\mathcal{H}_{LPN}$, $\mathcal{H}_{LMMN}$) or as a boolean value ($\mathcal{H}_{LPB}$, $\mathcal{H}_{LMMB}$) (e.g., $\alpha(p, \psi) < \alpha(q, \psi)$).

6.1.3 MC/DC independence pairs selection

In a BDD, a pair (tc, tc') of test cases showing the independence of some condition c_i has a vivid graphical interpretation on the BDD. It corresponds to a pair of paths $(\pi^{\perp}, \pi^{\top})$ such that:

1. the tests evaluate the opposite truth values (i.e., $D(tc) = \neg D(tc')$);
2. $tc \leqq \pi^{\perp}, tc' \leqq \pi^{\top}$ (order wlog., the test cases may contain more input than strictly necessary).
3. both reach some node v_{c_i} using the same path through BDD(D) (i.e., $\pi^{\perp}[j] = \pi^{\top}[j]$ for $0 \leqslant j < i$);
4. their paths from v_{c_i} exit on either edge (i.e, $\pi^{\perp}[i] = \neg\pi^{\top}[i]$);
5. after v_{c_i}, both test cases take compatible choices along the paths for the remaining conditions, so that the independence property holds (i.e., $\pi^{\perp}[j] =_3 \pi^{\top}[j]$ for $i < j < n$).

In particular, this means that the two paths cannot cross (after the condition-node v_{c_i}), since this would immediately indicate an incompatible choice.

In short, our approach is divided in two stages: during the first phase, it initializes the MC/DC test suite with paths that are extracted from the BDD through any of our predefined heuristics, which intend to maximize the reuse factor in order to reduce the differences among test cases. Secondly, the selected BDD paths are specialized so that the wildcards take a concrete value while preserving the independence effect.

Our algorithm is implemented in Python using the PyEDA library [56]. We test our algorithm on the Traffic Alert and Collision Avoidance System (TCAS II) benchmarks [156] which are widely used in the literature [71, 74, 86, 87, 161]. Our results present MC/DC solutions of size $n + 1$ by performing few permutations of conditions in a decision for all tested decisions. Other possible solutions which show full MC/DC coverage are presented in [4]. In general, our solutions have a size ranging from $n + 1$ to $2n$, with a high percentage of size $n + 1$ or $n + 2$ solutions, where even the latter, although not optimal, may be acceptable to a user. We proposed different heuristics and compared their properties.

6.2 Related Work on MC/DC Test Case Generation

Automatic test data generation approaches were proposed in [20, 69, 159] and are based on greedy or meta-heuristic search strategy. They use search algorithms to extract test paths from the control flow graph of a program, then invoke an SMT solver to generate test data [69] and afterwards reduce the test-suite with a greedy algorithm. The drawback for this approach is that often infeasible paths are selected, resulting in significant wasted computational effort. In our work, we did not investigate test *data* generation here, only boolean inputs to a single decision.

Yang et al. [161] presented the results in terms of size of the test suite and optimal solutions for MC/DC using SAT based approach in comparison to greedy approach for different Boolean expressions. Kitamura et al. [96] and Yang et al. [161] use a SAT solver to construct minimal MC/DC test suites. That is, the MC/DC criterion is encoded in a single query, and the solver produces a suitable assignment for test case inputs if it exists, or times out. In contrast to the exhaustive nature of SAT queries which may lead to timeouts, our approach delivers a single answer in much less time, but may require repetition to find an optimal solution.

The results from SAT solver do not satisfy UC-MC/DC in some cases, and generated test cases are only for Masking MC/DC. There are also some conditions which are reported as infeasible, while the MC/DC pairs for those conditions can be found. For example in [96], decisions 6 and 8 of the TCAS II benchmarks have test suites with 3 and 4 test cases for 8 and 9 conditions respectively which cannot satisfy MC/DC.

A study of enhanced MC/DC coverage criterion for software testing based on n-cube graphs and gray code is presented in [39]. It is an exhaustive approach that takes input as a Boolean expression, builds the n-cube graph, and deduces test cases from all vertices of the graph. Their test cases selection is based on the weight of each test case in a similar way as we calculate the reuse factor of a path. The main difference is that they have to construct the n-cube graph which have the same effect as exhaustive traversal of a truth table, and the resulting size of the test suite is not minimal.

Gay et al. [66, 67], developed a technique to automatically generate test cases using model checkers for masking MC/DC. Using the JKind model checker, they produce a list of all test inputs and then select the desired test cases while preserving the coverage effectiveness. Their test suite reduction algorithm used to reduced the original test-suite does not guarantee to find the smallest set. They tested their approach on different real-world avionics systems where they achieved an average MC/DC coverage of 67.67%.

Comar et al. [47] discussed MC/DC coverage in terms of BDD coverage. They examine the set of distinct paths through the BDD that have been taken based on the control flow graph. Based on BDDs they investigated the formalization and comparison of MC/DC to object branch coverage, but the test cases selection is not within the scope of their work. We extend the formalization and definitions of MC/DC in terms of BDDs in the context of test cases selection.

The roBDDs have been used in [71, 87] for test cases generation, where they highlight the properties and benefits of roBDDs. However, MC/DC was not considered as coverage criterion. Similar to our approach, their greedy approach incrementally selects a pair of paths where only one condition changes for every condition.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

This chapter provides the concluding remarks and summarizes our contributions. In addition, we discuss the limitations and challenges of our approaches and propose a way forward to addresses those limitations. Moreover, the future outlook of this thesis and possible extensions of our work are presented.

7.1 Revisiting of Research Questions

We investigated four research questions with respect to efficient techniques and tools for software testing based on traces and coverage analysis. The first two research questions focused on MC/DC measurement and data race detection without instrumentation and with lightweight instrumentation, respectively. The third research question explored the applicability of MC/DC on design level models. In the fourth research question, we investigated how to generate test cases satisfying MC/DC criterion.

RQ1: *Can we check modified condition decision coverage (MC/DC) without instrumentation?*

RQ2: *How can we monitor data races with low overhead instrumentation?*

RQ3: *Does MC/DC have applicability on the design level models?*

RQ4: *How can test cases satisfying MC/DC be efficiently generated?*

Below, we provide a summary of how we addressed these research questions.

To address **RQ1**, we developed and described a non-intrusive MC/DC measurement tool based on traces without software instrumentation [2, 9]. We measured MC/DC on the object code level by analyzing program traces and investigate how conditionals in the source code are reconstructed during the execution. This was achieved by finding out which conditions in the source code correspond to which conditional jumps in the object code and whether the conditional jump was taken or not. From this information we filled in a table on how the conditions were reconstructed which allowed us to evaluate MC/DC. Our trace based approach of measuring MC/DC complies with the position of CAST-17 [38] and the structure coverage analysis on the object code level where our short-circuit evaluation is equivalent to Masking MC/DC, which is accepted in avionic industry by the DO-178C [124].

In **RQ2** we presented a *non-intrusive* approach to monitoring applications on embedded system-on-chips (SoCs) for data race detection. We used the COEMS platform [49] which aims to eliminate the overhead of dynamic checking by offloading it to external hardware. We considered potential data races in applications that use locks for synchronisation. Our results show that developers can observe the control-flow in a digital twin of their application under test on an embedded systems using the COEMS platform, without affecting the behaviour of the system under test. Our hardware-based approach can be used in safety-critical systems such as the aerospace and railway-domains where certification is necessary.

RQ3 explored the applicability of MC/DC on design level models [5, 10]. In our case studies we considered coverage analysis of net inscriptions in CPN models. We integrate the coverage analysis and visualization of coverage information in CPN models. Our results show that coverage analysis is a useful feature not only for programs but also for models. The reflection from coverage results from the publicly available CPN models considered revealed that some parts of the model were not covered and some of the models yield a low coverage [5, 10]. Therefore, developers (authors of the models) need to consider the reason why some parts are not covered.

In **RQ4**, we investigated a novel and alternative approach to test case generation satisfying MC/DC based on BDDs [4]. We proposed and compared different heuristics-based methods and provided an algorithm for selecting MC/DC pairs given a Boolean expression. Our approach was evaluated on the TCAS II Benchmark and the results showed that we frequently find solutions which are equal or close to the minimal number of test cases (n+1 for a decision with n conditions) without expensive back-tracking.

7.2 Summary of our Contributions

This thesis provides contributions to the theoretical foundations, contributions to MC/DC measurement tools, contributions to data race detection, MC/DC application on case studies, and contributions to the test case generation. We build a theory foundation for testing approaches; MC/DC measurement and data race detection based on the state of the art in the form of related work. It encompasses approaches that helped us to develop efficient software testing tools and techniques, test cases and testing scripts, and their evaluation on SUT case studies and through experiments. We proposed an algorithm and heuristics for test cases generation satisfying MC/DC. We formulate definitions and theorems as well as their proofs of correctness.

7.2.1 *Contributions to MC/DC measurement tools and techniques*

We presented an approach and a tool that show the feasibility of measuring MC/DC without instrumentation based on program traces as an alternative to state-of-the-art solutions like software instrumentation. The tool is able to detect decisions and conditions in C source code and to find their corresponding conditional jumps in the object code. MC/DC can be measured by reconstructing condition assignments based on Intel PT traces.

Our approach of measuring MC/DC based on traces complies with the position of CAST-17, that provides certification authorities' concerns and position regarding the analysis of structural coverage at the object code level [38]. With the mapping between conditions and conditional jumps, we provide traceability between source and object code via the reconstruction of condition assignments on the source code level, we can provide the same level of assurance as measuring coverage directly on source code level via software instrumentation. The main contribution of our work as detailed in [2] is the novel concept of how to measure MC/DC based on existing trace-technologies. In addition, we provide tooling for MC/DC analysis and measurement which is freely available [1] for the research purposes.

7.2.2 Contributions to data race detection

The main contribution of our work to data race detection is the non-intrusive approach to monitoring applications on embedded system-on-chips (SoCs) for data races using the COEMS platform [49]. The platform has a main purpose of eliminating the overhead of dynamic checking by offloading it to external hardware[2]. Additionally, the platform offers control-flow reconstruction from Arm CoreSight processor-traces and data-traces through explicit instrumentation [125]. Race checking is executed on an FPGA on a separate hardware-platform to minimize impact on the system under observation.

Our hardware-based approach finds a wide use in safety-critical systems such as the aerospace and railway-domains where certification is necessary. In these domains, using a software inline race-checker such as ThreadSanitizer [137] is not possible as the tooling for instrumentation and online race-checking is not certified for those systems. In contrast to software-based approaches, our instrumentation for the application under test has low complexity and gives predictable performance overhead independent of whether or not race checking is enabled. This is especially important for software development in these safety-critical domains, as again for certification purposes, it is not permissible to, e.g., deploy a separate version for debugging or trouble-shooting on demand in the field. Any debugging and trace support must be already integrated in the final product.

Moreover, we have illustrated our approach with a case study, where we simulate a set of bankers sending random amounts of money from one bank account to another [85]. The bankers lock the source and target bank accounts before committing a transaction, so that transfers are protected against data races and deadlocks. We introduce a special case where one banker forgets one lock operation and hence, data may get corrupted. The complete example, including source code, execution traces and TeSSLa reports, is available at [6].

7.2.3 Contributions to MC/DC applicability on design model

The application of simulation and state space exploration for validating CPN models traditionally focuses on behavioural properties related to net structure, i.e., places and transitions. We developed tooling and techniques to analyse the extent to which the Boolean expressions in CPN model have been covered is by analyzing the coverage of the net inscriptions that were only implicitly validated.

There are five main contributions with respect to coverage analysis of net inscriptions in CPN models: 1) We developed an automatic lightweight instrumentation mechanism

[1]https://www.coems.eu/mc-dc/

[2]The EU Horizon 2020 project "COEMS–Continuous Observation of Embedded Multicore Systems", https://www.coems.eu.

that rewrite the guards and arc expressions in a form that allows to identify conditions and decisions without modifications of the net structure of the CPN model 2) we provide a library and annotation mechanism that intercept evaluation of Boolean conditions in guards and arcs in SML decisions in CPN models, and record how they were evaluated; 3) we compute the conditions' truth assignment and check whether or not particular decisions are MC/DC-covered in the recorded executions of the model; 4) we collect coverage data using our library from publicly available CPN models and report whether they are MC/DC and BC covered; 5) we visualize the coverage information in CPN model such that the covered and uncovered (coloured in green and red respectively) transitions and arcs are revealed.

7.2.4 *Contributions to the test cases generation*

We presented a novel and alternative approach to test cases generation satisfying MC/DC based on reduced-ordered binary decision diagrams (roBDDs) which are a concise representation of Boolean expressions.

We proposed an algorithm that takes as input the roBDD representing a Boolean expression and constructs a set of MC/DC pairs. For a decision of n conditions, we generate n pairs that contain between $n + 1$ to $2n$ test cases altogether. We select paths based on their length in roBDDs and reuse factor ($\alpha()$, cfr Definition 14). The reuse factor refers to the number of pairs that use a given path.

We proposed and compared our heuristics with different preferences with respect to three-valued truth-values (1, 0 and ?) and the length of paths in the roBDD. Our algorithm is implemented in Python and the PyEDA library [56]. We test our algorithm on the Traffic Alert and Collision Avoidance System (TCAS II) benchmarks [156] which are widely used in the literature [71, 74, 86, 87, 161].

7.3 Limitations and Future Work

The approaches and techniques proposed in this thesis provide several directions for future work in the context of MC/DC measurement and its applicability on real industrial examples and other type of models. New methods for test cases generation and data race detection can be envisaged as future outlook.

There are some limitations and challenges encountered in this thesis that are presented next along with possible future research directions. In this section, we discuss future work based on our research methods and activities, by outlining research directions for future work. Measuring MC/DC based on jumps has some general limitations. If the compiler uses any optimization level, it is likely that conditions are not directly translated to conditional jumps. It is possible that conditional moves, jump tables or indirect branches are used. Program traces deliver no information on how these instructions are evaluated and therefore none of these structure can be used to reconstruct conditions by analyzing the trace. This limitation is less severe in the domain of avionic, because other requirements, for example source code to object code traceability in DO-178C, make it already hard for developers to use high optimization levels [35]. It is necessary to conduct further investigations into this issue in the future.

Another limitation of our approach to measuring MC/DC based on traces is that the trace data becomes excessively large for longer executions. We used an offline tracing approach where available storage effectively limits the size of traces. In future work, we want to apply our approach to online trace reconstruction which would enable us to observe much longer execution times because only the very events that are used for coverage measurement are reconstructed.

We also want to support other architectures and instead of Intel PT, use tracing technologies such as ARM CoreSight and NEXUS for PowerPC since these processor architectures are widely used in the avionics and automotive industry, which would benefit the most from this new approach of MC/DC measurement.

In addition, measuring MC/DC on bigger industrial programs, for example from git repositories, and other models is another possible direction for the future. This would serve as a ground proof for the expected MC/DC for these programs and models since none exist to the best of our knowledge.

The data race analysis uses the LLVM compiler framework, and currently works with threads using `pthread_mutex_lock/unlock` operations for protecting the shared variables. For other ways of synchronization, e.g., through compare-and-swap instructions, or baremetal execution, we do not provide instrumentation and a template yet, but they can easily be adapted from our code.

A practical limitation of the data trace is the currently restricted value-range of the trace messages to 16 bits, which complicates e.g. the use of pointers in the trace. As currently we need multiple messages per event to transmit additional data such as debugging information (the current line number) and the thread identifier, we need to serialize use of the trace bus. This additional locking that is introduced through the instrumentation affects the performance of the application under test, whereas transmitting a single datum in principle has negligible execution overhead.

Our unoptimised performance measurements already put us in a competitive range with other approaches such as ThreadSanitizer, and we have the advantage that COEMS-based tracing can remain enabled in production. Future developments of the COEMS platform beyond its current prototype will make splitting the specification and post-processing in the interpreter superfluous: 18 (instead of the currently 8) available EPUs will already allow for setups without dynamic values to be handled completely in hardware. In the meantime, we are improving the instrumentation to produce effect summaries for basic blocks of code instead of instrumenting single instructions, which should decrease the overhead especially for tight loops. We are also preparing additional concurrency patterns that monitor actual deadlocks and so-called lock-order-reversal.

Our general approach to coverage analysis provides several possible directions forward which would help developers get a better understanding of their models: While generating the full state space is certainly the preferred approach, this is not feasible if the state space is inherently infinite or too large. Simulation of particular executions could then be guided by results from the coverage and try to achieve higher coverage in parts of the model that have not been explored yet. However, while selecting particular transitions to follow in a simulation is straight-forward, manipulating the data space for bindings used in guards is a much harder problem and closely related to test case generation (recall the CPNs also rely on suitable initial states, which are currently given by developers). Making use of feedback between the state of the simulation and the state of the coverage would, however, require much tighter integration of the tools. A related direction is to consider visualizing coverage information in the graphical user interface accompanied with a set of test cases that can be used to cover the uncovered part: CPN Tools already supports a broad palette of visual options that could be used, e.g., to indicate successful coverage of guards through colour, or the frequency that transitions have been taken through their thickness [153]. However, the related challenge would be that the suggested set of test cases may not be interesting with respect to the user defined intention of the CPN models under test.

As for the measured coverage results, it would be interesting to discuss with the original developers of the tested models, and check if the coverage results are within their expectations. While low coverage could indicate design flaws, our testing may not have exercised the same state space as the original developers did: they may have used their model in various configurations, whereof the state of the model just represents a snapshot, or we did not discover all possible configurations in the model.

The main limitation of our approach to MC/DC test cases generation based on BDDs is that BDDs are sensitive to conditions ordering, such that different orders yield different BDDs and their size in the worst case grows to 2^{2^n} nodes [117]. As the number of nodes increases, there are many paths to select MC/DC pairs from. We presented evidence that to find an optimal or "good enough" solutions, instead of a search with backtracking, it is sufficient to try a few different permutations.

Another challenge which is not directly related to our approach but to MC/DC is the coupled and masked conditions where it is difficult to get a full MC/DC coverage with masked condition. Further methods which are not based on BDDs can also be envisaged.

For the future work, it is interesting to extend our algorithm so that it can support data input coverage where conditions are not abstracted, which requires taking constraints into consideration. One can also attempt to integrate our test case generation algorithm into our MC/DC measurement tool and model [2, 10]. Although the experimental data shows that we always find an optimal solution, it remains open question if this is a general property of our approach.

Both the MC/DC measurement on the design model level and industrial examples requires having a test suite which in some models and programs may not even have been intended for testing. As part of future work, it is interesting to generate test-cases specifically with the aim to increase coverage.

BIBLIOGRAPHY

[1] M. A. Adamski, A. Karatkevich, and M. Wegrzyn. *Design of Embedded Control Systems*, volume 267. Springer, 2005. 2.6

[2] F. Ahishakiye, S. Jakšić, V. Stolz, F. D. Lange, M. Schmitz, and D. Thoma. Non-Intrusive MC/DC Measurement based on Traces. In D. Méry and S. Qin, editors, *Intl. Symp. on Theoretical Aspects of Software Engineering*, pages 86–92. IEEE, 2019. 1, 1, 1.5, 1.5, 1.6, 3, 3.1.2, 7.1, 7.2.1, 7.3

[3] F. Ahishakiye, J. I. R. Jarabo, L. M. Kristensen, and V. Stolz. Coverage Analysis of Net Inscriptions in Coloured Petri Net Models, 2020. 1.6

[4] F. Ahishakiye, J. I. R. Jarabo, L. M. Kristensen, and V. Stolz. MC/DC Test Cases Generation based on BDDs. In *Symposium on Dependable Software Engineering Theories, Tools and Applications (SETTA 2021)*, volume 13071 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2021. 1.5, 1.5, 1.6, 2.2.1, 2.7.2, 6, 6.1.1, 6.1.2, 6.1.3, 7.1

[5] F. Ahishakiye, J. I. R. Jarabo, L. M. Kristensen, and V. Stolz. Coverage Visualization and Analysis of Net Inscriptions in CPN Models. *Innovations in Systems and Software Engineering (ISSE) NASA Journal (Under review)*, pages 1–22, Mars 2022. 1.5, 1.5, 1.6, 5, 5.1.2, 7.1

[6] F. Ahishakiye, J. I. R. Jarabo, K. I. Pun, and V. Stolz. Open Data for Banker Example. https://doi.org/10.5281/zenodo.4381982, December 2020. 7.2.2

[7] F. Ahishakiye, J. I. R. Jarabo, V. K. I. Pun, and V. Stolz. *Hardware-Assisted Online Data Race Detection*, volume 13065 of *Lecture Notes in Computer Science*, pages 108–126. Springer International Publishing, Cham, 2021. 1.5, 1.6, 2.4, 4, 4.1.2

[8] F. Ahishakiye, J. I. R. Jarabo, and V. Stolz. Lock instrumentation tool. https://github.com/selabhvl/coems-racechecker, 2020. 1.6, 1

[9] F. Ahishakiye and F. D. Lange. Non-intrusive MC/DC Measurement based on Traces. In *Proceedings of the PhD Symposium at iFM'18 on Formal Methods: Algorithms, Tools and Applications (PhD-iFM'18)*, pages 15–17, Maynooth, Ireland, Sept 2018. 1, 1, 1.5, 1.6, 7.1

[10] F. Ahishakiye, J. I. Requeno Jarabo, L. M. Kristensen, and V. Stolz. Coverage Analysis of Net Inscriptions in Coloured Petri Net Models. In B. Ben Hedia, Y.-F. Chen, G. Liu, and Z. Yu, editors, *International Conference on Verification and Evaluation of Computer and Communication Systems (VECoS)*, volume 12519 of *Lecture Notes in Computer Science*, pages 68–83, Cham, 2020. Springer. 3, 1.5, 1.6, 5, 5.1.2, 7.1, 7.3

[11] F. Ahishakiye, V. Stolz, and L. M. Kristensen. Coverage Analysis of the Standard ML Expressions in a CPN model. In *Proceedings of the PhD Symposium at iFM'19 on Formal Methods: Algorithms, Tools and Applications (PhD-iFM'2019)*, Bergen, Norway, December 2019. 1.6

[12] F. Ahishakiye, V. Stolz, and L. M. Kristensen. Generating Test-cases Satisfying MC/DC from BDDs. In *The 31st Nordic Workshop on Programming Theory (NWPT'19)*, pages 12–14, Tallinn, Estonia, November 2019. 1.6

[13] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, USA, 1 edition, 2008. 1.1.1, 1.1.2, 1.1.4, 2.1, 2.1, 2.1, 2.1

[14] H. R. Andersen. An Introduction to Binary Decision Diagrams. In *Lecture Notes*, 1997. 2.7.1

[15] K. Andi. Cheat sheet for Intel Processor Trace with Linux perf and gdb, 1 June 2019. http://halobates.de/blog/p/410. 1, 1.2, 2.5.1, 2.5.2

[16] G. R. Andrew. *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley, University of Arizona, 1 edition, 2000. 2.3, 2.3.2, 2.3.2, 2.3.3

[17] ARM Limited. *ARM IHI 0029B: CoreSightTM Architecture Specification v2.0, issue D*, 2013. 2.3, 2.5.1, 2.5.3

[18] ArmDeveloper. CoreSight Access Tool (CSAT) User Guide User Guide Version 2.6.0, 2013. https://developer.arm.com/documentation/epm051792/latest/. 2.5.3

[19] ArmDeveloper. Arm® DS-5 Arm DSTREAM System and Interface Design Reference Guide Version 5.29, 2015. https://developer.arm.com/documentation/100956/latest/. 2.5.3

[20] Z. Awedikian, K. Ayari, and G. Antoniol. MC/DC automatic test input data generation. In *Annual Conference on Genetic and Evolutionary Computation Conference (GECCO)*, pages 1657–1664. ACM, 2009. 6.2

[21] V. Balasundaram and K. Kennedy. Compile-Time Detection of Race Conditions in a Parallel Program. In *Proceedings of the 3rd International Conference on Supercomputing*, ICS '89, pages 175–185, New York, NY, USA, 1989. Association for Computing Machinery. 2.3.2, 2.4

[22] J. Baumeister, B. Finkbeiner, M. Schwenger, and H. Torfah. FPGA stream-monitoring of real-time properties. *ACM Trans. Embed. Comput. Syst.*, 18(5s), Oct. 2019. 4.2

[23] M. Beaudouin-Lafon, W. E. Mackay, M. Jensen, P. Andersen, P. Janecek, M. Lassen, K. Lund, K. Mortensen, S. Munck, A. Ratzer, K. Ravn, S. Christensen, and K. Jensen. CPN Tools: A Tool for Editing and Simulating Coloured Petri Nets ETAPS Tool Demonstration Related to TACAS. In *Proc. of Tools and Algorithms for the Construction and Analysis of Systems*, pages 574–577, Berlin, Heidelberg, 2001. Springer. 2.6, 5.1

[24] B. Beizer. *Black Box Testing: Techniques for Functional Testing of Software and Systems*. Inc. John Wiley and Sons, Hoboken, NJ, 1995. 1.1.4

[25] J. Billington and M. Diaz. *Application of Petri Nets to Communication Networks: Advances in Petri nets*. Number 1605. Springer Science & Business Media, 1999. 2.6

[26] J. Billington, G. E. Gallasch, and B. Han. A Coloured Petri Net Approach to Protocol Verification. In *Advanced Course on Petri Nets*, pages 210–290. Springer, 2003. 2.6

[27] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The Gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, Aug. 2011. 4.2

[28] E. Bodden and K. Havelund. Aspect-Oriented Race Detection in Java. *IEEE Trans. Software Eng.*, 36(4):509–527, 2010. 2.3

[29] M. Bordin, C. Comar, T. Gingold, J. Guitton, O. Hainque, and T. Quinot. Object and source coverage for critical applications with the COUVERTURE open analysis framework. In *Proc. of Embedded Real Time Software and Systems Conference (ERTS)*, 2010. 3.2

[30] Q. Brainy. Steve Jobs Quotes. https://www.brainyquote.com/quotes/steve_jobs_416899?src=t_software. I

[31] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Comput.*, 35(8):677–691, August 1986. 2.7, 2.7, 2.7.1

[32] T. Byun, V. Sharma, S. Rayadurgam, S. McCamant, and M. Heimdahl. Toward rigorous object-code coverage criteria. Technical report, Technical Report, University of Minnesota, MN, USA, June 2017. 3.2

[33] Caesarea Medical Electronics. Niki T34 syringe pump instruction manual, June 2008. 5.1.1

[34] A. Cavalcanti, S. King, and C. O'Halloran. A Scientific Investigation of MC/DC Testing. 01 2007. 2.2.2, 2.2.3

[35] Certification Authorities Software Team. Guidelines for approving source code to object code traceability. Technical report, Technical Report: Position Paper CAST-12, 2002. 2.2.5, 3.1, 7.3

[36] Certification Authorities Software Team (CAST). Rationale for Accepting Masking MC/DC in Certification Projects. Technical report, Position Paper CAST-6, 2001. 2.2, 2.2.3

[37] Certification Authorities Software Team (CAST). What is a "Decision" in Application of Modified Condition/Decision Coverage (MC/DC) and Decision Coverage (DC)? Technical report, Position Paper CAST-10, 2002. 2.1, 2.2

[38] Certification Authorities Software Team (CAST). Structural Coverage of Object Code. *Technical Report: Position Paper CAST-17*, 2003. 1, 1.2, 2.2.5, 3, 3.1, 3.1.2, 7.1, 7.2.1

[39] J. R. Chang and C. Y. Huang. A study of enhanced MC/DC coverage criterion for software testing. In *Annual International Computer Software and Applications Conference (COMPSAC)*, pages 457–464, 2007. 6.2

[40] Q.-L. Chen, J.-J. Bai, Z.-M. Jiang, J. Lawall, and S.-M. Hu. Detecting data races caused by inconsistent lock protection in device drivers. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 366–376, 2019. 2.4

[41] J. J. Chilenski. An Investigation of Three Forms of the Modified Condition Decision Coverage (MC/DC) criterion. Technical report, Office of Aviation Research, 2001. 2.1, 2.2, 2.2.2, 2.2.3, 2.2.5, 3.1

[42] J. J. Chilenski and S. P. Miller. Applicability of Modified Condition/Decision Coverage to Software Testing. *Software Engineering Journal*, 9(5):193–200, 1994. 1, 1.1.7, 1.2, 2.1, 2.1, 2.1, 2.2, 2.2.1, 2.2.2, 3.2

[43] E. Clarke, O. Grumberg, and D. Long. Verification tools for finite-state concurrent systems. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *A Decade of Concurrency Reflections and Perspectives*, pages 124–175, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg. 2.7

[44] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999. 2.7, 2.7

[45] L. A. Clarke, A. Podgurski, D. J. Richardson, and S. J. Zeil. A comparison of data flow path selection criteria. In *Proceedings of the 8th International Conference on Software Engineering*, ICSE '85, pages 244–251, Los Alamitos, CA, USA, 1985. IEEE Computer Society Press. 2.7, 3.2

[46] CodeCover. An open-source glass-box testing tool. available at http://codecover.org/. 3.2

[47] C. Comar, J. Guitton, O. Hainque, and T. Quinot. Formalization and comparison of MC/DC and object branch coverage criteria. In *European Congress Embedded Real Time Software and Systems (ERTS)*, pages 1–10, 2011. 2.2.4, 6.2

[48] S. Cornett. Code Coverage Analysis, 1996-2014. available at https://www.bullseye.com/coverage.html, Accessed 16 August 2021. 2.1, 2.1

[49] N. Decker, B. Dreyer, P. Gottschling, C. Hochberger, A. Lange, M. Leucker, T. Scheffel, S. Wegener, and A. Weiss. Online analysis of debug trace data for embedded systems. In J. Madsen and A. K. Coskun, editors, *Design, Automation & Test in Europe Conference & Exhibition, DATE 2018*, pages 851–856. IEEE, 2018. 2, 3.2, 4.1.1, 7.1, 7.2.2

[50] N. Decker, P. Gottschling, C. Hochberger, M. Leucker, T. Scheffel, M. Schmitz, and A. Weiss. Rapidly Adjustable Non-intrusive Online Monitoring for Multi-core Systems. In S. Cavalheiro and J. Fiadeiro, editors, *Formal Methods: Foundations and Applications*, pages 179–196, Cham, 2017. Springer International Publishing. 3.2

[51] X. Devroey, G. Perrouin, A. Legay, M. Cordy, P.-Y. Schobbens, and P. Heymans. Coverage Criteria for Behavioural Testing of Software Product Lines. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, pages 336–350, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. 1.1.7

[52] T. V. Dijk, A. Laarman, and J. van de Pol. Multi-Core BDD Operations for Symbolic Reachability. *Electronic Notes in Theoretical Computer Science*, 296:127–143, 2013. 2.7

[53] E. W. Dijkstra. Notes on Structured Programming. April 1970. 1.1.2

[54] E. W. Dijkstra. *Solution of a Problem in Concurrent Programming Control*, pages 289–294. Berlin, Heidelberg, 2001. 2.3.2, 2.3.3

[55] Doxygen. BuDDy: A BDD package. available at http://buddy.sourceforge.net/manual/main.html. 2.7.2

[56] C. R. Drake. PyEDA: Data structures and algorithms for electronic design automation. In *Python in Science Conference (SciPy)*, 2015. 4, 2.7.2, 6.1.3, 7.2.4

[57] M. Factor, E. Farchi, Y. Lichtenstein, and Y. Malka. Testing concurrent programs: a formal evaluation of coverage criteria. In *Proceedings of the Seventh Israeli Conference on Computer Systems and Software Engineering*, pages 119–126, 1996. 2.1, 2.3

[58] FCAS Team. What is a" decision" in application of modified condition/decision coverage (C/DC) and decision coverage (DC). *Technical Report position paper*, 2002. 1, 1.1.7, 2.1, 2.1, 2.1, 2

[59] Federal Aviation Administration. Software approval guidelines, 2011. 1.1.6

[60] H. Felbinger, J. Sherrill, G. Bloom, and F. Wotawa. Test suite coverage measurement and reporting for testing an operating system without instrumentation. In *17th Real-Time Linux Workshop*, Oct. 2015. 3.2

[61] C. Flanagan and S. N. Freund. Fasttrack: efficient and precise dynamic race detection. In M. Hind and A. Diwan, editors, *Proc. 2009 ACM SIGPLAN Conf. on Programming Language Design and Implementation, PLDI 2009*, pages 121–133. ACM, 2009. 4.2

[62] P. Frankl and E. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498, 1988. 1.1.7, 2.1

[63] M. Fujita, H. Fujisawa, and N. Kawato. Evaluation and improvement of Boolean comparison method based on Binary Decision Diagrams. In *[1988] IEEE International Conference on Computer-Aided Design (ICCAD-89) Digest of Technical Papers*, pages 2–5, Nov 1988. 2.7

[64] J. Gait. A Probe Effect in Concurrent Programs. *Softw. Pract. Exper.*, 16(3):225–233, Mar. 1986. 2.3

[65] D. Gates and M. Baker. The inside story of MCAS: How Boeing's 737 MAX system gained power and lost safeguards, 9 July 2019. 1

[66] G. Gay, A. Rajan, M. Staats, M. Whalen, and M. P. E. Heimdahl. The effect of program and model structure on the effectiveness of MC/DC test adequacy coverage. *ACM Transactions on Software Engineering and Methodology*, 25(3), July 2016. 1.2, 6.2

[67] G. Gay, M. Staats, M. Whalen, and M. P. E. Heimdahl. The risks of coverage-directed test case generation. *IEEE Transactions on Software Engineering*, 41(8):803–819, 2015. 1.2, 6.1, 6.2

[68] X. Ge, W. Cui, and T. Jaeger. GRIFFIN: Guarding Control Flows Using Intel Processor Trace. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, page 585–598, New York, NY, USA, 2017. Association for Computing Machinery. 2.5.1, 2.5.2, 2.5.2

[69] K. Ghani and J. A. Clark. Automatic Test Data Generation for Multiple Condition and MCDC Coverage. In *Proc. of the Fourth International Conference on Software Engineering Advances*, ICSEA '09, page 152–157, USA, 2009. IEEE Comp. Society. 6.2

[70] A. Gkolfi, C. C. Din, E. B. Johnsen, L. M. Kristensen, M. Steffen, and I. C. Yu. Translating active objects into Colored Petri Nets for communication analysis. *Science of Computer Programming*, 181:1–26, 2019. 5.1.1

[71] H. Gong, J. Li, and R. Li. CTFTP: A test case generation strategy for general Boolean expressions based on ordered binary label-driven Petri nets. *IEEE Access*, 8:174516–174529, 2020. 4, 6.1.3, 6.2, 7.2.4

[72] O.-K. Ha. Case study of dynamic detectors for data races. *IERI Procedia*, 4:174–180, 2013. 2.4, 2.4

[73] A. Halin, A. Nuttinck, M. Acher, X. Devroey, G. Perrouin, and B. Baudry. Test them all, is it worth it? Assessing configuration sampling on the JHipster Web development stack. *Empirical Software Engineering*, 24(2):674 –717, 2019. 1.2, 3

[74] S. Hallé, E. La Chance, and S. Gaboury. Graph methods for generating test cases with universal and existential constraints. In *International Conference on Testing Software and Systems (ICTSS)*, pages 55–70. Springer, 2015. 1.2, 4, 3, 6.1, 6.1.3, 7.2.4

[75] M. P. E. Heimdahl, M. W. Whalen, A. Rajan, and M. Staats. On MC/DC and Implementation Structure: An empirical study. In *Proc. of IEEE/AIAA 27th Digital Avionics Systems Conference*, pages 5.B.3–1–5.B.3–13, 2008. 2.1

[76] A. J. Hu. Formal hardware verification with BDDs: an introduction. In *IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*, volume 2, pages 677–682. IEEE, 1997. 2.7

[77] IEEE Standards. IEEE Guide for Software Verification and Validation Plans. *IEEE Std 1059-1993*, pages 1–87, 1994. 1.1, 1.1.2, 1.1.3

[78] J. Jahic, M. Jung, T. Kuhn, C. Kestel, and N. Wehn. A framework for non-intrusive trace-driven simulation of manycore architectures with dynamic tracing configuration. In C. Colombo and M. Leucker, editors, *Runtime Verification*, pages 458–468. Springer, 2018. 4.2

[79] S. Jaksic, E. Bartocci, R. Grosu, R. Kloibhofer, T. Nguyen, and D. Nickovic. From signal temporal logic to FPGA monitors. In *13. ACM/IEEE Intl. Conf. on Formal Methods and Models for Codesign, MEMOCODE 2015*, pages 218–227. IEEE, 2015. 4.2

[80] S. Jakšic, D. Li, Ka I Pun, and V. Stolz. Stream-based dynamic data race detection. In *31st Norsk Informatikkonferanse, NIK 2018*. Bibsys Open Journal Systems, Norway, 2018. 2.4, 4.2

[81] G. Janssen. Application of BDD's in formal verification. In *Proc. 22nd International School and Conference on CAD*, pages 49–53, Gurzuff, Yalta, Ukraine, 1995. 2.7

[82] K. Jensen, S. Christensen, L. M. Kristensen, and W. Michael. CPN Tools, 2010. 2.6, 5.1

[83] K. Jensen and L. M. Kristensen. Colored Petri Nets: A graphical language for formal modeling and validation of concurrent systems. *Commun. ACM*, 58:61–70, 2015. 2.6, 5.1

[84] K. Jensen, L. M. Kristensen, and L. Wells. Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems. *International Journal on Software Tools for Technology Transfer*, 9, 06 2007. 2.6

[85] N. Joe. Concurrent programming, with examples. https://begriffs.com/posts/2020-03-23-concurrent-programming.html, March 2020. 7.2.2

[86] J. A. Jones and M. J. Harrold. Test-suite reduction and prioritization for Modified Condition/Decision Coverage. *IEEE Transactions on Software Engineering*, 29(3):195–209, 2003. 4, 6.1.3, 7.2.4

[87] A. Kalaee and V. Rafe. An optimal solution for test case generation using ROBDD graph and PSO algorithm. *Quality and Reliability Engineering International*, 32(7):2263–2279, 2016. 4, 6.1.3, 6.2, 7.2.4

[88] Z. Z. Kamal, R. A. A. Abdul, and H. Mohd. On test case generation satisfying the MC/DC criterion. *International Journal of Advances in Soft Computing and its Applications*, 5(3):104–115, 2013. 6.1

[89] S. Kandl and S. Chandrashekar. Reasonability of MC/DC for safety-relevant software implemented in programming languages with short-circuit evaluation. *Computing*, 97(30):261–279, Mar 2015. 1.2, 3.2, 3

[90] S. Kangoye, A. Todoskoff, and M. Barreau. Practical methods for automatic MC/DC test case generation of Boolean expressions. In *IEEE AUTOTESTCON*, pages 203–212. IEEE, 2015. 1.2, 6.1

[91] K. Kapoor and J. Bowen. Experimental evaluation of the variation in effectiveness for DC, FPC and MC/DC test criteria. In *2003 International Symposium on Empirical Software Engineering, 2003. ISESE 2003. Proceedings.*, pages 185–194, Sept 2003. 3.2

[92] K. Kapoor and J. P. Bowen. A Formal Analysis of MCDC and RCDC Test Criteria: Research Articles. *Softw. Test. Verif. Reliab.*, 15(1):21–40, mar 2005. 2.2.2

[93] H. P. Katseff. A New Solution to the Critical Section Problem. In *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, STOC '78, page 86–88, New York, NY, USA, 1978. Association for Computing Machinery. 2.3.2

[94] H. Kelly J., V. Dan S., C. John J., and R. Leanna K. A Practical Tutorial on Modified Condition/Decision Coverage. Technical report, NASA, 2001. 1.1.7, 1.2, 2.1, 2.1, 2.1, 2.1, 2.1, 2.2.4

[95] M. Kerrisk. perf-intel-pt(1) — Linux manual page, June 2021. available at https://man7.org/linux/man-pages/man1/perf-intel-pt.1.html. 2.5.2

[96] T. Kitamura, Q. Maissonneuve, E.-H. Choi, C. Artho, and A. Gargantini. Optimal test suite generation for Modified Condition Decision Coverage using SAT solving. In *Computer Safety, Reliability, and Security*, pages 123–138. Springer, 2018. 6.2

[97] M. F. Lau and Y. T. Yu. An Extended Fault Class Hierarchy for Specification-Based Testing. *ACM Trans. Softw. Eng. Methodol.*, 14(3):247–276, jul 2005. 2.2.1

[98] Lauterbach. Trace-based MC/DC Coverage, March, 2018. https://www.lauterbach.com. 3.2

[99] J. Lawrence, S. Clarke, M. Burnett, and G. Rothermel. How well do professional developers test with code coverage visualizations? an empirical study. In *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*, pages 53–60, 2005. 1.1.7, 2.1

[100] M. Leucker, C. Sánchez, T. Scheffel, M. Schmitz, and A. Schramm. TeSSLa: Runtime verification of non-synchronized real-time streams. In *ACM Symposium on Applied Computing (SAC)*, pages 1925–1933. ACM, 2018. 1, 2.4, 4.1.2, 4.2

[101] R. Lill and F. Saglietti. Test coverage criteria for autonomous mobile systems based on Coloured Petri Nets. In *Proc. of 9th FORMS/FORMAT 2012 - Symp. on Formal Methods for Automation and Safety in Railway and Automotive Systems*, pages 155–162, TU, Braunschweig, Germany, 2012. 2.6, 5.1

[102] R. Lill and F. Saglietti. Model-based Testing of Cooperating Robotic Systems using Coloured Petri Nets. In *Proc. of SAFECOMP 2013 - Workshop DECS (ERCIM/EWICS Workshop on Dependable Embedded and Cyber-physical Systems) of the 32nd International Conference on Computer Safety, Reliability and Security*, Toulouse, France, Sep 2013. 2.6, 5.2

[103] A. Ltd. CoreSightTM Program Flow TraceTM PFTv1.0 and PFTv1.1, 2015. 2.5.3

[104] S. Lu, W. Jiang, and Y. Zhou. A Study of Interleaving Coverage Criteria. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, page 533–536, New York, NY, USA, 2007. Association for Computing Machinery. 2.3

[105] S. Lu, P. Zhou, W. Liu, Y. Zhou, and J. Torrellas. PathExpander: Architectural Support for Increasing the Path Coverage of Dynamic Bug Detection. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 38–52, 2006. 2.1

[106] M. A. A. Mamun and A. Khanam. Concurrent Software Testing : A Systematic Review and an Evaluation of Static Analysis Tools. Master's thesis, School of Computing, 2009. 1, 1.1.3

[107] E. Manfred. Model Quotes. https://todayinsci.com/QuotationsCategories/M_Cat/Model-Quotations.htm. 4.2

[108] U. Mark and L. Bruno. *Practical Model-Based Testing: A Tools Approach*. Jul 2010. 1.1.5, 1.2

[109] U. Mark, A. Pretschner, and L. Bruno. A taxonomy of model-based testing. 2006. 1.1.5, 1.2

[110] H. Matthias and B. Armin. Standard Glossary of Terms Used in Software Testing, 2020. 1.1.2

[111] C. E. McDowell and D. P. Helmbold. Debugging Concurrent Programs. *ACM Comput. Surv.*, 21(4):593–622, Dec. 1989. 2.3, 2.3, 2.3.2

[112] C. Meinel and T. Theobald. *Algorithms and Data Structures in VLSI Design*. Springer-Verlag, Berlin, Heidelberg, 1st edition, 1998. 2.7

[113] H. Michael and R. Mark. *Logic in Computer Science: Modelling and Reasoning about Systems*, volume 18. Cambridge University Press, 2nd edition, 2008. 2.7.1

[114] M. Micheal and S. Tragoudas. ATPG for path delay faults without path enumeration. In *Proceedings of the IEEE 2001. 2nd International Symposium on Quality Electronic Design*, pages 384–389, March 2001. 2.7

[115] E. Miller. Introduction to software testing technology. *Tutorial: Software Testing & Validation Techniques, Second Edition, IEEE Catalog No. EHO*, pages 180–0, 1981. 1.1, 1.1.1

[116] P. Moosbrugger, K. Y. Rozier, and J. Schumann. R2U2: monitoring and diagnosis of security threats for unmanned aerial systems. *Formal Methods in System Design*, 51(1):31–61, 2017. 4.2

[117] J. Newton and D. Verna. A Theoretical and Numerical Analysis of the Worst-Case Size of Reduced Ordered Binary Decision Diagrams. *ACM Transactions on Computational Logic*, 20(1), Jan. 2019. 7.3

[118] S. C. Ntafos. A comparison of some structural testing strategies. *IEEE Transactio on Software Engineering*, 14(6):868–874, 1988. 3.2

[119] P. Ortiz López, M. Akashi, J.-M. Cosset, P. Gourmelon, S. Vatnitsky, J. Mettler, F.A., and M. Konchalovsky. Investigation of an Accidental Exposure of Radiotherapy Patients in Panama, 2001. 1

[120] C. Pascal and D. Panescu. A Colored Petri Net model for DisCSP algorithms. *Concurr. Comput. Pract. Exp.*, 29(18):1–23, 2017. 5.1.1

[121] T. K. Paul and M. F. Lau. A Systematic Literature Review on Modified Condition and Decision Coverage. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, SAC '14, page 1301–1308, New York, NY, USA, 2014. Association for Computing Machinery. 2.2.1

[122] J. Paul C. *Software Testing A Craftsman's Approach*. Software Paradigms. Auerbach Publication, 2002, Michigan, 2002. 1.1.1, 1.1, 1.1.4

[123] J. Paul C. *The Craft of Model-based Testing*. CRC Press. CRC Press, 2017. 1.1.5

[124] F. Pothon. DO-178C/ED-12C versus DO-178B/ED-12B: Changes and Improvements. Technical report, AdaCore, 2012. 1, 1.1.7, 2.1, 2.1, 2.2, 2.2.1, 3, 3.1.2, 5.2, 7.1

[125] T. Preußer and A. Weiss. The CEDARtools platform - massive external memory with high bandwidth and low latency under fine-granular random access patterns. In I. Sourdis, C. Bouganis, C. Álvarez, L. A. T. Díaz, P. Valero-Lara, and X. Martorell, editors, *29th Intl. Conf. on Field Programmable Logic and Applications, FPL 2019*, pages 426–427. IEEE, 2019. 1, 2, 4.1.1, 4.2, 7.2.2

[126] T. B. Preußer, S. Gautham, A. D. Rajagopala, C. R. Elks, and A. Weiss. Everything You Always Wanted to Know About Embedded Trace. *Computer*, 55(2):34–43, 2022. 3.2

[127] Rapita Systems. RapiCover: Low-overhead coverage analysis for critical software. available at https://www.rapitasystems.com/products/rapicover. 3.2

[128] S. Reda and A. M. Salem. Combinational equivalence checking using Boolean satisfiability and binary decision diagrams. In *Design, Automation and Test in Europe. Conference and Exhibition (DATE)*, pages 122–126. IEEE, 2001. 2.7

[129] J. Reinders. Processor Tracing, Jly, 2019. 1

[130] W. Reisig. *Elements of distributed algorithms: modeling and analysis with Petri nets*. Springer Science & Business Media, 1998. 2.6

[131] W. Reisig. *Petri nets: an introduction*, volume 4. Springer Science & Business Media, 2012. 2.6

[132] L. Rierson. *Developing safety-critical software: a practical guide for aviation software and DO-178C compliance*. CRC Press, 2013. 1.1.6, 1.2, 2.1, 2.1, 5.2

[133] A. Rodríguez, L. M. Kristensen, and A. Rutle. Formal Modelling and Incremental Verification of the MQTT IoT Protocol. In *Proc. of Trans. Petri Nets and Other Models of Concurrency*, volume 11790 of *Lecture Notes in Computer Science*, pages 126–145, Berlin, Heidelberg, 2019. 2.6, 5.1.1

[134] R. Rudell. Dynamic Variable Ordering for Ordered Binary Decision Diagrams. In *Proceedings of the 1993 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '93, pages 42–47, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press. 2.7

[135] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997. 2.4, 2.4, 4.2

[136] W. Schutz. A test strategy for the distributed real-time system MARS. *COMPEURO'90: Proceedings of the 1990 IEEE International Conference on Computer Systems and Software Engineering@m_Systems Engineering Aspects of Complex Computerized Systems*, pages 20–27, 1990. 2.3.2

[137] K. Serebryany, A. Potapenko, T. Iskhodzhanov, and D. Vyukov. Dynamic race detection with LLVM compiler - compile-time instrumentation for ThreadSanitizer. In S. Khurshid and K. Sen, editors, *2nd Intl. Conf. on Runtime Verification, RV 2011*, volume 7186 of *Lecture Notes in Computer Science*, pages 110–114. Springer, 2011. 2.4, 4.1.2, 4.2, 7.2.2

[138] B. C. F. Silva, G. Carvalho, and A. Sampaio. Test Case Generation from Natural Language Requirements Using CPN Simulation. In *SBMF*, 2015. 5.1.1

[139] Simulink. Types of Model Coverage. Accessed 06 March 2020. 5.2

[140] A. Simão, S. Do, S. Souza, and J. Maldonado. A family of coverage testing criteria for Coloured Petri Nets. In *Proc. of 17th Brazilian Symposium on Software Engineering (SBES'2003)*, pages 209–224, 2003. 5.2

[141] R. K. Singh, P. Chandra, and Y. Singh. An Evaluation of Boolean Expression Testing Techniques. *SIGSOFT Softw. Eng. Notes*, 31(5):1–6, sep 2006. 2.2.2, 2.2.3

[142] N. Sinha. The BDD Library. available at https://www.cs.cmu.edu/~modelcheck/bdd.html. 2.7.2

[143] S. K. Sinha and S. L. Tripathi. BDD Based Logic Synthesis and Optimization for Low Power Comparator Circuit. In *International Conference on Intelligent Circuits and Systems (ICICS)*, pages 37–41, Berlin, Heidelberg, 2018". IEEE Computer Society. 2.7

[144] Sudipto Ghosh, R. France, C. Braganza, Nilesh Kawane, A. Andrews, and Orest Pilskalns. Test adequacy assessment for UML design model testing. In *Proc. of 14th Intl. Symp. on Software Reliability Engineering, ISSRE'03.*, pages 332–343, 2003. 5.2

[145] G. Tassey. The economic impacts of inadequate infrastructure for software testing, 2002. 6

[146] Testwell. Testwell CTC++: Test Coverage Analyzer for C/C++. available at http://www.testwell.fi/ctcdesc.html. 3.2

[147] J. Thalheim, P. Bhatotia, and C. Fetzer. INSPECTOR: Data Provenance Using Intel Processor Trace (PT). In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, pages 25–34, June 2016. 2.5.2

[148] The Clang Team. Matching the Clang AST. Website, 2017. https://clang.llvm.org/docs/LibASTMatchers.html. 3.1.1, 3.1.1

[149] M. Tofte. Standard ML language. *Scholarpedia*, 4(2):7515, 2009. 2.6

[150] A. Trujillo and A. Stuchlik. Reviewing Coverage Information. Parasoft C++test documentation. available at https://docs.parasoft.com/display/CPPDESKE1033/Reviewing+Coverage+Information. 3.2

[151] Vector Software. VectorCAST/MCDC. available at https://www.vectorcast.com/software-testing-products/embedded-mcdc-unit-testing. 3.2

[152] S. Vilkomir and J. Bowen. Reinforced Condition/Decision Coverage (RC/DC): A New Criterion for Software Testing. In *Proc. of ZB 2002:Formal Specification and Development in Z and B.*, volume 2272 of *Lecture Notes in Computer Science*, pages 291–308. Springer, 2002. 2.1

[153] R. Wang, C. Artho, L. M. Kristensen, and V. Stolz. Visualization and abstractions for execution paths in model-based software testing. In W. Ahrendt and S. L. T. Tarifa, editors, *Proc. of Integrated Formal Methods - 15th International Conference (IFM 2019)*, volume 11918 of *Lecture Notes in Computer Science*, pages 474–492, 2019. 2.6, 7.3

[154] R. Wang, L. M. Kristensen, H. Meling, and V. Stolz. Automated test case generation for the Paxos single-decree protocol using a Coloured Petri Net model. *J. Logical and Algebraic Methods in Programming*, 104:254–273, 2019. 2.6, 5.1.1

[155] C. Watterson and D. Heffernan. Runtime verification and monitoring of embedded systems. *IET software*, 1(5):172–179, 2007. 4.2

[156] E. Weyuker, T. Goradia, and A. Singh. Automatically generating test data from a Boolean specification. *IEEE Transactions on Software Engineering*, 20(5):353–363, 1994. 4, 6.1.3, 7.2.4

[157] E. J. Weyuker, S. N. Weiss, and D. Hamlet. Comparison of Program Testing Strategies. In *Symposium on Testing, Analysis, and Verification (TAV)*, pages 1–10. ACM, 1991. 1.2

[158] C. William. Determining whether an application has poor cache performance, 2014. 3.1.2

[159] T. Wu, J. Yan, and J. Zhang. Automatic test data generation for unit testing to achieve MC/DC criterion. In *International Conference on Software Security and Reliability (SERE)*, pages 118–126. IEEE Computer Society, 2014. 6.2

[160] D. Xu, W. Xu, M. Kent, L. Thomas, and L. Wang. An automated test generation technique for software quality assurance. *IEEE Reliab.*, 64(1):247–268, 2015. 5.2

[161] L. Yang, J. Yan, and J. Zhang. Generating minimal test set satisfying MC/DC criterion via SAT based approach. In *Annual ACM Symposium on Applied Computing (SAC)*, pages 1899–1906. ACM, 2018. 4, 6.1.3, 6.2, 7.2.4

[162] Q. Yang, J. J. Li, and D. Weiss. A Survey of Coverage Based Testing Tools. In *Proceedings of the 2006 International Workshop on Automation of Software Test*, AST '06, pages 99–103, New York, NY, USA, 2006. ACM. 3.2

[163] Y. T. Yu and M. F. Lau. A comparison of MC/DC, MUMCUT and several other coverage criteria for logical decisions. *Journal of Systems and Software*, 79(5):577–590, 2006. Quality Software. 2.1, 2.1, 2.1, 5, 6, 2.2.1, 1

[164] Z. Yu, Z. Yang, X. Su, and P. Ma. Evaluation and comparison of ten data race detection techniques. *International Journal of High Performance Computing and Networking*, 10(4-5):279–288, 2017. 4.2

[165] J. Zander, I. K. Schieferdecker, and P. J. Mosterman. *Model-Based Testing for Embedded Systems*. CRC Press. CRC Press, 2017. 1.1.4

[166] C. Zhang, Y. Yan, H. Zhou, Y. Yao, K. Wu, T. Su, W. Miao, and G. Pu. Smartunit: Empirical Evaluations for Automated Unit Testing of Embedded Software in Industry. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '18, pages 296–305, New York, NY, USA, 2018. ACM. 3.2

[167] H. Zhu, P. A. V. Hall, and J. H. R. May. Software Unit Test Coverage and Adequacy. *ACM Comput. Surv.*, 29(4):366–427, Dec. 1997. 1.1.7, 2.1

Part II

ARTICLES

PAPER A

NON-INTRUSIVE MC/DC MEASUREMENT BASED ON TRACES

Faustin Ahishakiye, Svetlana Jakšić, Felix Dino Lange, Volker Stolz, Malte Schmitz, Daniel Thoma

In Proceedings of the 13th International Symposium on Theoretical Aspects of Software Engineering (TASE), **IEEE**, Guilin, China, pages 86-92, Jul. 2019, https://doi.org/10.1109/TASE.2019.00-15.

Non-intrusive MC/DC Measurement based on Traces

Faustin Ahishakiye, Svetlana Jakšić, Volker Stolz
Department of Computing, Mathematics and Physics
Western Norway University of Applied Sciences
Bergen, Norway
firstname.lastname@hvl.no

Felix D. Lange, Malte Schmitz, Daniel Thoma
Institute for Software Engineering
and Programming Languages
University of Lübeck
Lübeck, Germany
lastname@isp.uni-luebeck.de

***Abstract*—We present a novel, non-intrusive approach to MC/DC coverage measurement using modern processor-based tracing facilities. Our approach does not require recompilation or instrumentation of the software under test.**

Instead, we use the Intel Processor Trace (Intel PT) facility present on modern Intel CPUs. Our tooling consists of the following parts: a frontend that detects so-called decisions (Boolean expressions) that are used in conditionals in C source code, a mapping from conditional jumps in the object code back to those decisions, and an analysis that computes satisfaction of the MC/DC coverage relation on those decisions from an execution trace. This analysis takes as input a stream of instruction addresses decoded from Intel PT trace data, which was recorded while running the software under test. We describe our architecture and discuss limitations and future work.

***Keywords*-Code coverage, MC/DC, Software testing, Software verification**

I. Introduction and Motivation

In order to prevent disastrous events, certification standards, for example the DO-178C [1] in the domain of avionic software systems, are used by certification authorities, like the Federal Aviation Administration (FAA) and the European Aviation Safety Agency (EASA), to approve safety-critical software and ensure that the software used in the systems follows certain software engineering standards. DO-178C requires that structural coverage analysis is performed during the verification process mainly as a completion criterion for the testing effort and to identify design faults as well as finding dead code.

Software with the highest safety level (Level A) in avionics systems is required to show modified condition decision coverage (MC/DC) [2]. Unlike weaker coverage criteria, MC/DC is sensitive to the complexity of decisions, because every condition in each decision has to show its independent effect on the decision's outcome.

Usually MC/DC is measured by instrumenting the source code (see III) in order to observe information about taken paths, executed statements and evaluated conditions. Instrumentation is intrusive (it may change characteristics like memory consumption, affect the cache and scheduling) and it is necessary for certification purposes to show that the behavior of the code does not change after the coverage is measured and the instrumentation is removed. Alternatively it is possible to leave the instrumentation in the release code but that consumes resources which are especially valuable in embedded systems, which are widely used in the domain of safety-critical systems.

We present an approach how MC/DC can be measured non-intrusively by analyzing program traces. Our novel approach is based on the idea that every condition in the source code is translated into a conditional jump on the object code level. We first record the trace of an executing program and then analyze it offline [3]. Program traces contain information about taken jumps during the execution and make it possible to reconstruct the evaluation of each condition without instrumentation.

The rest of this paper is organized as follows: Section II introduces coverage criteria of safety-critical software. An overview of state-of-the-art solutions is given in Section III. Section IV describes Intel Processor Tracing (Intel PT) and trace reconstruction. Section V explains the idea and the implementation of our tool. Section VI presents the experiment setup. Finally, we provide related work (Section VII) and concluding remarks and future work in Section VIII.

II. MC/DC in Context of Safety-Critically Software

Depending on the software safety-level, which is assessed by examining the effects of a failure in the system, different coverage criteria have to be fulfilled during software verification:

Software level C (major effect) requires *statement coverage* and software level B (hazardous effect) requires *decision coverage* [4]. Statement coverage is a relatively weak criterion, because it only requires that every statement has been executed but it is insensitive to control flow. Decision coverage is a fairly stronger criterion because it makes sure that every possible outcome of each decision (e.g. the Boolean expression in an if-then-else) has been executed at least once, and therefore there is no unexpected behavior caused by an unexpected outcome of a decision.

Software Level A (catastrophic effect) requires *modified condition/decision coverage* (MC/DC). The coverage criterion has been chosen as the coverage criterion for the highest

This work was supported in part by the European Horizon 2020 project COEMS under number 732016 and the BMBF project ARAMiS II with funding ID 01 IS 16025.

safety level software because it is sensitive to the complexity of the structure of each decision [2] – a decision is made up of one or more conditions. Compared to even stronger criteria like multiple condition coverage (MCC), that requires every possible combination of all conditions which leads to an exponential growth of the minimum numbers of test cases, MC/DC may be satisfied with only $n + 1$ test cases for a decision with n conditions. The following definition has been provided in the DO-178C [4]:

Definition 1 (Modified condition/decision coverage):

- Every point of entry and exit in the program has been invoked at least once,
- every condition in a decision in the program has taken all possible outcomes at least once,
- every decision in the program has taken all possible outcomes at least once, and
- each condition in a decision has shown to independently affect that decision's outcome by: (1) varying just that condition while holding fixed all other possible conditions, or (2) varying just that condition while holding fixed all other possible conditions that could affect the outcome.

Additionally, the terms *Condition* and *Decision* are defined as:

Definition 2 (Condition): A Boolean expression containing no Boolean operators except for the unary operator (NOT).

Definition 3 (Decision): A Boolean expression composed of conditions and zero or more Boolean operators. If a condition appears more than once in a decision, each occurrence is a distinct condition.

For example, in Figure 1a the if-statement contains a decision `a<5||(b==5&&c>5)`. This decision is composed of three conditions `a<5`, `b==5` and `c>5`.

By showing the independent effect of each condition, MC/DC assures that condition's defined purpose. The most challenging and most discussed part in the definition of MC/DC is showing this independent effect: item (2) in the definition has been introduced in the DO-178C to clarify that the so called *Masked MC/DC* is allowed [1], [5]. Masked MC/DC means that it is sufficient to show the independent effect of a condition by holding fixed only those conditions that could influence the outcome. This is important for programming languages that use short-circuit evaluation, because certain executions of decisions are not distinguishable, if the outcome of the decision is determined before every condition has been evaluated.

III. State-of-the-Art

Today there is a number of testing tools for measuring coverage developed for both industrial use and academic purpose. For instance, a survey conducted in [6] described and compared 17 tools primarily focusing on, but not restricted to, coverage measurement. These tools are focusing on weaker coverage criteria for C, C++ and Java programs. Most of them are used only for code coverage, but some, such as Agitar, Dynamic, JCover, Jtest and Semantic Designs, provide debugging assistance as well.

Currently, there are not so many testing tools which focus on MC/DC measurement. VectorCAST/MCDC [7] is a well established tool for measuring MC/DC coverage for C/C++. The tool supports both unique cause and masking MC/DC analysis. Beside reporting and documenting the results, the tool supports automatic test case generation to quicken the development of a full set of MC/DC test cases. Parasoft C++test [8] is a C/C++ testing tool that is capable of measuring MC/DC. MC/DC is evaluated by calculating the ratio of the number of conditions with independence effect and the total number of conditions in all decisions. Testwell CTC++ tool [9] measures line, statement, function, decision, multiple condition, MC/DC and condition coverage for C, C++, Java, and C# on target and on host. The generated report is showing coverage percentage. CodeCover [10] is an open-source, white-box testing tool developed at the University of Stuttgart. It implements the Ludewig term coverage and they claim that it is similar to MC/DC (subsumes MC/DC).

RapiCover [11] analyzes code coverage including MC/DC on-host and on-target. They visualize coverage by folder, file, function and test case, and filter results to highlight missing coverage. All these tools measure MC/DC intrusively by instrumenting the source code..

In [12], SmartUnit tool which supports statement, branch, boundary value and MC/DC coverage is described. They aim at the unit coverage-based testing and automatically generating MC/DC coverage test cases in industry environment. The percentage of MC/DC coverage is calculated as the ratio of covered conditions and the total conditions in the source code. The commercial Lauterbach tool [13] (see Sec. VII) uses a dedicated hardware-interface to transfer tracing data from the system-under-test into the developer's machine for analysis. They support a variety of trace sources, among others also Intel PT, and use it to measure MC/DC in a similar manner as we describe below.

Another alternative, non-intrusive approach is running the system-under-test within an emulator. The QEMU emulator has been used to this end within the Adacore community [14], and in the RTEMS operating system [15]. Through the emulator it is easy to observe the execution of a program on the object code level, very much like through Intel PT that we will present below. The obvious threat to validity is of course how closely the emulator setup can reflect the real system, especially when considering certification.

In the following we propose a novel approach how to measure MC/DC without instrumentation and a tool that implements this approach on a live system without the need for additional hardware.

IV. Trace-based Approach

The main idea of our trace-based approach is that each condition in the source code is translated into a single conditional jump in the object code. If we can accurately trace execution, we will be able to reconstruct the evaluation of conditions along the execution paths. On modern Intel processors, through the IntelPT framework, we are able to unobtrusively record the execution traces of applications. Through operating system support, tracing can be easily enabled for a single application. We first describe the general mode of operation of IntelPT, and continue then with our analysis of the recorded traces.

A. *Intel Processor Tracing (Intel PT)*

Program tracing is an important mechanism for developers in the context of gathering useful information for debugging, monitoring and performance analysis of a program executions.

Intel Processor Tracing (Intel PT) is an extension of the Intel Architecture that traces program execution with low overhead [16]. It can be used by modern Intel CPUs such as Intel Broadwell (5th generation) CPU or better. Intel PT was introduced to provide an accurate and detailed trace with triggering and filtering capabilities [17]. Intel PT works by capturing information about software execution on each hardware thread using dedicated hardware facilities so that after execution completes software can do processing of the captured trace data and reconstruct the exact program flow.

Intel PT uses an extremely compact format that makes it possible to overcome the small bandwidth and limited buffer space by basically only storing information about taken and not taken branches, indirect branches, function returns and interrupts. Based on these the complete program execution flow can be reconstructed. With Intel PT, it is easy to extract and report a much deeper view on loop behavior, from entry and exit down to specific back-edges and loop trip-counts. The traces contain instructions executed by the processor, but there are no data values. For example, for the C-level instruction `x = y + z`, as the trace essentially only consists of instruction pointers, we can only reconstruct the assembly instructions for loading the values, summation and storing the result in memory, and maybe even map them onto the source-code, but we have no information about the actual values of x, y and z or their location in memory during execution.

IntelPT has some drawbacks related to trace file size and speed since the trace bandwidth can exceed 10 Gbits/s. That means that the program trace data and the decoded trace become huge, fast. Recording program executions of more than a few milliseconds requires large and fast writable storage, so that information can be stored quickly enough for offline- or parallel processing, without losing events due to full buffers.

B. *Trace Analysis*

By analyzing program traces it is possible to see if the jumps corresponding to conditionals in the source code have been taken during the execution and to reconstruct how the conditions have been evaluated. If the statement following the conditional jump in the trace equals the target of this jump, the jump has been taken.

Which conditional jumps occur in the object code depends on the condition in the source code. Because the compiler

Nr.	A	B	C	$A \vee (B \wedge C)$
1	**false**	**false**	?	**false**
2	**false**	**true**	**false**	**false**
3	**false**	**true**	**true**	**true**
4	**true**	?	?	**true**

TABLE I: Short-circuit evaluation for $A \vee (B \wedge C)$

```
if (a<5 || (b==5 && c>5)){
  return 1;
}
```

(a) C code with decision containing three conditions.

```
400494: cmpl   $0x5,-0x8(%rbp)
400498: jl     4004b2
40049e: cmpl   $0x5,-0xc(%rbp)
4004a2: jne    4004be
4004a8: cmpl   $0x5,-0x10(%rbp)
4004ac: jle    4004be
4004b2: movl   $0x1,-0x4(%rbp)
```

(b) Object code with three conditional jumps.

Fig. 1: C code and corresponding Object code compiled with clang version 5.0 on default parameters.

sometimes uses the negation of the operator, there are two assembly instruction possible for each relational operator.

From tracing execution in the object code, we can then reconstruct the outcome of an entire decision by analyzing the trace. If the decision statement is followed by the instructions corresponding to the then-branch, the decision has been evaluated as True, otherwise False. Figures 1a and 1b show a decision as part of a C program with three conditions, and the corresponding assembly code (with compiler optimizations disabled). The comparisons ($<, >$ and $==$) are translated by the compiler into small sequences of assembly instructions. Typically these consist of a compare operator (`cmpl`) and a conditional jump (`jl, jne`). This structure makes it possible to map a conditional jump to each condition.

C. *Short Circuit Evaluation*

In C (as in most modern programming languages) short-circuit evaluation is used to evaluate Boolean expressions. That means that the expression is evaluated from left to right and if the left-hand operand of a conjunction is **false** or, respectively, if the left-hand operand of a disjunction is **true**, the right-hand operand is not further evaluated. As mentioned in Section II, Masked MC/DC is accepted by the DO-178C. Because short-circuit evaluation skips exactly those conditions that cannot influence the outcome of a decision, it is possible to measure Masked MC/DC based on traces.

D. *Condition Reconstruction*

The decoded program trace contains information about each executed instruction and therefore whether each jump has been taken or not. This makes it possible to look at each execution of a decision and to note which jumps have been taken. A table can be generated where each row contains one evaluation of a decision and each column contains the assignment of each condition during that evaluation. The last column shows

Relational Operator:	Possible Conditional Jumps:		Condition Value of Detected Jump:
	x86-64	ARM	
no operator	`jne`	`bne`	**True**
	`je`	`beq`	**False**
==	`je`	`beq`	**True**
	`jne`	`bne`	**False**
<	`jl`	`blt`	**True**
	`jge`	`bge`	**False**
<=	`jle`	`ble`	**True**
	`jg`	`bgt`	**False**
>	`jg`	`bgt`	**True**
	`jle`	`ble`	**False**
>=	`jge`	`bge`	**True**
	`jl`	`blt`	**False**

TABLE II: Multiple interpretations of jumps in the x86-64 and ARM instructions sets compiled with clang version 5.0

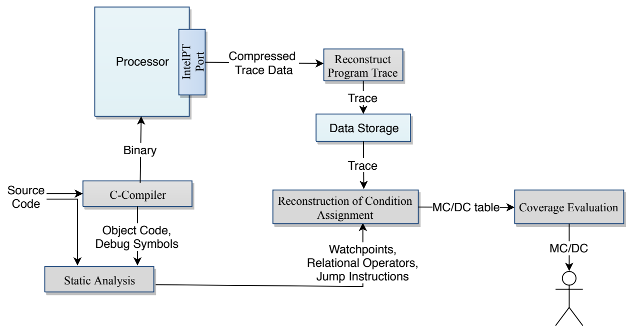

Processor
IntelPT Port
Compressed Trace Data
Reconstruct Program Trace
Trace
Data Storage
Trace
Binary
Source Code
C-Compiler
Object Code, Debug Symbols
Static Analysis
Reconstruction of Condition Assignment
MC/DC table
Coverage Evaluation
Watchpoints, Relational Operators, Jump Instructions
MC/DC

Fig. 2: Overview of the implementation.

the outcome of the decision during the execution. The table has n rows and $m + 1$ columns for a decision that has been executed n times and has m conditions. Table I shows the table for the decision $A \vee (B \wedge C)$ with some example observations/outcomes that satisfy MC/DC (see explanation below).

Because of short-circuit evaluation not all conditions are generally evaluated during one execution and can therefore not be reconstructed by analyzing the trace. In the table these entries are filled as "?".

Depending on the relational operator in the condition (<, <=, ==, etc.) two different possible conditional jumps can be generated by the compiler because conditions can be translated to their negation (it is up to the compiler to choose "jump-if-equal" or "jump-if-not-equal"). If a condition is translated as its negation, this has implications for the reconstruction of the assignments by analyzing the trace as a taken jump shows that the condition has been evaluated as **false**. The possible combinations for the Intel x86-64 instruction set and its ARM counterpart are shown in Table II, which have to be taken into account when the reconstruction is performed. We call the addresses of instructions relevant to our analysis *watch-points* (i.e., conditional jumps and their targets).

E. MC/DC Measurement

After we have recorded all reconstructed condition values in a table per decision, MC/DC can be measured as follows. All rows with a different outcome are compared. If they contain a different entry for exactly one condition, these two assignments show the independent effect for this condition. Two entries for a condition are considered different if one contains a **true** and another one contains **false**. If one of them contains the unknown reconstruction **"?"**, the independent effect of this condition cannot be shown based on these cases.

For the example in Table I with short-circuit evaluation, the independent effect of condition A can only be shown by ignoring the other two conditions, because they cannot influence the outcome, if A is **true**. So executions number 1 and 4 show the independent effect of condition A. The independent effect of B can only be shown, if A is **false** and therefore B is actually evaluated. The outcome of the decision and value of B changes in this example in executions number 1 and 3. Likewise, the independent effect of condition C can be shown with executions number 2 and 3, because the value of condition C and the outcome are changing.

This corresponds as Masked MC/DC and hence complies with the definition of MC/DC in the DO-178C.

We define the measured coverage as the ratio of all decisions satisfying MC/DC and the number of all decisions in the source code.

V. Implementation

An overview of the implementation is provided in Figure 2. The source code is analyzed by our tool in order to detect decisions and their conditions. Additionally, we extract information about their corresponding conditional jumps from the object code. With this information and the program trace provided by Intel PT it is possible to reconstruct the condition assignments and measure MC/DC.

A. Decision Detection with LLVM

In the first step, decisions in the source code have to be found. In order to find decisions in the source code we use the *Abstract Syntax Tree* (AST) representation provided by LLVM. With *LibTooling* and the *AST-matcher* [18] we have built a tool that detects all if-, for- and while-statements in the source code and we gather corresponding information such as line and column numbers and then-statements. We focus on finding traditional branch points (if-, while-, for-statements), but we are aware that certification authorities require other structures, for example assignments containing Boolean expressions, to be covered as well [19].

B. Mapping with Debug Symbols

After the decisions and their conditions in the source code have been detected, debug-symbols are used to map the conditions to conditional jumps in the object code.

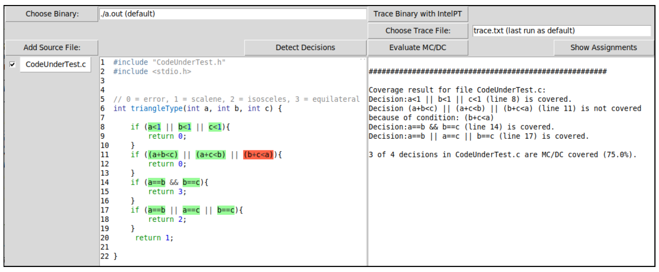
The direct mapping is possible by utilizing debug symbols provided by the compiler. We use *clang 5.0* because this compiler provides rich debug symbols containing line and column information with the compiler option `-g -XClang -dwarf-column-info`. Combined with the detected decisions from the LLVM-tool we then can detect all conditional jumps that are needed for measuring MC/DC based on traces.

Because the outcome of the decision during the execution has to be reconstructed as well, it is necessary to find the *then-statement* which is the statement executed in case of a decision being evaluated as **true**. This statement is also mapped using debug-symbols to its corresponding instruction in the object code.

The result are the decisions, conditions and then-statements in the source code and their translation in the object code.

C. Program Trace Generation

We use Intel Processor Trace (Intel PT) to generate a trace of the execution of a program. The technology is widely available, which makes it suitable for this proof-of-concept tool. With *perf*[1] the Linux-kernel provides an easy-to-use implementation of the recording and reconstruction of Intel PT traces. The reconstructed traces become quiet large even for short execution times. To reduce the size, we filter the trace against the watch-points and only store those parts of the trace that are relevant for measuring MC/DC.



```
1  #include "CodeUnderTest.h"
2  #include <stdio.h>
3
4
5  // 0 = error, 1 = scalene, 2 = isosceles, 3 = equilateral
6  int triangleType(int a, int b, int c) {
7
8      if (a<1 || b<1 || c<1){
9          return 0;
10     }
11     if ((a+b<c) || (a+c<b) || (b+c<a)){
12         return 0;
13     }
14     if (a==b && b==c){
15         return 3;
16     }
17     if (a==b || a==c || b==c){
18         return 2;
19     }
20      return 1;
21
22 }
```

```
########################################################

Coverage result for file CodeUnderTest.c:
Decision:a<1 || b<1 || c<1 (line 8) is covered.
Decision (a+b<c) || (a+c<b) || (b+c<a) (line 11) is not covered
because of condition: (b+c<a)
Decision:a==b && b==c (line 14) is covered.
Decision:a==b || a==c || b==c (line 17) is covered.

3 of 4 decisions in CodeUnderTest.c are MC/DC covered (75.0%).
```

Fig. 3: Screenshot of the GUI.

D. Graphical User Interface

The tool chain of detecting all decisions, mapping conditions to conditional jumps, running and tracing the program and measuring MC/DC based on the trace can be used via a graphical user interface (GUI, see Figure 3) or through the command line as described in Section VI. Via the GUI, we show the detected decisions and the measured coverage in the source code, allowing the user to directly see which conditions are not covered. This should help developers in finding new test cases that cover the missing combinations.

A typical workflow with our tool is the following:

1) *Choose Binary* opens a file dialog and the binary can be selected.
2) *Add Source File* opens a file dialog and source files can be added for which MC/DC should be measured.
3) The source files are listed and can be viewed by clicking on them.
4) *Detect Decisions* detects the decisions in the selected source code and maps the conditions to their corresponding conditional jumps.

[1]https://perf.wiki.kernel.org

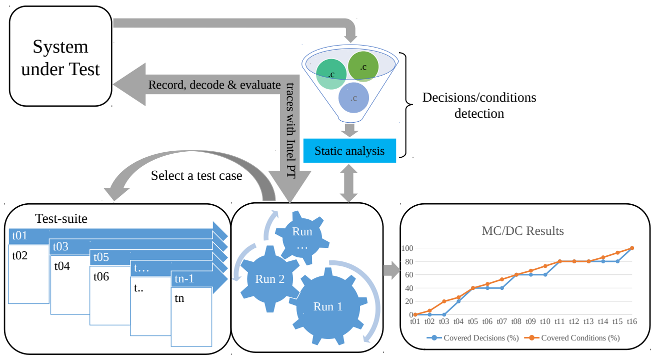


Fig. 4: MC/DC measurement experiment setup.

5) *Trace Binary with Intel PT* calls Linux' *perf* and saves the trace in the file chosen in *Choose Trace File*.
6) *Evaluate MC/DC* analyzes the recorded trace and measures MC/DC of the detected decisions. The result is shown directly in the source code.
7) *Show Assignments* opens a new window containing an overview of all detected decisions, conditions and their reconstructed values.

The tool is available for academic evaluation purposes[2]. On the website you can find an example application and a trace recorded with Intel PT, which can be analyzed with the tool.

VI. Experimental Setup

Our experimental setup for MC/DC coverage measurement consists of two examples as C code, together with their unit tests. The function in the first unit has four decisions (if-statements), containing in total eleven conditions. The second unit contains one decision (also an if-statement) with three conditions. The entire test suite contains 16 test cases. Our tooling allows us to execute the entire test suite and measure coverage, or to just run and measure a single test case. The test suite contains twelve test cases for the first unit where MC/DC coverage is achieved with eleven test cases. Note that this is not directly related to the number of $n+1$ test cases before, as the decisions in subsequent if-statements are not independent. The second unit has four test cases, and all four test cases need to be executed to achieve MC/DC coverage. In addition to the use of our tool via GUI as described in Section V, in this section we set up our experiment for MC/DC measurement via the command line on a Linux OS. After the compilation of the program under test, we conduct the experiment in the following steps as shown in Figure 4:

First, we conduct a static analysis in order to find out which conditions in the source code correspond to which conditional jumps in the object code. The static analysis results in a JSON file with all information related to decisions and conditions and their location (line and column), as well as their mapping to the object code. That is, conditions are mapped to addresses and conditional jumps in the object code. This mapping is necessary because MC/DC is a criterion that is defined on

[2]https://www.coems.eu/mc-dc/

the source code level and there are no equivalent metrics defined on the object code level. In other words, we ignore conditional jumps in the object code that do not directly come from conditionals in the source code. With this information, it is possible to reconstruct the assignment of the conditions during an execution by analyzing the performed jumps and inferring if a condition has been evaluated as true or false. If the program address following a jump instruction in the trace equals the target address that is recorded in the conditional jump instruction, that jump has been performed, otherwise it has not. We use this information to reconstruct the assignment of the condition.

Secondly, we created a wrapper that allows to easily run one particular test from the command line. For each particular test, we record and decode the trace using Intel PT, and we incrementally evaluate the trace with respect to previous results, measure MC/DC and query the MC/DC results. We track the percentage of MC/DC coverage that is achieved through the incremental runs. The tool iterates randomly through the test cases, selecting one at a time and it stops once 100% MC/DC coverage is achieved, otherwise it continues picking other test cases, i.e., we run a test case at most once.

Finally, the tool reports the MC/DC result with the set of test cases that have been executed. From the recorded data, it is easy to plot curves as to which test case contributes to decision or condition coverage. Note that we are not replacing the unit tests, but rather see this as a way to minimize testing overhead: in practice, one would suggest a run of all unit tests without measuring MC/DC, and having occasional runs with tracing enabled that verify that a known set of test cases achieves a predetermined threshold of MC/DC coverage.

VII. Related Work

The interesting discussion on the applicability of MC/DC to software testing for safety-critical systems have been introduced by Chilenski in [2]. Different comparisons for code coverage metrics have been investigated in the context of structure based metrics [20], data-flow metrics [21], decision coverage and MC/DC [22], comparison of multiple condition coverage (MCC) and MC/DC with short-circuit evaluation [23]. MC/DC and object branch coverage (OBC) criteria were compared in [24] and [25]. Even though aforementioned research gives the foundation, none provides a deep MC/DC analysis based on the trace of the program-under-test.

A non-intrusive online monitoring for multi-core systems based on the embedded trace of the system under test is proposed in [26]. Online reconstruction and analysis of debug trace data are based on FPGA and TeSSLa [27]. This combination could be used to implement coverage-calculation on the FPGA, instead of doing it on the host or offline, as in our setting here.

Lauterbach offers the `t32cast` command line tool for MC/DC measurement based on a real-time trace recording, which can analyze the C/C++ source code [13]. The user must ensure that the selected compiler translates each condition in the source code into a conditional jump at the object code level, e.g. by disabling optimizations. In contrast to our approach, which uses features present in most modern Intel processors, the trace data are transferred through a dedicated hardware-connection to a monitor.

VIII. Conclusion

We present a tool that shows the feasibility of measuring MC/DC without instrumentation based on program traces. The tool is able to detect decisions and conditions in C source code and to find their corresponding conditional jumps in the object code. MC/DC can be measured by reconstructing condition assignments based on Intel PT traces.

The advantage of our approach is that there is no need for intrusive software instrumentation. Traditionally, the coverage of the instrumented code is measured, and the instrumentation has to be removed before release, but with our approach it is possible to measure coverage directly on the release code by only using debug symbols that are not altering the behavior of the code and therefore are not considered intrusive.

Our approach of measuring MC/DC based on traces complies with the position of CAST-17, that provides certification authorities' concerns and position regarding the analysis of structural coverage at the object code level [28]. With the mapping between conditions and conditional jumps we provide traceability between source and object code and the reconstruction of condition assignments on the source code level, we can provide the same level of assurance as measuring directly on source code level via software instrumentation.

However there are some limitations. It is necessary to disable optimizations during the compilation because even on low optimization levels conditions are usually not directly translated into conditional jumps but into conditional moves, jump tables or indirect branches [29]. Because regular program traces contain no information how these instructions are evaluated, they cannot be used to reconstruct the evaluation of conditions. This limitation is less severe in the domain of avionic, because other requirements, for example source code to object code traceability in DO-178C, make it already hard for developers to use high optimization levels [30]. Also it is necessary to have a modern compiler like clang version ≥ 5.0 because the DWARF debug symbols need to have column and line information.

Another problem of our approach is that the trace data becomes excessively large for longer executions. Here, we used an offline tracing approach [3], where available storage effectively limits the size of traces. In future work, we want to apply this approach to online trace reconstruction which would enable us to observe much longer execution times because only the very events that are used for coverage measurement are reconstructed. We also want to support other architectures and instead of Intel PT, use tracing technologies such as ARM CoreSight and NEXUS for PowerPC since these processor architectures are widely used in avionics and automotive industry, which would benefit the most from this new approach of MC/DC measurement.

References

[1] F. Pothon, “DO-178C/ED-12C versus DO-178B/ED-12B: Changes and Improvements,” AdaCore, Tech. Rep., 2012, available at https://www.adacore.com/books/do-178c-vs-do-178b.

[2] J. J. Chilenski and S. P. Miller, “Applicability of modified condition/decision coverage to software testing,” *Software Engineering Journal*, vol. 9, no. 5, pp. 193–200, 1994.

[3] F. D. Lange, “Modified Condition/Decision Coverage based on jumps,” 2018, master’s thesis, available at http://www.isp.uni-luebeck.de/thesis/modified-conditiondecision-coverage-based-jumps.

[4] L. Rierson, *Developing Safety-Critical Software: A Practical Guide for Aviation Software and DO-178C Compliance*. CRC Press, 2013.

[5] Certification Authorities Software Team (CAST), “Rationale for Accepting Masking MC/DC in Certification Projects,” *Technical Report: Position Paper CAST-6*, 2001.

[6] Q. Yang, J. J. Li, and D. Weiss, “A Survey of Coverage Based Testing Tools,” in *Proc. of the 2006 Intl. Workshop on Automation of Software Test*, ser. AST ’06. New York, NY, USA: ACM, 2006, pp. 99–103.

[7] Vector Software, “Vectorcast/mcdc,” available at https://www.vectorcast.com/software-testing-products/embedded-mcdc-unit-testing.

[8] A. Trujillo and A. Stuchlik, “Reviewing coverage information,” Parasoft C++test documentation, available at https://docs.parasoft.com/display/CPPDESKE1033/Reviewing+Coverage+Information.

[9] Testwell, “Testwell CTC++: Test Coverage Analyzer for C/C++,” available at http://www.testwell.fi/ctcdesc.html.

[10] CodeCover, “CodeCover: an open-source glass-box testing tool,” available at http://codecover.org/.

[11] Rapita Systems, “RapiCover: Low-overhead coverage analysis for critical software,” available at https://www.rapitasystems.com/products/rapicover.

[12] C. Zhang, Y. Yan, H. Zhou, Y. Yao, K. Wu, T. Su, W. Miao, and G. Pu, “Smartunit: Empirical evaluations for automated unit testing of embedded software in industry,” in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP ’18. New York, NY, USA: ACM, 2018, pp. 296–305. [Online]. Available: http://doi.acm.org/10.1145/3183519.3183554

[13] Lauterbach, “Trace-based MCDC Coverage,” 2018, available at https://www.lauterbach.com/new2018_cov_mcdc.pdf.

[14] M. Bordin, C. Comar, T. Gingold, J. Guitton, O. Hainque, and T. Quinot, “Object and source coverage for critical applications with the COUVERTURE open analysis framework,” in *Proc. of Embedded Real Time Software and Systems Conference (ERTS)*, 2010. [Online]. Available: http://www.open-do.org/wp-content/uploads/2010/06/couverture_ertss2010.pdf

[15] H. Felbinger, J. Sherrill, G. Bloom, and F. Wotawa, “Test suite coverage measurement and reporting for testing an operating system without instrumentation,” in *17th Real-Time Linux Workshop*, 10 2015. [Online]. Available: https://gedare.github.io/pdf/FelShe15A.pdf

[16] A. Kleen, “Cheat sheet for Intel Processor Trace with Linux perf and gdb,” April 2017, available at http://halobates.de/blog/p/410.

[17] J. Thalheim, P. Bhatotia, and C. Fetzer, “INSPECTOR: Data Provenance Using Intel Processor Trace (PT),” in *2016 IEEE 36th Intl. Conf. on Distributed Computing Systems (ICDCS)*, June 2016, pp. 25–34.

[18] The Clang Team, “Matching the Clang AST,” Clang documentation, available at https://clang.llvm.org/docs/LibASTMatchers.html.

[19] Certification Authorities Software Team (CAST), “What is a “Decision” in Application of Modified Condition/Decision Coverage (MC/DC) and Decision Coverage (DC)?” *Technical Report: Position Paper CAST-10*, 2002.

[20] S. C. Ntafos, “A comparison of some structural testing strategies,” *IEEE Transaction on Software Engineering*, vol. 14, no. 6, pp. 868–874, 1988.

[21] L. A. Clarke, A. Podgurski, D. J. Richardson, and S. J. Zeil, “A Comparison of Data Flow Path Selection Criteria,” in *Proc. of the 8th Intl. Conf. on Software Engineering*, ser. ICSE ’85. Los Alamitos, CA, USA: IEEE Computer Society Press, 1985, pp. 244–251.

[22] K. Kapoor and J. Bowen, “Experimental evaluation of the variation in effectiveness for DC, FPC and MC/DC test criteria,” in *2003 International Symposium on Empirical Software Engineering, 2003. ISESE 2003. Proceedings.*, Sept 2003, pp. 185–194.

[23] S. Kandl and S. Chandrashekar, “Reasonability of MC/DC for safety-relevant software implemented in programming languages with short-circuit evaluation,” *Computing*, vol. 97, no. 30, pp. 261–279, Mar 2015.

[24] C. Comar, J. Guitton, O. Hainque, and T. Quinot, “Formalization and Comparison of MCDC and Object Branch Coverage Criteria,” in *Proc. of Embedded Real Time Software and Systems Conference (ERTS)*, 2012.

[25] T. Byun, V. Sharma, S. Rayadurgam, S. McCamant, and M. Heimdahl, “Toward rigorous object-code coverage criteria,” Technical Report, University of Minnesota, MN, USA, Tech. Rep., 2017.

[26] N. Decker, P. Gottschling, C. Hochberger, M. Leucker, T. Scheffel, M. Schmitz, and A. Weiss, “Rapidly Adjustable Non-intrusive Online Monitoring for Multi-core Systems,” in *Formal Methods: Foundations and Applications*, S. Cavalheiro and J. Fiadeiro, Eds. Springer, 2017, pp. 179–196.

[27] N. Decker, B. Dreyer, P. Gottschling, C. Hochberger, A. Lange, M. Leucker, T. Scheffel, S. Wegener, and A. Weiss, “Online analysis of debug trace data for embedded systems,” in *Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2018, pp. 851–856.

[28] Certification Authorities Software Team (CAST), “Structural Coverage of Object Code,” *Technical Report: Position Paper CAST-17*, 2003.

[29] Free Software Foundation, “Options That Control Optimization,” GCC documentation, available at https://gcc.gnu.org/onlinedocs/gcc-5.4.0/gcc/Optimize-Options.html.

[30] Certification Authorities Software Team (CAST), “Guidelines for approving source code to object code traceability, position paper 12,” Certification Authorities Software Team, Tech. Rep., 2003.

PAPER B

HARDWARE-ASSISTED ONLINE DATA RACE DETECTION

Faustin Ahishakiye, José Ignacio Requeno Jarabo, Violet Ka I Pun, Volker Stolz

In Proceedings of the Bartocci E., Falcone Y., Leucker M. (eds) Formal Methods in Outer Space. LNCS, volume 13065, Springer, Cham, Rhodes, Greece, October 2021, https://doi.org/10.1007/978-3-030-87348-6_6.

Hardware-Assisted Online Data Race Detection

Faustin Ahishakiye[1], José Ignacio Requeno Jarabo[1,2], Violet Ka I Pun[1(✉)], and Volker Stolz[1(✉)]

[1] Western Norway University of Applied Sciences, Bergen, Norway
`{fahi,jirj,vpu,vsto}@hvl.no`
[2] Complutense University of Madrid, Madrid, Spain
`jrequeno@ucm.es`

Abstract. Dynamic data race detection techniques usually involve invasive instrumentation that makes it impossible to deploy an executable with such checking in the field, hence making errors difficult to debug and reproduce. This paper shows how to detect data races using the COEMS technology through continuous online monitoring with low-impact instrumentation on a novel FPGA -based external platform for embedded multicore systems. It is used in combination with formal specifications in the high-level stream-based temporal specification language TeSSLa, in which we encode a lockset-based algorithm to indicate potential race conditions. We show how to instantiate a TeSSLa template that is based on the Eraser algorithm, and present a corresponding light-weight instrumentation mechanism that emits necessary observations to the FPGA with low overhead. We illustrate the feasibility of our approach with experimental results on detection of data races on a sample application.

Keywords: Runtime verification · Data race detection · FPGA · Lockset algorithm

1 Introduction

Data races occur in multi-threaded programs when two or more threads access the same memory location concurrently, with at least one write access, and the threads are not using any exclusive locks to control their accesses to that location. They are usually difficult to detect using tests as they depend on the interleaving and scheduling of tasks at runtime. Static analysis techniques frequently suffer from false positives due to over-abstraction, although precise results for source code written in a particular style is certainly feasible. We do not want to discount this field and recent advances, but focus on dynamic techniques for the present occasion.

Races and other concurrency issues have featured prominently in area of Runtime Verification (RV), where precise formal specifications are used at runtime to monitor, and possibly influence, a running system (as opposed to static verification). Our guest of honor (see [12] for an extensive account of his work)

E. Bartocci et al. (Eds.): Havelund Festschrift, LNCS 13065, pp. 108–126, 2021.
https://doi.org/10.1007/978-3-030-87348-6_6

has been one of the founders of the RV workshop- and conference series, and has indeed contributed to the study of the dynamic nature of races with a contribution to the very first workshops [13,14], which has since withstood the test of time [15]. His exploits—which indicate that he is rather aiming for a marathon than a sprint in the race to formal verification—did not stop there: Together with Artho and Biere, he lifted the abstract formal concepts into a practical software engineering setting, where, although race-free in the original sense of the definition, they captured patterns that indicate flawed access to data structures [3]. Source-code instrumentation is one of the go-to solutions to inject RV mechanisms into existing software [10]. His further research with Bodden in a similar direction resulted in the suggestion of a new feature for aspect-orientated programming, a technique for manipulating programs on a higher level, that would facilitate better addressing of concurrency concerns [6].

Although runtime checking only give a limited view on the behaviour of the concretely executed code, it allows precise reporting of actual occurrences, which can be used to predict potential erroneous behaviour across different runs [16]. However, this runtime analysis is not enabled in the final product: inline dynamic data race detection techniques come with invasive instrumentation for each memory access that makes it prohibitive to deploy an executable with such checking in the field [22]. This also makes reproducing errors challenging.

In this article, we take earlier RV attempts for race analysis further and present a *non-intrusive* approach to monitoring applications on embedded system-on-chips (SoCs) for data races using the COEMS platform [8] which aims to eliminate the overhead of dynamic checking by offloading it to external hardware.[1] The platform offers control-flow reconstruction from processor-traces (here: the Arm CoreSight control-flow trace), and data-traces through explicit instrumentation [26]. Race checking is executed on an FPGA on a separate hardware-platform to minimize impact on the system under observation. Our experimental results show that the necessary instrumentation in the target application incurs the expected *fixed, predictable* overhead, and is not affected by the time required for race checking. Our use of the high-level stream-based temporal specification language, TeSSLa [21], means that the reconfiguration of the monitor is substantially faster than synthesising VHDL (few seconds vs. dozens of minutes), and allows end-users to customize the race checker specification to their needs without being FPGA -experts.

This is not possible with other specification-based approaches that directly aim to use the integrated FPGA of a SoC. These approaches do not offer the quick reconfiguration possible with the COEMS platform but require full time-consuming reconfiguration, and do not support the use of control-flow tracing due to the limited capacity of the SoC.

The approach proposed in this paper is more flexible than a dedicated race checker implemented on the FPGA : to the best of our knowledge, such a general solution does not exist, though it is of course in principle possible. It would

[1] The EU Horizon 2020 project "COEMS–Continuous Observation of Embedded Multicore Systems", https://www.coems.eu.

not offer the end-user flexibility in terms of fast reconfiguration that we gain through TeSSLa, and, again due to the space restrictions on SoCs, would not be able to benefit from control-flow tracing features that are important for future optimisations and integration with our analyses.

Our *hardware*-based approach can be used in safety-critical systems such as the aerospace and railway-domains where certification is necessary. In these domains, using a software inline race-checker such as ThreadSanitizer [28] is not possible as the tooling for instrumentation and online race-checking is not certified for those systems, if it even exists. For example, ThreadSanitizer support for Arm32 SoCs is not part of the LLVM toolchain [23]. In contrast to software-based approaches, our instrumentation for the application under test has straightforward complexity and gives predictable performance overhead independent from whether or not race checking is enabled. This is especially important for software development in these safety-critical domains, as again for certification purposes, it is not permissible to, e.g., deploy a separate version for debugging or troubleshooting on demand in the field. Any debugging and trace support must be already integrated in the final product.

The COEMS FPGA requires a compiled monitor-configuration. As this configuration needs to be generated for a specific binary under test, we present in this paper our approach where we instantiate a template that monitors a fixed number of memory locations for consistent access through a fixed number of locks. Although these numbers need to be determined before starting the monitoring, the flexibility of TeSSLa allows us to also deal with an unbounded number of threads, and limited monitoring of dynamically allocated memory and locks. Additionally, our instrumentation supports recording traces in files, and offline analysis of execution traces with the TeSSLa interpreter only. This aids in quick prototyping of new specifications on vanilla developer machines without replicating a full setup of SoC and COEMS hardware.

The paper is structured as follows. After this introduction, Sect. 2 explains the related work. Section 3 details the data race detection in the COEMS framework. Section 4 illustrates the feasibility of our approach and presents performance data on detecting data race errors in a Linux `pthreads`-based case study. Experimental results and software are published in public repositories. Finally, Sect. 5 gathers the conclusions and future work.

2 Our Approach and Related Work

Traditionally, data races have been approached from two sides: *static analyses* check the source code and report potential errors. To that end, over-approximations of program behaviours are used (e.g., in terms of variable accesses and lock operations), which may lead to uncertainties on whether a particular behaviour will actually occur during runtime due to general issues on decidability. This frequently generates too many warnings of potential problems for developers to be useful. These uncertainties can be minimised if decisions such as the number of threads to spawn are fixed at compile time. Static analysers may also have limited support for particular language features.

In contrast to checking the code before it runs, *dynamic analyses* look at individual executions of a program. Although this can only analyse the behaviour of the concretely executed code, it can accurately identify the actual occurrences of defects. These can then be traced back to the buggy code that resulted in the potentially erroneous behaviour. Both techniques in general rely on the availability of the source code, and, in the case of dynamic analyses, the possibility of recompilation with additional instrumentation.

As dynamic analyses for data race detection need to record historic behaviour during execution, they often interfere in terms of computation time and memory consumption. For example, the popular dynamic ThreadSanitizer integrated with the LLVM compiler toolchain slows down executions by a factor of 10 to 100, depending on the workload [28,30]. This is one reason why dynamic analysers are traditionally only employed during development and testing, but not included on the production system [29].

The COEMS project developed a hardware-based solution, in which a *field-programmable gate array* (*FPGA*) checks the execution trace in parallel to the running system with minimal interference. The hardware is adapted for analysing events described in the stream-based specification language TeSSLa [21]. Using an intermediate specification language that is executable on the FPGA can avoid the time-consuming re-synthesisation of the FPGA when changing specifications.

We have ported the gist of the Eraser algorithm [27] to the subset of the TeSSLa language that is supported on the hardware. An alternative approach already used the TeSSLa-interpreter, but was not suitable for compilation onto an FPGA due to the dynamic data structures (sets and maps) that only the interpreter offers [19].

Firstly, we adapt the software-based analysis, which relied on dynamic data structures such as sets and maps in the TeSSLa interpreter, to the hardware-specific implementation of the COEMS trace box (see Sect. 3 for details). As the complete specification does not fit onto the FPGA, we then split the specification into two parts: the performance-relevant portion of the TeSSLa specification is processed on the FPGA (filtering accesses), and the final tracking of which lock protects which memory is done in the interpreter which receives the intermediate output from the FPGA. Additionally, we also allow monitoring of dynamically allocated memory and locks.

The Eraser algorithm is certainly no longer the state of the art in dynamic race detection (or rather, checking locking discipline), but has the advantage that it can be captured in a state machine that is instantiated per memory location and set of locks. Even though it conceptually uses sets, assuming that the number of used memory locations and locks is statically known, we can statically derive the necessary streams.

Such a static encoding should be possible also for the more modern FastTrack-algorithm by Flanagan and Freund [11], which uses lightweight vector clocks and the happens-before relation to avoid false positives. Their article includes a detailed description of the necessary data structures, and uses thread-ids as offset into arrays. Our approach here requires focusing on a fixed number of memory locations and locks, but can deal with an arbitrary number of threads. We leave an encoding into TeSSLa of FastTrack to future work—for a statically decidable set of threads it should certainly be possible, with the necessary vector-clocks also being maintained on the FPGA-side.

An observation-based race checker that tracks memory accesses and lock operations can also be implemented through the help of virtualisation. Gem5 [5, 17] is such a framework. Virtualisation means on the one hand that observation cannot be done on a deployed system in the field but only in the lab and with a limited number of supported peripherals. On the other hand, control-flow events can easily be explicitly generated, no expensive reconstruction is necessary: in full virtualisation, we can directly match on any assembly instruction, and not only branches like with the COEMS hardware. In fact, in such a scenario, it would be straightforward to use Gem5 as event source, where the virtualisation sends events on to a TeSSLa interpreter checking the trace against our specification. Gem5 does not directly offer a high-level specification language for monitoring. Given the high event rate of observations on memory accesses, we expect a similar performance impact like the one reported for ThreadSanitizer.

Another prominent example where a high-level specification is synthesised into an FPGA is RTLola [4]. This differs from our approach in the following: the specification language puts a stronger emphasis on periodic data than we do with our discrete TeSSLa events. Furthermore, RTLola is synthesized via VHDL onto the FPGA, and hence has a high turn-around time for reconfiguration. Communication between the system-under-test and the verification logic is left open to the user and requires knowledge of VHDL, though of course in principle data events can then be emitted through instrumentation. In contrast to our solution, an RTLola specification cannot benefit from control-flow tracing, since control-flow reconstruction is not available as specification and hence cannot be compiled onto the FPGA, and furthermore would exceed the capacity of current SoCs both in terms of space and execution speed [26]. We leave performance evaluation of RTLola execution for race checking purposes on the FPGA to future work, but note that providing an API for the instrumentation to the monitor requires VHLD-knowledge.

A similar direct approach via hardware-synthesis has been taken for Signal Temporal Logic (STL) [18]. It would certainly be feasible to encode a race checker in STL, but that would not be playing to STL's strength in terms of timing properties (which are not relevant for race checking) and observing signals on a wire (as opposed to a programmable interface to send values from the instrumented code to the monitor).

The R2U2 [25] monitoring system for unmanned aerial vehicles provides a generic observation component on an SoC. Again, events must be explicitly emitted, and no control-flow reconstruction is available. Similar to our approach, and unlike in RTLola, this component is generic and is parametrised by compiled specifications. R2U2 uses Metric Temporal Logic specifications (MTL), which are very suitable to describe, e.g. timing properties. While it is certainly possible to specify our race checker in MTL, we leave it to future experimental evaluation

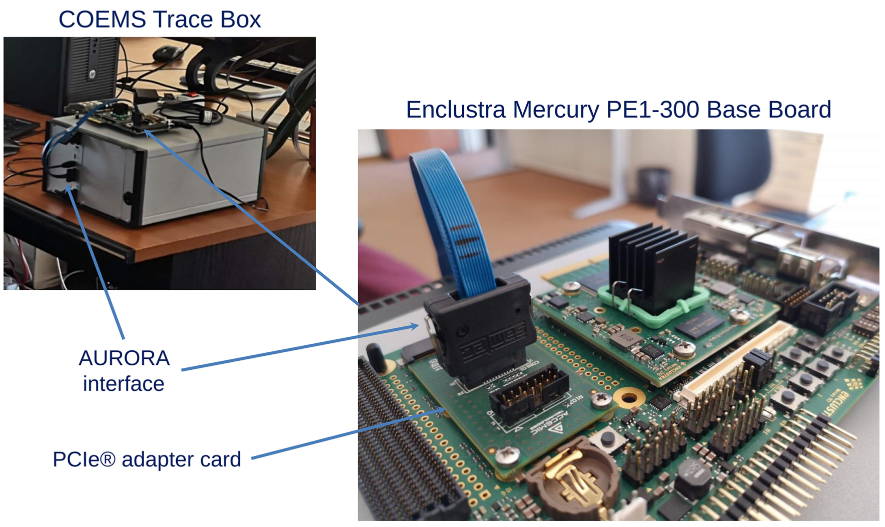


Fig. 1. COEMS trace box containing FPGA (left) and SoC (right)

how many instances of the race pattern (in terms of memory location/protecting lock) would be feasible, and how the communication bus would uphold under varying event rates.

3 Data Race Detection with COEMS

In the following, we first briefly introduce COEMS infrastructure. Then, we describe the workflow of data race detection with the COEMS tools. After that, we explain the idea of the lockset-based Eraser algorithm and our translation into TeSSLa.

3.1 COEMS Infrastructure

The COEMS project provides a novel observer platform for online monitoring of multicore systems. It offers a non-intrusive way to gain insights of the system behaviour, which are crucial for detecting non-deterministic failures caused by, for example, accessing inconsistent data as a result of race conditions.

To observe SoCs, the platform uses the tracing capabilities that are available on many modern multicore processors. Such capabilities provide highly compressed tracing information over a separate tracing port. This information allows the COEMS system to reconstruct the sequence of instructions executed by the processor [26]. The instruction sequence- and data trace can then be analysed online by a reconfigurable monitoring unit. Figure 1 shows the COEMS FPGA enclosure, the Arm-based Enclustra SoC that serves as system under test, and the AURORA interface connecting both.

As soon as the program starts running on the Enclustra board, control flow messages are generated via the Arm CoreSight module and transmitted, together with user-specified data trace messages from any instrumentation, through the AURORA interface to the COEMS trace box. Internally, the Instruction

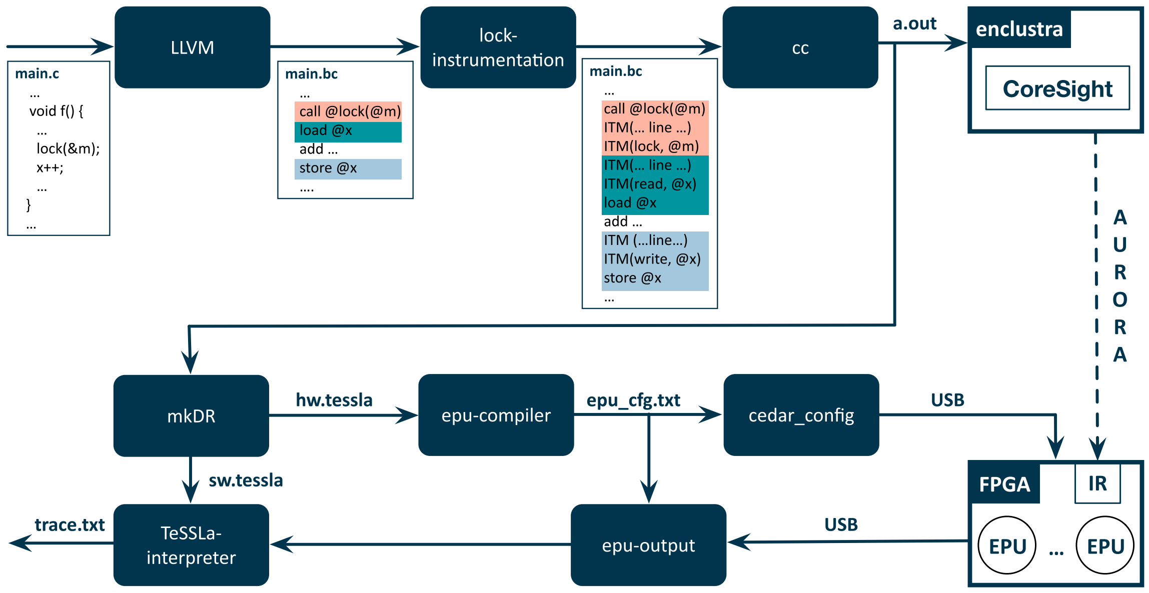


Fig. 2. Lock instrumentation and race monitoring using the COEMS technology

Reconstruction (IR) module reconstructs an accurate execution trace of both cores from the platform-specific compressed format into a stream-based format suitable for analysing properties defined in the TeSSLa language. The flexibility of the TeSSLa language allows expressing different kinds of analyses for functional or timing properties in terms of stream events. A TeSSLa specification is then compiled to a configuration of the Event Processing Units (EPUs) [7] of the monitoring unit, which are specialised units on the FPGA implementing low-level TeSSLa stream operations. The events of the TeSSLa streams are efficiently processed by the EPUs in the trace box. To cope with the potentially massive amount of tracing data generated by the processors, the COEMS system is implemented in hardware using an FPGA-based event processing system. The current COEMS prototype implements eight parallel EPUs.

Compared to existing monitoring approaches, COEMS provides several advantages, most notably is its non-intrusive method to observe and verify the actual behaviour of the observed system, i.e., the system behaviour will not be affected by the monitoring. As no trace data has to be stored, systems can be monitored autonomously for extended periods of time. Furthermore, the trace box reports results of a TeSSLa analysis almost immediately as the processing delay introduced by the trace box is negligible. In contrast to other hardware-based runtime verification techniques [9,29], changing the specification does not require circuit synthesis, but only a TeSSLa compilation. Hence, the focus of observation can be changed during runtime by reconfiguring the EPUs quickly.

3.2 Instrumentation and Data Race Monitoring

The current COEMS framework supports data race detection for `pthread` programs that can be recompiled using LLVM. We illustrate the workflow of instrumentation and data race monitoring in Fig. 2.

We first instrument the application under test during compilation, so that the executable emits information to the COEMS trace box at runtime. We insert calls to instrumentation (i) after taking a lock, (ii) before releasing a lock, and (iii) on shared memory accesses with the help of LLVM, which will send the thread-id and observed action. As an optimisation, we use the LLVM analysis framework to only instrument memory accesses to potentially shared memory: through escape-analysis, this can already eliminate instrumentation e.g. on iteration variables of tight loops. Then, we compile and link the instrumented LLVM intermediate code (.bc) into a binary file (a.out). The instrumentation should hence be easy to integrate into existing build-setups.

Secondly, we copy the binary to the system under observation (enclustra) where we will later run it. The mkDR-script instantiates a TeSSLa specification template with the memory addresses and mutexes to be observed, based on the names of global variables. Expert users have the option of more fine-grained control on the instantiation, e.g., to monitor dynamically allocated memory. The specification is tailored to each program; thus, it should be regenerated from the template every time the application is recompiled. The instantiated specification is then split into two halves, as its size exceeds the currently available number of eight EPUs on the prototype hardware. The first half hw.tessla filters the high event rate stream of observations on the FPGA. It is translated by the epu-compiler into a configuration file (epu_cfg.txt), and then uploaded to the FPGA by cedar_config. The second half sw.tessla receives the output of the first stage, and does the final processing on a stream that now has a lower event rate in the TeSSLa interpreter.

Then, we run the binary file, which will automatically start sending trace data to the FPGA. The epu-output tool decodes the FPGA output into a TeSSLa event stream. Note that the behaviour of the application is independent of whether the COEMS FPGA is actually connected or not. If not, trace data is silently discarded, but does not affect the timing of the application.

Finally, we analyse this trace with the second part of the TeSSLa specification (sw.tessla) with the TeSSLa interpreter, which will emit race warnings if necessary. Our data race specification uses mostly data trace events, since we require the addresses of memory and locks, except for a control flow event when `pthread_create` is called and to signal termination of program under test.

In addition to the online (hardware-based) monitoring analysis with the COEMS trace box, the COEMS framework also supports offline (software-based) analysis of execution traces in a personal computer. In the case of software-based analysis, the user only needs the TeSSLa interpreter and the COEMS lock instrumentation tool. Most of the initial steps in Fig. 2 such as the LLVM instrumentation or the instantiation of the TeSSLa template are similar for the software-trace analysis. Instead of compiling the TeSSLa specification for the EPUs, the software TeSSLa interpreter will now run the entire TeSSLa specification for detecting data races on a locally generated software trace-file. Writing the trace data into a file first or piping them into the TeSSLa interpreter has a higher overhead than transmitting them via the AURORA interface.

3.3 Lockset-Based Algorithm in TeSSLa

Conceptually, the algorithm tracks which set of locks is held at every memory access. The current set is intersected with the previous set on a read or write (for simplicity of presentation, we do not distinguish between read- and write accesses, although only read/write and write/write-conflicts are relevant). This defines the alphabet of observations of the algorithm: pairs of reads or writes with a memory address and thread-identifier, and locking or unlocking operations with lock- and thread-identifier. The algorithm initialises the lockset for each memory address with the set containing all locks, and should the intersection ever yield the empty set, then we can conclude that an inconsistent locking discipline has been used. This means that one part of the execution uses no or disjoint locks from another part of the execution when accessing this memory, which hence gives rise to a potential data race if those executions are assumed to be possible concurrently.

As we do not have dynamic memory available to maintain potentially unbounded sets in the TeSSLa-specification on the FPGA, we need to find a static encoding. To achieve this, for each pair of memory location X and lock identifier (also an address) L, we create a boolean stream `protecting_X_with_L` that is initialised to *true*. The set of all these streams for a given X hence models the lockset as a bit-vector. Note that updates are monotone, i.e., once a stream takes the value *false*, it can no longer revert to *true*. If all these streams for a given X carry *false*, we know that no common lock is protecting the current access, and we emit a race warning on the `error_X` stream for that memory location. This encoding means that we need to know the set of all memory locations and all lock identifiers before we configure the FPGA, as we cannot declare new streams dynamically.

On every memory access to X, we check if L is being held by the current thread and update the current value if necessary. We track this through the streams `holding_L`, which carry the identity of the thread currently holding this lock, if any. Again, updating the value on these streams is trivial upon each locking or unlocking operation.

```
in threadid: Events[Int]
in readaddr: Events[Int]
in writeaddr: Events[Int]
in mutexlockaddr: Events[Int]
in mutexunlockaddr: Events[Int]
@FunctionCall("__pthread_create_2_1")
in pcreateid: Events[Unit]
in line: Events[Int]
in dyn_base: Events[Int]
in dyn_lock: Events[Int]
```

Fig. 3. Header of the TeSSLa specification, including all the incoming events from the instrumented code.

```
 1 def lock_0 := filter(mutexlockaddr ==1, mutexlockaddr ==1)
 2 def unlock_0 := filter(mutexunlockaddr ==1, mutexunlockaddr ==1)
 3 def lock_1 := filter(mutexlockaddr ==24808, mutexlockaddr
      ==24808)
 4 def unlock_1 := filter(mutexunlockaddr ==24808, mutexunlockaddr
      ==24808)
 5 def dyn_temp := 4 * (((dyn_lock >> 1) >> 1))
 6 def slot := dyn_lock - dyn_temp
 7 def dyn_lock_0 = filter(dyn_temp, slot == 0)
 8 def lock_4 := filter(mutexlockaddr == dyn_lock_0, mutexlockaddr
      == dyn_lock_0)
 9 def unlock_4 := filter(mutexunlockaddr == dyn_lock_0,
      mutexunlockaddr == dyn_lock_0)
10 def read_0 := filter(readaddr ==24532, readaddr ==24532)
11 def write_0 := filter(writeaddr ==24532, writeaddr ==24532)
12 def access_0 := merge(read_0,write_0)
13 def access_after_pc_0 := on(last(pcreateid,access_0),line)
14 def thread_accessing_0 := last(threadid*32768 + line,
      access_after_pc_0)
15 def holding_0 := default(merge(last(threadid, lock_0), last(-1,
      unlock_0)), -1)

16 def protecting_0_with_0 := detect_change(default(
      thread_accessing_0/32768 == last(holding_0,
      thread_accessing_0), true))
17 def error_0 := on(thread_accessing_0,!(protecting_0_with_0 ||
      protecting_0_with_1 || protecting_0_with_2 ||
      protecting_0_with_3 || protecting_0_with_4))
```

Fig. 4. TeSSLa fragment, where lines 1–15 are from the hardware stage, while lines 16–17 are from the software stage.

Figures 3 and 4 show an excerpt of the resulting TeSSLa specification. It has been created for one dynamic lock and three static locks, and tracks accesses to memory address 24532. Static locks are stored at memory addresses 24808, 24528 and 24560 (only lock 24808 is shown in Fig. 4), while the memory address of the dynamic lock is emitted at runtime. The static lock at address 1 is artificial and is used for the main-thread only. TeSSLa built-in stream operations are emphasised. As input streams, we receive instrumented events on `mutexlockaddr`, `mutexunlockaddr`, `readaddr`, `writeaddr`, `threadid`, `dyn_base` and `dyn_lock`. The `pcreateid` event reduces false-positives by signalling to only start observing memory accesses after additional threads have actually been created. The annotation `@FunctionCall` indicates that this is a control flow-event which is triggered by a function call, and not through instrumentation. The symbol name corresponds to the function `pthread_create` from Linux' system library. To aid in debugging, the instrumentation also emits the current source code `line` number with each event.

Currently, due to the limited availability of EPUs on the prototype FPGA, the specification is actually split into two halves, with the fast address-filtering done on hardware, and only processing the derived information in the coloured lines 16–17 as a post-processing stage in the interpreter. The multiplication and division are binary left- and right-shifts that reduce the amount of events emitted from the hardware stage to the software stage by encoding the line number of an instruction with its event in the unused lower 16 bits, and encoding dynamically allocated locks (see below).

4 Case Study

In this section, we illustrate our approach with a case study, where we simulate a set of bankers sending random amounts of money from one bank account to another [20]. The bankers lock the source and target bank accounts before committing a transaction, so that transfers are protected against data races and deadlocks. We introduce a special case where one banker (id 0) forgets one lock operation and hence, data may get corrupted. Figure 5 shows the core of the example with locks and memory accesses. The complete example, including source code, execution traces and TeSSLa reports, is available at [1].

```
/* get an exclusive lock on both balances before
   updating (there's a problem with this, see below) */
pthread_mutex_lock(transaction_mtx);
if( !DATA_RACE || (DATA_RACE & (id != 0)) ){
    /* In case of DATA_RACE flag is 'on', the thread_id 0
    forgets to lock the accts[from].mtx mutex */
    pthread_mutex_lock(&accts[from].mtx);
                }
pthread_mutex_lock(&accts[to].mtx);
pthread_mutex_unlock(transaction_mtx);
/* Do the actual transfer. */
if (accts[from].balance > 0) {
        payment = 1 + rand_range(accts[from].balance);
        accts[from].balance -= payment;

pthread_mutex_unlock(&accts[to].mtx);
if( !DATA_RACE || (DATA_RACE & (id != 0)) ){
    /* For symmetry -- don't unlock if racy: */
    pthread_mutex_unlock(&accts[from].mtx);
}
```

Fig. 5. Example of incorrect locking

TeSSLa Specification. We instantiate the corresponding COEMS data race template (see [2] for all files used here) using the mkDR-script and the instrumented binary, and the names of all mutexes (`accts[0].mtx`, ...) and shared variables (`accts[0].balance`, ...) as parameters.

The size of the TeSSLa specification for the Eraser algorithm is proportional to the number of locks and shared memory addresses to monitor, and independent from the number of threads. More precisely, the first half hw.tessla grows linearly with respect to both variables.

For each lock `L`, the TeSSLa specification includes the `lock`/`unlock` pairs and `holding_L` (plus an additional stream in the case of dynamic locks). For each memory address `X`, the TeSSLa specification includes a block of five streams (i.e., `read_X ... thread_accessing_X`). Hence, `3L + 5X` streams are generated for the first half, where `L` is the number of locks and `X` the number of memory addresses. Regarding the second half, the sw.tessla file, the number of streams grows proportionally to `(X+1)*L` (i.e., `protecting_X_with_L` plus `error_X`).

As the TeSSLa specification is a text file and the size is constrained by the previous parameters, the instantiation of the TeSSLa template for the data race checking is done almost instantaneously, and compilation of the largest specification for the hardware stage into the FPGA completes in around 5 min (see below for detailed measurements) including uploading the configuration to the FPGA, hence a big advantage over approaches that need to translate via VHDL.

Working with Dynamic Allocations. As we have noted above, the addresses of memory locations and locks need to be available at compile-time of the specification. This limits our approach to statically declared resources in a program. However, it fits our application domain in embedded systems for these SoCs, where development may follow the MISRA guidelines [24], which strongly recommends against using e.g., `malloc`. For more flexibility, we have developed an extension where developers can at runtime send the location of dynamically allocated memory or locks through additional instrumentation.

Conceptually, we parametrise the specification with a placeholder for the argument of the comparison operations above. A write to a particular stream will make the specification use that value in addition to any hard-coded values. A programmer can use the instrumentation to send the address of a (potentially dynamically allocated) lock to the monitor using the function `emit_dynamic_lock_event(const short slot,const pthread_mutex_t* addr)`.

We can encode potentially multiple "slots" into the lower four bits, since these pointers are suitably aligned (see lines 5–8 in Fig. 4). Similarly, the developer can register the base address of dynamically allocated storage for monitoring through `emit_dynamic_addr_event(const uintptr_t base)`. The range of bytes to monitor is given when instantiating the specification.

Due to the size limitation on the FPGA, we currently can only provide this filter for a limited number additional memory addresses or locks. As the number of possible EPUs on the external FPGA increases, these numbers for dynamic allocations should also go up. Note that the number of statically encoded resources underlies different resource constraints, and we report the general numbers below in the performance characteristics.

Running the Experiment. The epu-compiler translates the first part (hw.tessla) into a configuration file (epu_cfg.txt), and then uploads it via USB to the FPGA through cedar_config. The compilation time depends on the number of streams in the TeSSLa specification and, hence, the number of locks and

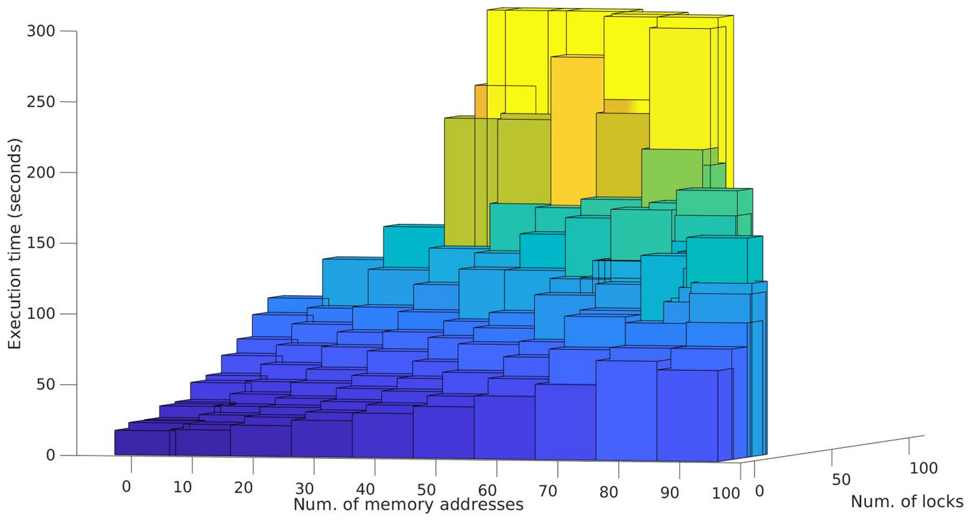


Fig. 6. EPU compilation time for hw.tessla

shared memory addresses in the C code. Figure 6 shows the time required for compiling the TeSSLa specification into the binary configuration format for different scenarios of the bankers example in terms of the number of locks and memory locations. As this involves allocating streams and their operations to the available on-board resources of the FPGA, the following outcomes are possible: (i) successful configuration, (ii) aborted because FPGA-resources have been exceeded, (iii) timeout. Due to the combinatorial growth of the specification, we see that compilation times go up towards the timeout that we have chosen (300 s) for growing numbers of locks/memory locations. After that we reach ranges where it is quickly obvious for the compiler that a configuration cannot be fit onto the FPGA. As compilation is a resource-allocation problem that involves constraint-solving, this slope for compilation time is to be expected: the closer a specification gets to the available resource limits, the harder the constraint-solver has to try searching for a suitable allocation.

After compilation, uploading a new binary configuration into the FPGA after compilation is done in between 3 to 7 s.

The second half of the specification, sw.tessla, receives the output of the first stage, and does the final processing on a stream that now has a lower event rate in the TeSSLa interpreter. Naturally, the interpreter has some startup-cost that also scales with the size of the specification due to parsing and type-checking. Figure 7 shows a similar slope as for the EPU compilation, where startup-time goes up towards the upper right corner, where we also reach up to 300 s.

Since compilation and start-up of the interpreter can be done in parallel and hence lead to the envisioned advantage of quick reconfiguration over approaches going via VHDL.

The COEMS trace box currently supports TeSSLa specifications in the range of hundreds to a few thousands of streams depending on the complexity in the logic of the TeSSLa streams. For our race checker this translates into checking

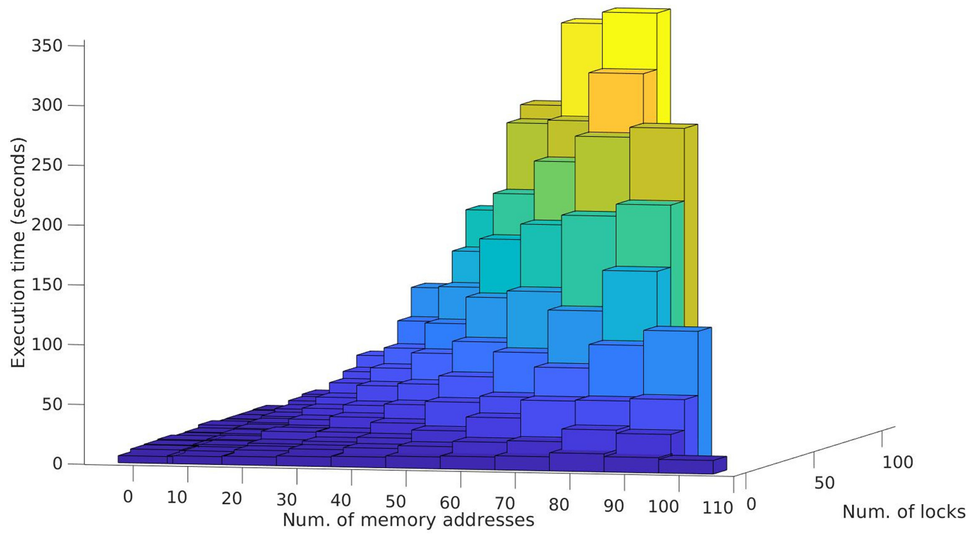


Fig. 7. TeSSLa interpreter startup time for sw.tessla

between around 40 memory locations with 100 locks and 90 memory locations with 20 locks.

```
105923234788: holding_1 = 21653
105923846011: holding_3 = 21653
105923949854: holding_1 = -1
105924265689: holding_1 = 21525
105924621853: thread_accessing_1 = 709525559
105924716727: thread_accessing_1 = 709525559
105924764106: holding_2 = 21525
105924822656: holding_3 = -1
105924872330: holding_1 = -1
105925531285: thread_accessing_0 = 705331255
105925621081: thread_accessing_0 = 705331255
105925674037: holding_1 = 21653
105925731366: holding_2 = -1
105926266497: holding_2 = 21653
```

Fig. 8. Events emitted by the COEMS trace box after processing the hardware half of the TeSSLa specification

```
105847960743: error_0_in_line = 52
105847960743: error_0 = true
105848052196: error_0_in_line = 53
105848052196: error_0 = true
105848316093: error_0_in_line = 54
105848316093: error_0 = true
```

Fig. 9. Race report obtained by processing Fig. 8 including debug information

```
1 for (i = 0; i < N_THREADS; i++)
2       pthread_join(ts[i], NULL);
3 for (total = 0, i = 0; i < N_ACCOUNTS; i++)
4       total += accts[i].balance;
5 printf("Total money in system: %ld\n", total);
```

Fig. 10. Example of false positive after threads have terminated

When we run our C code with the DATA_RACE flag on, the first stage produces the output stream shown in Fig. 8. This stream contains summaries on which thread is holding which lock and accessing any of the selected memory addresses. We then pipe those events into the TeSSLa interpreter with the other half of the specification.

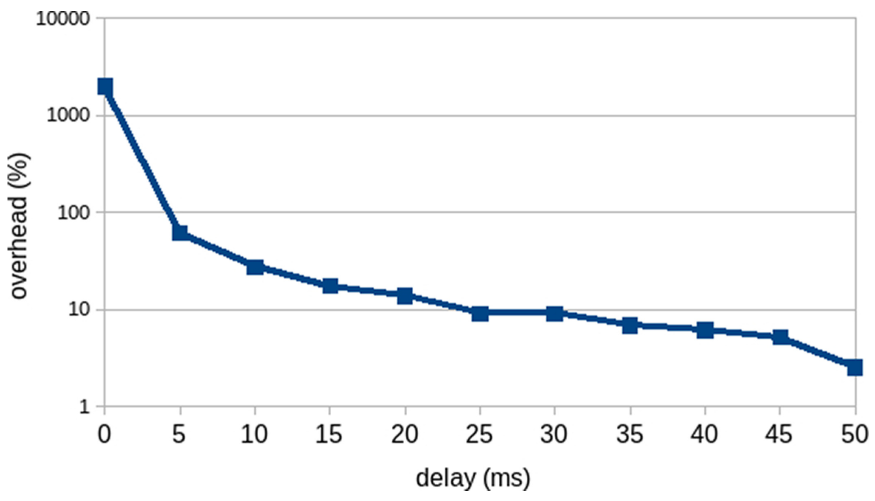


Fig. 11. Overhead introduced by the instrumentation.

The TeSSLa interpreter correctly reports (Fig. 9) the data race errors on the `error_X_in_line` streams, triggered by the accesses in lines 12–15 in Fig. 5 (corresponding to lines 52, 53 and 54 in the source code). These data race errors are caused by the missing lock of mutex `accts[from].mtx` by thread `id_0`. Our tool also detects a false positive in line 4 in Fig. 10 that happens when the `main()` thread accesses the balance in each account once all banker-threads have terminated. The barrier introduced by `pthread_join()` cannot be detected by our lockset-based approach, as it does not actually keep track of the threads in use by the program.

Performance. We have so far only obtained partial performance measurements, as currently our instrumentation needs a global lock to serialize transmission of three events per observation to the FPGA. We transmit the thread-id, the address of the relevant datum, and the line number in the source code to aid in debugging. The lock guards against interleaving between the two cores and context-switches on the same core. This leads to a penalty factor of about 20 in execution time over the original uninstrumented code. At least 50% of that overhead can be attributed to the lock, effectively linearising the above example.

Figure 11 shows the overhead in percent of the instrumented version against the original code. To simulate other workloads with less contention (i.e., larger regions where no race checking is required), we have introduced a configurable `usleep()` instruction between both accesses to the bank accounts. The graph shows that the high overhead is due to the tight loop accessing the accounts, and naturally becomes less prominent as contention goes down.

Another factor affecting overhead is that our instrumentation has not yet been optimised and shares code with the software-tracing for prototyping, which among other things means that even when doing hardware-tracing, additional arguments are passed on the stack that the hardware-tracing does not actually consume. Future improvements in the low level runtime support for data trace events may also bring better performance by allowing larger payloads in a single message, which would allow our instrumentation to avoid the explicit global lock.

We have not yet devised a setup that can measure the performance of evaluating the specification, apart from considerations based purely on the use of EPUs and clock rate of the FPGA: our measurements are currently completely dominated by the USB-interface overhead of polling the output events from the FPGA, and do not allow to precisely factor out the processing time. Additionally, intermediate output from the FPGA to the interpreter is transferred in a verbose ASCII-format.

As for memory consumption, our instrumentation does not need to maintain any data structures, and only passes primitive values such as pointer addresses that are already computed and presumably available in registers anyway.

5 Conclusion

In this paper, we followed the path initiated by our guest of honor in the direction of practical approaches for runtime verification. More in detail, we have shown how to use the COEMS technology, a novel platform for online monitoring of multicore systems, and contextualized it to check for potential data races in applications that use locks for synchronisation, one of our guest's research areas.

Through the COEMS platform, developers can observe the control-flow in a digital twin of their application under test on an embedded systems without affecting the behaviour. Additional instrumentation of the application can send more detailed data at negligible cost.

We presented an outline on how the lockset-based Eraser algorithm can be encoded in the TeSSLa-specification language for a given application. This specification is then compiled onto the external COEMS FPGA and uses the data- and control flow trace emitted from the system under test to observe a specified set of locks and memory locations. As the full specification exceeds the capabilities (in terms of size) of the available prototype, we combine a hardware- and a software stage to report on potential races.

Races may still hide in parts of the code that have not been executed, and our checker may report false positives, which is also a general limitation of tools based on the Eraser algorithm. On the positive side, the COEMS hardware race checker does not negatively affect the performance of the application, so potential users need to carefully assess this tradeoff and structure their code to minimize warnings.

The data race analysis uses the LLVM compiler framework, and currently works with threads using `pthread_mutex_lock/unlock` operations for protecting the shared variables. For other ways of synchronization, e.g., through compare-and-swap instructions, or baremetal execution, we do not provide instrumentation and a template yet, but they can easily be adapted from our code.

A practical limitation of the data trace is the currently restricted value-range of the trace messages to 16 bits, which complicates e.g. the use of pointers in the trace. As currently we need multiple messages per event to transmit additional data such as debugging information (the current line number) and the thread identifier, we need to serialize use of the trace bus. This additional locking that is introduced through the instrumentation affects the performance of the application under test, whereas transmitting a single datum in principle has negligible execution overhead.

Our unoptimised performance measurements already puts us in a competitive range with other approaches such as ThreadSanitizer, and we have the advantage that COEMS-based tracing can remain enabled in production. Future developments of the COEMS platform beyond its current prototype will make splitting the specification and post-processing in the interpreter superfluous: 18 (instead of the currently 8) available EPUs will already allow for setups without dynamic values to be handled completely in hardware. In the meantime, we are improving the instrumentation to produce effect summaries for basic blocks of code instead of instrumenting single instructions, which should decrease the overhead especially for tight loops.

We are also preparing additional concurrency patterns that monitor actual deadlocks and so-called lock-order-reversal.

References

1. Ahishakiye, F., Jarabo, J.I.R., Pun, K.I., Stolz, V.: Open data for banker example, December 2020. https://doi.org/10.5281/zenodo.4381982
2. Ahishakiye, F., Jarabo, J.I.R., Stolz, V.: Lock instrumentation tool (2020). https://github.com/selabhvl/coems-racechecker
3. Artho, C., Havelund, K., Biere, A.: High-level data races. Softw. Test. Verif. Reliab. **13**(4), 207–227 (2003). https://doi.org/10.1002/stvr.281
4. Baumeister, J., Finkbeiner, B., Schwenger, M., Torfah, H.: FPGA stream-monitoring of real-time properties. ACM Trans. Embed. Comput. Syst. **18**(5s) (2019). https://doi.org/10.1145/3358220
5. Binkert, N., et al.: The Gem5 simulator. SIGARCH Comput. Archit. News **39**(2), 1–7 (2011). https://doi.org/10.1145/2024716.2024718
6. Bodden, E., Havelund, K.: Aspect-oriented race detection in Java. IEEE Trans. Software Eng. **36**(4), 509–527 (2010). https://doi.org/10.1109/TSE.2010.25

7. Convent, L., Hungerecker, S., Scheffel, T., Schmitz, M., Thoma, D., Weiss, A.: Hardware-based runtime verification with embedded tracing units and stream processing. In: Colombo, C., Leucker, M. (eds.) RV 2018. LNCS, vol. 11237, pp. 43–63. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03769-7_5
8. Decker, N., et al.: Online analysis of debug trace data for embedded systems. In: Madsen, J., Coskun, A.K. (eds.) Design, Automation & Test in Europe Conference & Exhibition, DATE 2018, pp. 851–856. IEEE (2018)
9. Drzevitzky, S., Kastens, U., Platzner, M.: Proof-carrying hardware: towards runtime verification of reconfigurable modules. In: 2009 International Conference on Reconfigurable Computing and FPGAs, pp. 189–194. IEEE (2009)
10. Filman, R., Havelund, K.: Source-code instrumentation and quantification of events. In: Foundations of Aspect-Oriented Languages (FOAL 2002), No. TR 02-06, April 2002. http://www.cs.ucf.edu/~leavens/FOAL/papers-2002/TR.pdf
11. Flanagan, C., Freund, S.N.: FastTrack: efficient and precise dynamic race detection. In: Hind, M., Diwan, A. (eds.) Proceedings 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, pp. 121–133. ACM (2009)
12. Havelund, K., Reger, G., Roşu, G.: Runtime verification past experiences and future projections. In: Steffen, B., Woeginger, G. (eds.) Computing and Software Science. LNCS, vol. 10000, pp. 532–562. Springer, Cham (2019). https://doi.org/10.1007/978-3-319-91908-9_25
13. Havelund, K., Rosu, G.: Monitoring Java programs with Java PathExplorer. Electron. Notes Theor. Comput. Sci. **55**(2), 200–217 (2001). https://doi.org/10.1016/S1571-0661(04)00253-1
14. Havelund, K., Rosu, G.: An overview of the runtime verification tool Java PathExplorer. Formal Methods Syst. Des. **24**(2), 189–215 (2004). https://doi.org/10.1023/B:FORM.0000017721.39909.4b
15. Havelund, K., Roşu, G.: Runtime Der. In: Colombo, C., Leucker, M. (eds.) RV 2018. LNCS, vol. 11237, pp. 3–17. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03769-7_1
16. Hong, S., Kim, M.: A survey of race bug detection techniques for multithreaded programmes. Softw. Test. Verif. Reliab. **25**(3), 191–217 (2015)
17. Jahic, J., Jung, M., Kuhn, T., Kestel, C., Wehn, N.: A framework for non-intrusive trace-driven simulation of manycore architectures with dynamic tracing configuration. In: Colombo, C., Leucker, M. (eds.) RV 2018. LNCS, vol. 11237, pp. 458–468. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03769-7_28
18. Jaksic, S., Bartocci, E., Grosu, R., Kloibhofer, R., Nguyen, T., Nickovic, D.: From signal temporal logic to FPGA monitors. In: 13. ACM/IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE 2015, pp. 218–227. IEEE (2015)
19. Jakšic, S., Li, D., Pun, K.I., Stolz, V.: Stream-based dynamic data race detection. In: 31st Norsk Informatikkonferanse, NIK 2018. Bibsys Open Journal Systems, Norway (2018). https://ojs.bibsys.no/index.php/NIK/article/view/511
20. Joe, N.: Concurrent programming, with examples, March 2020. https://begriffs.com/posts/2020-03-23-concurrent-programming.html
21. Leucker, M., Sánchez, C., Scheffel, T., Schmitz, M., Schramm, A.: TeSSLa: runtime verification of non-synchronized real-time streams. In: ACM Symposium on Applied Computing (SAC), pp. 1925–1933. ACM (2018)

22. Lucia, B., Ceze, L., Strauss, K., Qadeer, S., Boehm, H.: Conflict exceptions: simplifying concurrent language semantics with precise hardware exceptions for data-races. In: Seznec, A., Weiser, U.C., Ronen, R. (eds.) 37th International Symposium on Computer Architecture (ISCA 2010), pp. 210–221. ACM (2010)
23. Matar, H.S., Tasiran, S., Unat, D.: EmbedSanitizer: runtime race detection tool for 32-bit embedded ARM. In: Lahiri, S., Reger, G. (eds.) RV 2017. LNCS, vol. 10548, pp. 380–389. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67531-2_24
24. MIRA Ltd.: MISRA C:2012 Guidelines for the use of the C language in critical systems (2013)
25. Moosbrugger, P., Rozier, K.Y., Schumann, J.: R2U2: monitoring and diagnosis of security threats for unmanned aerial systems. Formal Methods Syst. Design **51**(1), 31–61 (2017). https://doi.org/10.1007/s10703-017-0275-x
26. Preußer, T., Weiss, A.: The CEDARtools platform - massive external memory with high bandwidth and low latency under fine-granular random access patterns. In: Sourdis, I., Bouganis, C., Álvarez, C., Díaz, L.A.T., Valero-Lara, P., Martorell, X. (eds.) 29th International Conference on Field Programmable Logic and Applications, FPL 2019, pp. 426–427. IEEE (2019)
27. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.E.: Eraser: a dynamic data race detector for multithreaded programs. ACM Trans. Comput. Syst. **15**(4), 391–411 (1997)
28. Serebryany, K., Potapenko, A., Iskhodzhanov, T., Vyukov, D.: Dynamic race detection with LLVM compiler. In: Khurshid, S., Sen, K. (eds.) RV 2011. LNCS, vol. 7186, pp. 110–114. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29860-8_9
29. Watterson, C., Heffernan, D.: Runtime verification and monitoring of embedded systems. IET Softw. **1**(5), 172–179 (2007)
30. Yu, Z., Yang, Z., Su, X., Ma, P.: Evaluation and comparison of ten data race detection techniques. Int. J. High Perform. Comput. Network. **10**(4–5), 279–288 (2017)

PAPER **C**

MC/DC TEST CASES GENERATION BASED ON BDDS

Faustin Ahishakiye, José Ignacio Requeno Jarabo, Lars Michael Kristensen, Volker Stolz

In Proceedings of the Symposium on Dependable Software Engineering Theories, Tools and Applications, **SETTA 2021**, LNCS, volume 13071, Springer, Cham Beijing, China, November 2021

MC/DC Test Cases Generation based on BDDs ⋆

Faustin Ahishakiye[1], José Ignacio Requeno Jarabo[1,2], Lars Michael Kristensen[1], and Volker Stolz[1]

[1] Western Norway University of Applied Sciences, Bergen, Norway
[2] Complutense University of Madrid, Madrid, Spain
{fahi,jirj,lmkr,vsto}@hvl.no, jrequeno@ucm.es

Abstract. We present a greedy approach to test-cases selection for single decisions to achieve MC/DC-coverage of their Boolean conditions. Our heuristics take into account "don't care" inputs through three-valued truth values that we obtain through a compact representation via reduced-ordered binary decision diagrams (roBDDs). In contrast to an exhaustive, resource-consuming search for an optimal solution, our approach quickly gives frequently either optimal results, or otherwise produces "good enough" results (close to the optimal size) with little complexity. Users obtain different — possibly better — solutions by permuting the order of conditions when constructing the BDD, allowing them to identify the best solutions within a given time budget. We compare variations on metrics that guide the heuristics.

1 Introduction

Software testing techniques that achieve coverage effectiveness and provide test cases are cost intensive [31]. Certification standards for safety assurance such as DO-178C [28] in the domain of avionic software systems require software with the highest safety level (Level A) to show modified condition decision coverage (MC/DC) [10]. One of the advantages of MC/DC is that for a decision with n conditions, it may be satisfied with less test cases: between a lower-bound of $n+1$ and upper-bound of $2n$ test cases, compared to multiple condition coverage (MCC) which requires 2^n test cases. MC/DC requires that each condition in a decision shows an independent effect on that decision's outcome by (1) varying just that condition while holding fixed all other possible conditions (**UC-MC/DC**), or (2) varying just that condition while holding fixed all other possible conditions that could affect the outcome. This criterion of showing independence effect for conditions is unique for MC/DC compared to other structure coverage criteria.

While trying all possible combinations is exhaustive and requires tremendous resources [18], as well as becoming impracticable for a high number of conditions

⋆ This work was supported by the Spanish Ministry of Science and Innovation under project FAME (grant nr. RTI2018-093608-B-C31), the Comunidad de Madrid under project FORTE-CM (grant nr. S2018/TCS-4314) co-funded by EIE Funds of the European Union, the SFI Smart Ocean NFR Project 309612/F40, and the NFR Project COEMS Training Network 309527.

[23,19], finding a test set equal or closer to $n+1$ with MC/DC assurance is also a non-trivial task [24,15]. Therefore, it is important to investigate new strategies for generating good test suites both in terms of number of test cases [10] and coverage adequacy [34,14] with little complexity and with reasonable resources.

In this paper, we present a novel and alternative approach to test case generation satisfying MC/DC based on reduced-ordered binary decision diagrams (roBDDs) which are a concise representation of Boolean expressions. roBDDs are widely used in different areas such as computer aided design (CAD) tasks [26], symbolic model checking [11,26], and verification of combinational logic [20,29]. Due to their reduced form compared to other Boolean expressions representations such as disjunctive or conjunctive normal form, truth tables and formula equivalence [35]; roBDDs offer a unique normal form and were also already used in test cases generation [17,22] for different coverage criteria other than MC/DC.

We present an algorithm that takes as input the roBDD representing a Boolean expression and constructs a set of MC/DC pairs. For a decision of n conditions, we generate n pairs that contain between $n+1$ to $2n$ test cases altogether. We select paths based on their length in roBDDs and reuse factor ($\alpha()$). The reuse factor refers to the number of pairs that use a given path.

We propose and compare heuristics with different preferences with respect to three-valued truth-values (1, 0 and ?) and the length of paths in the roBDD. All of them maximize the reuse factor ($\alpha()$) together with a second criteria, namely: the longest paths in BDD ($\mathcal{H}_{LPN}$, $\mathcal{H}_{LPB}$), the longest paths which may merge ($\mathcal{H}_{LMMN}$,$\mathcal{H}_{LMMB}$), and the longest paths with better size ($\mathcal{H}_{LPBS}$). Each type of heuristic implements two different flavors which sort the BDD paths depending on the interpretation of the reuse factor as a natural number ($\mathcal{H}_{LPN}$, $\mathcal{H}_{LMMN}$) or as a boolean value ($\mathcal{H}_{LPB}$, $\mathcal{H}_{LMMB}$)(e.g., $\alpha(p,\psi) < \alpha(q,\psi)$). Our algorithm is implemented in Python and the PyEDA library [13]. We test our algorithm on the Traffic Alert and Collision Avoidance System (TCAS II) benchmarks [33] which are widely used in the literature [19,21,37,22,17].

BDDs are sensitive to conditions ordering, such that different orders yield different BDDs and their size in the worst case grows to 2^{2^n} nodes [27]. As the number of nodes increases there are many paths to select MC/DC pairs from. We present evidence that to find an optimal or "good enough" solutions, instead of a search with backtracking, it is sufficient to try a few different permutations.

The rest of this paper is organized as follows: in Section 2 we present our terminology, notations and a background on MC/DC and BDDs. Section 3 describes our approaches and algorithm for generating test cases satisfying MC/DC based on BDDs. Section 4 explains the implementation of our algorithm and discuss the results. In Section 5 we provide the state of the art of the existing related work. Finally, we present the concluding remarks and future work in Section 6.

2 Background

In this section, we provide the background on MC/DC and BDDs. We present several basic definitions and terminology which are used throughout this paper.

Conditionals in source code, as well as logical expressions in software specifications can be formalized as Boolean expressions. Both BDDs and MC/DC deal with Boolean expressions.

Definition 1 (Boolean expression). *A Boolean expression is defined as an expression that can be evaluated to either* true *(T) or* false *(F) and can contain connectives: NOT, AND, OR, XOR (exclusive-or), denoted by $\neg$, $\wedge$, $\vee$, and $\oplus$ respectively.*

There has been some confusion on what is a condition and decision in the context of source code and the Certification Authorities Software Team (CAST) provided suitable definitions [7]: each occurrence of a condition is considered as a distinct condition, whereas we treat multiple occurrences of a variable as one condition, where c and $\neg c$ are strongly coupled conditions.

Definition 2 (Condition). *A* condition *denotes a logical indivisible (atomic) expression containing no Boolean operators except for the unary operator ($\neg$). It contains a Boolean variable represented by a, b, c,..., defined over "0" or "1".*

Definition 3 (Decision). *A* decision *is a Boolean expression composed of conditions and zero or more Boolean operators. It is denoted by $D = c_1 \Box c_2 \Box c_3 \cdots \Box c_i \Box \cdots \Box c_n$, where c_i, $(1 \leq i \leq n)$ are Boolean conditions and $\Box$ stands for a binary Boolean operator. A decision is also known as a Boolean function.*

Definition 4 (Two/Three-valued test case). *Given a decision D, a* test case *is a truth vector $tc = (I_1, I_2, I_3, \cdots, I_n)$ where $I_i \in \{0, 1\}$ (respectively, $\{0, 1, ?\}$) are the inputs assigned to each conditions. ? is known as "don't care" meaning that a condition does not need to be evaluated due to short-circuiting. A* set of test cases *for a given decision is called a* test suite*. We denote the projection onto the truth-value at the position corresponding to some condition c in the test case tc as $tc[c]$.*

2.1 Modified condition decision coverage (MC/DC) criterion

We first give the well-known definitions for two-valued truth values, and will later extend the definitions into the three-valued setting. MC/DC subsumes the existing logical coverage criteria such as condition coverage (CC): each condition tested once true and false, decision coverage (DC): a decision is evaluated once true and once false, and multiple condition coverage (MCC): an exhaustive testing that requires all possible combination of inputs. For MC/DC each condition has to independently affect the decision's outcome. According to DO-178C [28] and CAST-10 [7] the following definition has been provided for MC/DC:

Definition 5 (MC/DC [30]).

A decision is said to be MC/DC covered iff: (i) Every point of entry and exit in the program has been invoked at least once, (ii) every condition in a decision in the program has taken all possible outcomes at least once, (iii) every decision in the program has taken all possible outcomes at least once, (iv) *each condition in a decision has shown to independently affect that decision's outcome by: (1) varying just that condition while holding fixed all other possible conditions(**UC-MC/DC**), or (2) varying just that condition while holding fixed all other possible conditions that could affect the outcome (**Masking MC/DC**).*

The coverage of program entry and exit in the Definition 5 is not directly connected with the main point of MC/DC [32], as we only consider expressions, not programs. The most interesting part of the MC/DC definition is showing the independent effect, which demonstrates that each condition of the decision has a defined purpose. The item (1) in the definition defines the unique cause MC/DC which original MC/DC [9]. The item (2) has been introduced in DO-178C to clarify that so-called *Masked MC/DC* is allowed [6,28]. Masked MC/DC means that it is sufficient to show the independence effect of a condition by holding fixed only those conditions that could actually influence the outcome. In our analysis, we are interested in generating MC/DC test cases that show an independence effect of each condition in the decision with acceptable size.

Definition 6 (Independence effect of a condition, independence pair, $\oplus_c$). *Given two test cases tc, tc' for a decision D, we call tc* independent from *tc'* on condition *c, iff i) $D(tc) = \neg D(tc')$ (they evaluate to opposite truth values), and ii) $tc \oplus_c tc'$, where $\oplus_c$ means they differ exactly only in the input position corresponding to condition c. We then say that "tc and tc' form an independence pair" (for some condition c), written $uc(tc, tc')$.*

We will later see that in our three-valued interpretation, a test case cannot form an independence pair if it does not contain enough concrete input to evaluate to either true or false. We now reformulate the general definition of MC/DC from Def. 5 for our purposes:

Definition 7 (MC/DC-cover). *Given a decision D and set of test cases ψ, we say that ψ* MC/DC-covers *D, iff $\forall c \in D, \exists tc, tc' \in \psi : tc \oplus_c tc' \wedge uc(tc, tc')$ (tc is independent from tc' for every condition c).*

In other words, a set is an MC/DC-cover for a decision D, if for every condition, there exists a pair of test cases in that set which shows the independence effect of that condition by evaluating to opposing truth values.

Example 1. Consider a decision $D = (a \wedge b) \vee c$. The truth table representing MCC and all possible MC/DC pairs is given in Table 1(a). Each pair is showing the independence effect for a condition. The advantage of MC/DC over MCC can be seen from Table 1(a). MCC requires eight test cases whereas all possible MC/DC pairs contain seven test cases. Indeed, only the four test cases shown in Table 1(b) are required to achieve MC/DC [10,9]. However, choosing a set equal or closer to minimal number of test cases is non-trivial for testers, especially when there is more than one MC/DC pair for a certain condition, for example, condition c can be covered by either of three pairs (indicated in parentheses), as shown in Table 1(a).

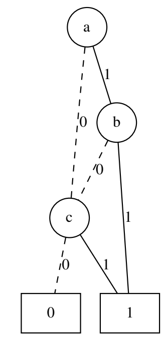


Fig. 1: roBDD: $D = (a \wedge b) \vee c$

tc	a	b	c	D	MC/DC pairs
1	0	0	0	0	
2	0	0	1	1	c(1,2)
3	0	1	0	0	
4	0	1	1	1	c(3,4)
5	1	0	0	0	
6	1	0	1	1	c(5,6)
7	1	1	0	1	a(3,7),b(5,7)
8	1	1	1	1	

(a) MCC & All MC/DC pairs

π	a	b	c	D	MC/DC pairs
1	0	?	0	0	
2	1	1	?	1	a(1,2)
3	1	0	0	0	b(2,3)
4	1	0	1	1	c(3,4)

(b) MC/DC set of paths

tc	a	b	c	D	MC/DC pairs
1	0	1	0	0	
2	1	1	0	1	a(1,2)
3	1	0	0	0	b(2,3)
4	1	0	1	1	c(3,4)

(c) MC/DC set of test cases

Table 1: MCC & MC/DC pairs for $D = (a \wedge b) \vee c$.

Chilenski et al. [10,9] investigated that for a decision D with n conditions, UC-MC/DC can be achieved with a minimal number of $n+1$ tests while Masking MC/DC be achieved with a minimal number of $\lceil 2 * (\sqrt{n}) \rceil$ tests. This is achieved by choosing MC/DC pairs that overlap where every condition past the first one (which requires two test cases), only adds a single test case to the existing set.

Lemma 1 (Minimal MC/DC-Covers [9,1]). *If a coverage set exists for a decision D with n conditions, then there also exists a smaller set (possibly with different test cases) thereof with exactly $n+1$ test cases such that it MC/DC-covers D for UC MC/DC.*

2.2 Overview on binary decision diagrams (BDDs)

BDDs are canonical representations of Boolean functions compared to other Boolean expressions representations such as disjunctive normal form (DNF), conjunctive normal form (CNF), truth tables and formula equivalence [35]. To reduce BDDs, conditions in a decision need to be ordered and duplicated terminals and isomorphic sub-trees have to be merged. The resulting graph is known as reduced ordered BDD (roBDD) and is shown in Figure 1 for the Example 1.

BDDs represent formulas *compact* in the sense that it takes little memory to store the representation, the number of nodes in a roBDD is reduced and there is exactly one optimal and unique graph for each Boolean expression [35].

Definition 8 (Path through an roBDD, $\pi, \pi[x]$). *Given an roBDD for some decision D over Boolean variables $x_0, \ldots, x_1$. We denote a path from the root of the BDD to a terminal with π, and write $\pi[x] = 1$ if the path takes the true-branch in the node labelled with condition x (0/false respectively), and $\pi[x] =?$ if the path does not pass through a node labelled with condition x. That is, although paths through the roBDD can be of different lengths, for uniformity we always represent them as a vector with n elements.*

We also extend the evaluation of a decision wrt. some inputs ($D(0 \ldots 0)$) to BDDs and use $D(\pi)$ to denote the three-valued truth-value that the path represents. The obvious correspondence between a test case and a path through the roBDD is that a test case may provide more truth-values as inputs than are strictly necessary on this path. For example, an MC/DC pair of paths for condition a is $\{(0?0), (11?)\}$ as shown in row 1 & 2 of Table 1(b). The fully instantiated test cases for this pair are $\{(010), (110)\}$ (row 1 & 2, Table 1(c)).

3 Approaches and algorithm for test cases generation

Our approach and heuristics for test case generation are based on roBDDs that guide our search for test case selection. We start with a set of roBDDs paths from the root and construct sets satisfying MC/DC for all conditions, where each set contains n MC/DC pairs.

BDDs are sensitive to variable ordering: to deal with the ordering effect, we collect solutions for a number of permutations on the variable ordering. As the number of conditions in a decision increases, the number of permutations ($n!$ for n conditions) increases over-exponentially. Since generating the set of solutions for all permutation would be infeasible in those cases, we show that for few permutations we generate some test suites of minimal size, based on the selection methods defined in Subsection 3.2. In the following, we assume that all BDDs that occur are roBDDs.

3.1 Theorems and definitions for MC/DC in terms of BDDs

The core of our contribution is as follows: our algorithm produces a set of three-valued test cases, which we can instantiate to fulfill the original definition of MC/DC. We first extend general results from the standard two-valued Boolean logic to a three-valued logic.

Definition 9 (Three-valued independence pair, $\oplus_c^3$). *Given two* three-valued *test cases tc, tc' for a decision D, we write $uc3(tc, tc')$ iff*
i) $D(tc) = \neg D(tc')$ (they evaluate to opposite concrete *truth values), and ii) $tc \oplus_c^3 tc'$, where $\oplus_c^3$ means at least one of the inputs for some condition c is a concrete truth value, and for every other condition the three-valued inputs coincide or one of them is "?".*

Example 2. Let $D(X, Y, Z) = X \wedge ((\neg Y \wedge \neg Z) \vee (Y \vee Z))$. Consider $tc = (0??)$ with $D(tc) = 0$ and $tc' = (11?)$ with $D(tc') = 1$ respectively, hence $uc3(tc, tc')$. Observe that hence also e.g. $uc3(011, 11?)$ and $uc(011, 111)$.

We next show that each three-valued independence pair can be instantiated to some two-valued independence pair by suitable substitution of unknown values. In the following, for readability, we describe functions from our implementation through their properties instead of operationally. The first function combines two compatible test cases into a single one. We need this later in our algorithm to refine existing test cases such that we keep only one test case when two cases overlap.

Definition 10 $(merge(tc, tc'))$**.** *Given test cases tc, tc', we obtain $\sigma = merge(tc, tc')$, where $\forall c \in C, (\sigma[c] = tc[c] \wedge tc'[c] = ?) \vee (\sigma[c] = tc'[c] \wedge tc[c] = ?)$.*

In other words, *merge* substitutes some ? in a pair of paths, such that all conditions have equal values. The result is undefined if they disagree in one position where one has true and the other false. This can be understood as unifying both test cases with each other, taking ? as free variables.

Note that we ignore the actual outcome when merging wrt. a decision, but only ever consider the inputs. As we will also consider test cases that differ in *exactly one* position, we define the following variation:

Definition 11 $(merge_x(tc, tc'))$**.** *Given test cases tc, tc', we obtain $\sigma = merge_x(tc, tc')$, where $\forall c \in C \setminus \{x\}, (\sigma[c] = tc[c] \wedge tc'[c] = ?) \vee (\sigma[c] = tc'[c] \wedge tc[c] = ?) \wedge \sigma[\mathbf{x}] = \mathbf{tc}[\mathbf{x}]$ (emphasis added).*

Note that this definition is biased to reproduce the truth-value in the designated position x from the first input, and we will consequently later see it applied *twice*, once from left to right argument, and also from right to left argument.

Example 3. We have $merge_{c_2}((1?0), (11?)) = (110)$, but $merge_{c_2}((11?), (1?0)) = (11?)$, with c_2 the condition that is placed in the last position.

Definition 12 (Specialization $\leqq$). *Given three-valued test cases p, q, we say that $p \leqq q$ iff $\exists p' : p = merge(p', q)$ ("p specializes q").*

Due to the same format for a test case and for a roBDD path (see Def. 8), both concepts are interchangeable and $\leqq$ can specialize any of them. The relation $\leqq$ is a partial order (straightforward).

Theorem 1 (Usefulness of three-valued MC/DC). *Given a decision D and set φ of three-valued test-cases that is a three-valued MC/DC cover for D, i.e., $\forall c \in D : \exists tc, tc' \in \varphi, tc \oplus_c^3 tc' \wedge uc3(tc, tc')$. Then there exists a two-valued set of test cases $\psi \subseteq 2^{\mathcal{B}^{|D|}}$, such that:*

(1) $\forall tc, tc' \in \varphi : uc3(tc, tc') \Rightarrow \exists u, u' \in \psi : u \oplus_c u' \wedge u \leqq tc \wedge u' \leqq tc'$
(each test case pair in φ has been specialised)
(2) $\forall u, u' \in \psi : D(u) = \neg D(u') \wedge u \oplus_c u' \Rightarrow \exists tc, tc' \in \varphi : uc3(tc, tc')$
$\wedge u \leqq tc \wedge u' \leqq tc'$($\psi$ is the smallest set that specialises φ).

It follows that ψ is an MC/DC-cover for D.

Proof. (1) Because of $uc3(tc, tc')$, tc or tc' have a concrete value in c and coincide for the rest of conditions c_i, except for those positions c_i where one of the test cases is ?. Hence, $u = merge_{c_i}(tc, tc')$ returns a new test case where $u \leqq tc$ as the ? are instantiated (symmetrically, $u' = merge_{c_i}(tc', tc)$), excluding condition

c. MC/DC imposes that $u[c] = \neg u'[c]$, so the selection of tc and tc' satisfies that either a) $tc[c] = \neg tc'[c]$, or b) $tc[c] =?$ or $tc'[c] =?$. In b), $u[c] = tc[c]$ and $u'[c] = tc'[c]$: if any of these values is a ?, then they are properly instantiated so that $u \oplus_c u'$.

(2) As $u \oplus_c u'$, u and u' are equal except for condition c. Then, tc and tc' are constructed by replacing a finite number of positions in u (similarly, u') with ? such that they keep $uc3(tc, tc')$. Because tc and tc' are abstractions of u and u', $u \leqq tc \wedge u' \leqq tc'$.

Due to the specialization relation, multiple sets of two-valued test cases can be constructed that satisfy the above property: φ may contain a test case tc with "don't care" for some condition c, and also "don't care" for every other partner tc' in the pairs it is participating in. Then, this input c can be instantiated to either truth value. Our algorithm 1, which uses the roBDD to populate φ, guarantees that there will exist at least a pair tc, tc' such that $tc[c] = \neg tc'[c]$ for every condition c, if the decision can be MC/DC-covered.

Next, we define the function that identifies suitable test cases that we might want to add our set ψ. Based on the following criteria, for every uncovered condition the algorithm adds a new test case together with a complementary one such that the pair shows the independence effect of the condition.

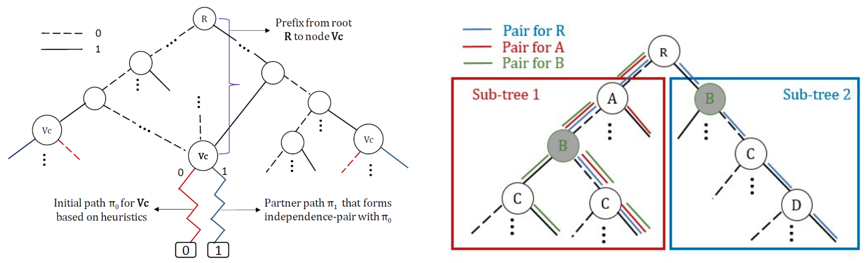
Definition 13 (Reuse factor $\alpha(\pi, \psi), \alpha_{=_3}(\pi, \psi)$). *Given the set of MC/DC pairs of paths $(\pi^\perp, \pi^\top) \in \psi$ with $D(\pi^\perp) = 0$ and $D(\pi^\top) = 1$, the reuse factor $\alpha(\pi, \psi)$ represents the number of pairs in ψ that use π. It is calculated as $\alpha(\pi, \psi) := |\{(\pi, (\pi^\perp, \pi^\top)) \mid \pi = \pi^\perp \vee \pi = \pi^\top, (\pi^\perp, \pi^\top) \in \psi\}|$.*

Relation to BDDs. A pair (tc, tc') of test cases showing the independence of some condition c_i has a vivid graphical interpretation on the BDD. It corresponds to a pair of paths $(\pi^\perp, \pi^\top)$ such that:

1. the tests evaluate the opposite truth values (i.e., $D(tc) = \neg D(tc')$);
2. $tc \leqq \pi^\perp, tc' \leqq \pi^\top$ (order wlog., the test cases may contain more input than strictly necessary).
3. both reach some node v_{c_i} using the same path through BDD(D) (i.e., $\pi^\perp[j] = \pi^\top[j]$ for $0 \leq j < i$);
4. their paths from v_{c_i} exit on either edge (i.e, $\pi^\perp[i] = \neg\pi^\top[i]$);
5. after v_{c_i}, both test cases take compatible choices along the paths for the remaining conditions, so that the independence property holds (i.e., $\pi^\perp[j] =_3 \pi^\top[j]$ for $i < j < n$).

This means especially that the two paths cannot cross (after the condition-node v_{c_i}), since this would immediately indicate an incompatible choice.

Figure 2 represents the overview on the selection of MC/DC pairs from the roBDD. The roBDD contains the root node labeled by R, non-terminal nodes labeled with conditions and two terminal nodes (0 and 1). The nodes are connected by solid and dashed edges representing assignments of 1 and 0 to each condition respectively. Every condition c may be represented multiple times on


0
1
R
Prefix from root R to node Vc
Vc
Vc
Vc
0
1
Initial path π0 for Vc based on heuristics
Partner path π1 that forms independence-pair with π0
0
1
Pair for R
Pair for A
Pair for B
R
Sub-tree 1
A
B
C
C
Sub-tree 2
B
C
D

(a) Pair selection (b) Reuse effect

Fig. 2: Overview on MC/DC pairs selection path from BDD and reuse effect

the BDD (nodes v_c). There may exist multiple paths to such a node. For every path reaching a (non-terminal) node, we attempt to extend it to construct pairs that show the independence effect of that condition. It is not guaranteed that the two complementary paths lead to opposite terminal nodes and our algorithm must explicitly check it step-by-step (modulo “don’t care”-steps). The figure shows a representative of such pairs, $(\pi^{\perp}, \pi^{\top})$: they share the same prefix for all ordered conditions up to v_c. They then proceed in lock-step through the two branches to the terminals.

Figure 2 (b) illustrates some of the effects that we aim to achieve: as we search for pairs in the order of the roBDD, we will obtain some pair (shown in blue) from the heuristics (e.g. based on “longest path”) which differs directly in the condition R for the root node. The next condition A in the order exists only in the left subtree, and we prefer a pair for it that reuses one of the previous path. Here, this can only be the left path for R, and hence we check if for the path that condition A shares with condition R we can construct a compatible path to the opposite terminal after leaving the node for A through the opposite edge (red pair). For condition B, we attempt to construct a pair by reusing the right branch for condition R (blue), and another one that uses the path that we used before both for R and A. We either take the only pair that fulfils our criteria, or again have the heuristics break a potential tie, here resulting in the green pair for condition B.

Due to the structure of roBDD, the derived test cases correspond to *MC/DC + short circuit* [6,5] where a test case can be composed with a three-valued assignment (*0: false, 1:true, and ?:not evaluated(a condition does not appear along the path)*). Therefore, to find the test cases that satisfy Unique Cause MC/DC [9], the “don’t care” assignments will be replaced by either 0 or 1 pairwise (by the corresponding value at the same position in the partner path).

3.2 Algorithm and heuristics for test cases generation

Our approach for MC/DC test case generation for a decision D is based on the three-valued paths that are extracted from the equivalent roBDD. The MC/DC coverage criteria requires a pair of test cases that shows the independence effect for every condition. The presence of "don't care" values in a BDD path gives us some flexibility when instantiating it to a test case and finding the complementary test case that leads to the opposite Boolean evaluation. As the wildcards may specialize to any Boolean value, we propose a greedy algorithm that tries to minimize the overall number of test case pairs for a decision D with n conditions from $2n$ to a value as close as possible to $n+1$.

To this end, our method is divided in two stages: during the first phase, it initializes the MC/DC test suite with paths that are extracted from the BDD through any of our predefined heuristics, which intend to maximize the reuse factor in order to reduce the differences among test cases. Secondly, the selected BDD paths are specialized so that the wildcards take a concrete value while preserving the independence effect. We lift this property to sets of pairs of test cases with the definition of *instantiate* which computes the smallest set such that it guarantees that all members have been merged if possible:

$$\begin{aligned}
&\forall(s,s') \in \mathit{instantiate}(S):\\
&\quad \exists(p,p') \in S : uc3(p,p') \land s \leqq p \land s' \leqq p' && \text{(instantiated from } S\text{)}\\
&\quad \land\ \forall(p,p')\ \forall(q,q') \in S : s \leqq p \land p \leqq q \land p \leqq q' \Rightarrow s = p \land s' = p' && \text{(least upper bound)}\\
&\quad \land\ \exists c : \mathit{merge}_c(s,s') = s \land \mathit{merge}_c(s',s) = s' && \text{(fully merged).}
\end{aligned}$$

This approach takes $n = |C|$ iterations, and each iteration adds a pair consisting of at most two new paths to the set. If S is empty, we can abort as this means there does not exist any pair showing the independence effect of that condition, and hence the decision D cannot be covered with the MC/DC-property. Correspondingly, unless we abort, the final set will contain n pairs, consisting of at most $2n$ individual paths. By construction, these pairs will provide three-valued MC/DC-coverage of the decision.

This leaves us two points to address: i) can we avoid constructing the set of *all* pairs for a condition, but instead only use a relevant, smaller subset as input to the heuristics, and ii) can we present evidence that our heuristics have a high likelihood of picking pairs that not only reuse a path from the already

Algorithm 1: MC/DC Test case generation

```
   Input: An roBDD over conditions $C$ with root $r$ for a formula $\varphi$
   Output: Set $\psi$ of pairs of test cases that MC/DC-cover $\varphi$ with
           $|C|+1 \leq |\bigcup\{\{tc, tc'\} | (tc, tc') \in \psi\}| \leq 2|C|$.
1  $\psi = \emptyset$;
2  forall $c \in C$ do
3      Let $S := \{(\pi^{\top}_{v_c}, \pi^{\perp}_{v_c}) \mid$ where $\pi^{\top}_{v_c}, \pi^{\perp}_{v_c}$ are paths from the root $r$ via some $v_c$
         to $\top$ and $\perp$ respectively, such that $[\pi^{\top}_{v_c}] \oplus_c [\pi^{\perp}_{v_c}]\}$.
4      Abort if $S = \emptyset$: no MC/DC cover of $\varphi$ possible.
5      Let $(p, q) := \mathcal{H}(\psi, S)$ be the result of applying a given heuristics $\mathcal{H}$, such
         that $\exists(p', q') \in S : p = merge_c(p', q'), q = merge_c(q', p')$.
6      $\psi = instantiate(\psi \cup \{p, q\})$
7  end
```

selected pairs (if possible), but also contributes a fresh path that will be reused in the future. We address the first point through algorithmic construction, and evaluate the second through a series of experiments using the TCAS case study.

Algorithmic description. Any approach to a potentially optimal solution must reuse a test case that has already been selected as a partner in a pair for some other condition when selecting a pair for some other condition. It is hence clear that not all pairs for a condition may have to be constructed and evaluated. Rather, we first attempt to directly derive a pair from the existing set of test cases (by flipping only the corresponding condition), and only revert to deriving a new pair of completely fresh paths if such a derived path does not exist. Depending on the heuristics, identifying a completely fresh pair may entail a complete enumeration of pairs: it may be looking for the longest path with most reuse-potential (least number of "don't care"), which could ultimately be the last pair a given traversal of the BDD yields.

The representation as a BDD gives us an advantage in building fresh pairs: by exploring the tree from the root, the ordered labels tell us when we can preempt a search because the condition of interest does not exist in the remaining subtree, and we can continue our search in a sibling. Compared to an exploration of the corresponding truth-table, this effectively allows us to skip over irrelevant rows. We next formalize the notion of path-length in the roBDD.

Definition 14 (Length of a path/test case, $|\sigma|/|tc|$). *Given a path σ in the roBDD for a decision D from the root to a terminal, we denote the length of the path with $|\sigma|$. The length of a* test case $|tc|$ *is that of the underlying path.*

Note that since a test case can have more concrete inputs than are necessary for the path we have in the BDD, the length of a test case may be lower than the number of concrete inputs in that test case.

We propose five selection methods for test cases generation. All of them maximize the reuse factor ($\alpha()$) together with a second criteria, namely: the longest paths in BDD ($\mathcal{H}_{LPN}$, $\mathcal{H}_{LPB}$), the longest paths which may merge ($\mathcal{H}_{LMMN}$, $\mathcal{H}_{LMMB}$), and the longest paths with better size ($\mathcal{H}_{LPBS}$). Each type of heuristic implements two different flavors which sort the BDD paths depending on the interpretation of the reuse factor as a natural number ($\mathcal{H}_{LPN}$, $\mathcal{H}_{LMMN}$) or as a boolean value ($\mathcal{H}_{LPB}$, $\mathcal{H}_{LMMB}$)(e.g., $\alpha(p,\psi) < \alpha(q,\psi)$). We compare them with the *random reuser* ($\mathcal{H}_{RR}$) method as a baseline, which takes the first new path that forms a new pair with an existing test.

$\mathcal{H}_{LPN}/\mathcal{H}_{LMMN}$: This method chooses pairs of paths satisfying MC/DC based on the longest paths in BDDs with the highest reused factor. In case multiple pairs have equal reuse, we choose one where additionally the sum of the lengths is longest. The longest path or higher reuse factor may be better since it can be reused by many conditions that appear along the path.

$$\mathcal{H}_{LPN}(\psi, S) := (merge_c(p,q), merge_c(q,p)) \text{ where } (p,q) \in S$$

such that either (in order):

1. $\alpha(p,\psi) > 0 \land \alpha(q,\psi) > 0 \ \land \forall (p',q') \in S : \alpha(p',\psi) > 0 \land \alpha(q',\psi) > 0$
$\Rightarrow |p| + |q| \geq |p'| + |q'|$
(both test cases were already in the set)
2. $\forall (p',q') \in S : \alpha(p,\psi) + \alpha(q,\psi) \geq \alpha(p',\psi) + \alpha(q',\psi)$ (highest reuse)
$\land\ (\alpha(p,\psi) + \alpha(q,\psi) = \alpha(p',\psi) + \alpha(q',\psi) \Rightarrow |p| + |q| \geq |p'| + |q'|)$
(longest path).

$\mathcal{H}_{LPB}/\mathcal{H}_{LMMB}$: The previous heuristic $\mathcal{H}_{LPN}$ looks at the reuse of the paths in a pair: the existing path may have reuse > 0, and may occur in multiple pairs in the existing set. Its partner path may also be derived from another existing pair. Since it is not clear that past performance ("high reuse = used in multiple pairs by someone before") is an indication for future performance ("does it have more likelihood to be useful in future pairs?"), we also evaluate a variant that only prefers that there is *some* reuse, but not *how much*:

$$\begin{aligned}
&\mathcal{H}_{LPB}(\psi, S) := (\mathit{merge}_c(p,q), \mathit{merge}_c(q,p)) \text{ where } (p,q) \in S : \\
&\quad \alpha(p,\psi) + \alpha(q,\psi) > 0 \qquad \text{(has some reuse)} \\
&\land\ \forall (p',q') \in S : \alpha(p',\psi) + \alpha(q',\psi) > 0 \Rightarrow |p| + |q| \geq |p'| + |q'| \text{(longest path)}.
\end{aligned}$$

The difference between this method and the previous one is that here we consider the reuse factor as Boolean. That is, we choose a pair with the longest paths in BDDs and we check if one of the paths is already reused as a part of an earlier pairs or not. This may give rise to greater non-determinism since more potential partners are considered equivalent.

Longest paths best size ($\mathcal{H}_{LPBS}$): selects MC/DC pairs where the paths have together the highest reuse and the sum of the lengths is strictly the longest.

4 Implementation of MC/DC test cases selection

In this section we describe how we evaluate our approach for the heuristics proposed in Section 3. For each heuristic, one run of Alg. 1 derives a set of test cases for a decision with MC/DC-coverage if it exists. Our heuristics are sensitive to exactly one parameter: the ordering of conditions when constructing the BDD. Furthermore, there is some inherent non-determinism: a heuristic picks randomly among equally best-ranked pairs. It is quite common to observe equivalent pairs with identical reuse and identical path-length. Secondary sources of non-determinism include e.g. iteration over unordered structures like sets which are implementation-specific to a given Python platform.

To give a proper evaluation, we control these in the following way: every heuristic is applied for a number of permutations of the order of the conditions for each decision. For decisions with a low number of conditions, we can hence even exhaustively evaluate the outcome of the heuristics for all permutations. In addition, we repeat a run on a given permutation, exploring different random choices within the equivalent best pairs.

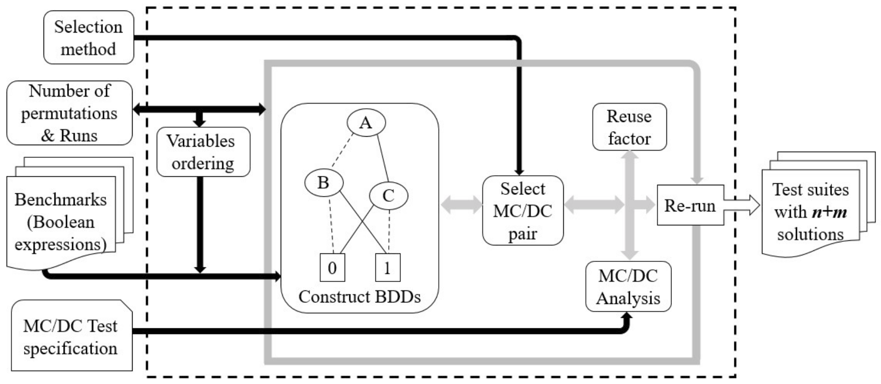


Fig. 3: Test cases generation framework

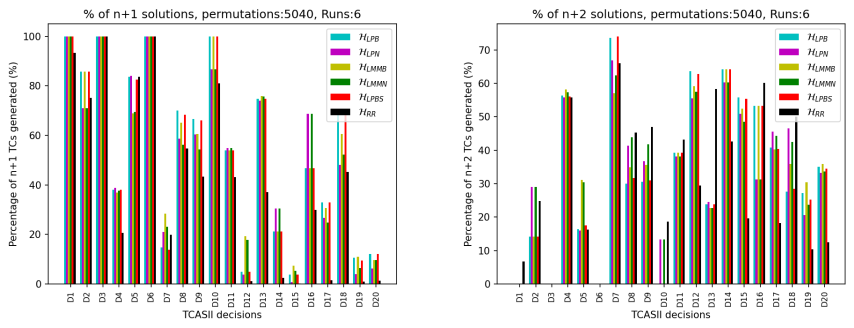
Our framework is based on the PyEDA library [13] and implemented in Python. We test our algorithm on the Traffic Alert and Collision Avoidance System (TCAS II) benchmark [33,25] which has been frequently used in literature [19,21,37,22,17]. The benchmark refers to specifications written as Boolean expressions (decisions) which are logically evaluated to true or false depending on the truth values assigned to the contained conditions.

Below, we present detailed results for a well-known set of TCAS II decisions that can be reproduced with the code in our open source repository [3]. We do not report execution times for our experiment, as our implementation is not optimized in any way beyond obvious algorithmic constructions to minimize BDD-traversal.

4.1 Experimental setup

Fig. 3 shows our test cases generation framework. Our setup takes as input the roBDD for a given decision, the number of permutations, and the number of runs that we perform for each process of test cases generation. The selection method refers to the different heuristics proposed in Section 3: $\mathcal{H}_{LPN}$, $\mathcal{H}_{LPB}$, $\mathcal{H}_{LMMN}$, $\mathcal{H}_{LMMB}$, $\mathcal{H}_{LPBS}$ and $\mathcal{H}_{RR}$. The benchmarks refer to the specifications written as Boolean expressions (decisions) which are logically evaluated to true or false depending on the truth values assigned to the contained conditions. MC/DC test specifications are the meaning of what is MC/DC in the context of roBDDs and three values logic (cfr. Theorem 1 and Def. 9). We consider the reuse factor in our MC/DC analysis to reuse as much as possible the existing selected TCs and finally, we produce n MC/DC pairs as output for each decision with the size of n+m solutions. Our results show that we produce mostly $n + 1$ solutions and the rest of solutions are less than $2n$ with 100% MC/DC.

[3] https://github.com/selabhvl/py-mcdc/



(a) % of n+1 solutions (b) % of n+2 solutions

Fig. 4: Comparison of % for n+1 and n+2 solutions for different heuristics

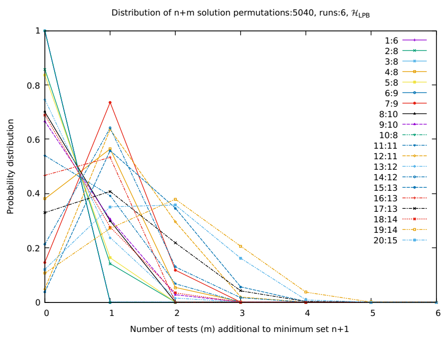
4.2 Experimental Results

Figure 4(a) and (b) present our results as the percentage of generated solutions of sizes $n + 1$ and $n + 2$ for TCAS II based on our heuristics and the baseline RR heuristic. We consider 5040 different orders at most for each decisions (this exhaustively covers all orders for decisions with up to seven conditions). This sample size already yields evidence that repeated application of the algorithm to different orders will discover a (close to) optimal solution reasonably quickly.

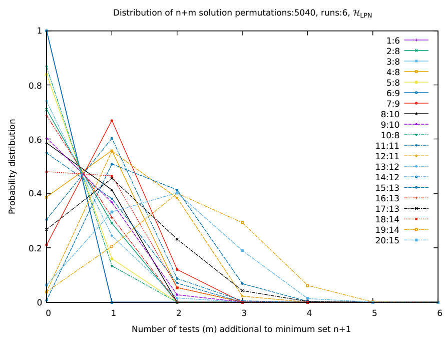
For each heuristic we collect all possible sets of MC/DC covering test cases. MC/DC coverage is calculated as the percentage of the number of covered conditions to the total number of conditions in a decision. In case the MC/DC coverage percentage is less than 100%, it means that MC/DC is not fulfilled for that decision. We present results for solutions of size $n + 1$ (optimal) and $n + 2$ for our heuristics as shown in Figure 4. The charts for the heuristics can be reproduced from our open repository. From the TCAS II benchmark results in Figure 4 and 5, we highlight the following:

1. Our heuristics find the test suite sets of $n + 1$ solutions for each decision, whereas $\mathcal{H}_{RR}$ failed to find any minimal solution for D15. Our heuristics perform better compared to $\mathcal{H}_{RR}$ for 18 out of 20 decisions and have equal results for two decisions in terms of which heuristic has frequently the highest of $n + 1$ solutions with 100% MC/DC. This shows that the approach of permuting order is a viable strategy to eventually obtain an optimal results.
2. $\mathcal{H}_{LPB}$ and $\mathcal{H}_{LMMB}$ out-perform all others with 10 cases (50%) having the highest % of n+1 solutions.
3. Comparing the $\mathcal{H}_{LPB}$ to $\mathcal{H}_{LMMB}$,$\mathcal{H}_{LPB}$ is 2 cases (10%) higher than $\mathcal{H}_{LMMB}$.
4. We observed that $\mathcal{H}_{LMMN}$ is 2 cases (10%) higher than $\mathcal{H}_{LPN}$.
5. $\mathcal{H}_{LPBS}$ has better results in some decisions than $\mathcal{H}_{LMMN}$ and $\mathcal{H}_{LPN}$.
6. In three decisions (D2, D5 and D7), $\mathcal{H}_{RR}$ has better results than some of our our heuristics. We attribute this outcome to random chance.

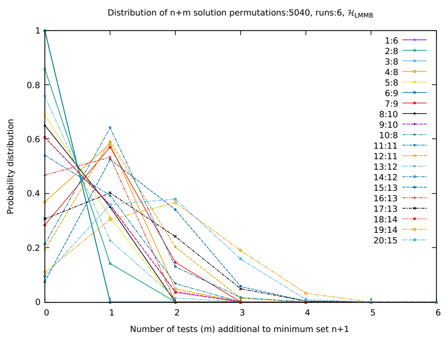
7. From Figure 4 (b) which represent the n+2 solutions, we can see that for the decisions in which we did not find the highest percentage of n+1 solutions now we have a high % of n+2 solutions, which indicates that our test suites generated are closer to lower bound (n+1) of MC/DC minimal set.



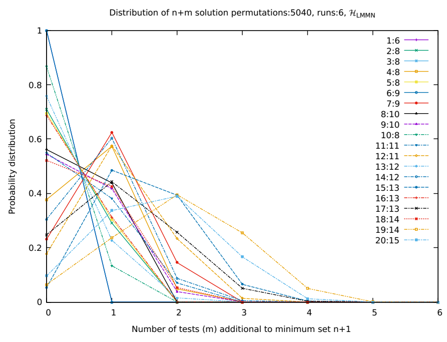
(a) Longest Path Boolean($\mathcal{H}_{LPB}$)



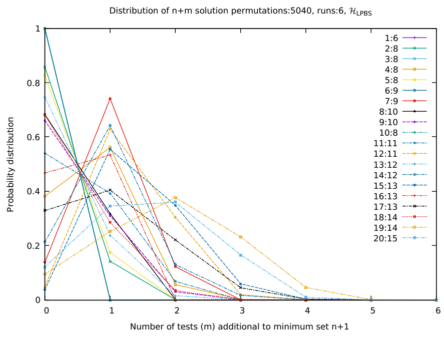
(b) Longest Path Natural($\mathcal{H}_{LPN}$)



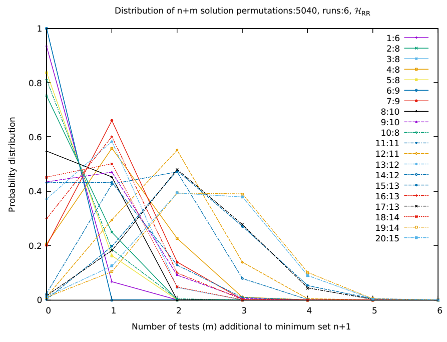
(c) Longest May Merge Boolean($\mathcal{H}_{LMMB}$)



(d) Longest May Merge Natural($\mathcal{H}_{LMMN}$)



(e) Longest Path Better Size($\mathcal{H}_{LPBS}$)



(f) Random Reuser($\mathcal{H}_{RR}$)

Fig. 5: Probability distribution of n+m solutions for 5040 permutations, 6 runs

In summary, our results show that we produce mostly $n+1$ solutions and the rest of solutions are less than $2n$ with 100% MC/DC adequacy. Figures 5 (a)-(f) show the probability distribution of $n+m$ TCs generated for 5040 permutations, 6 runs for different heuristics. The x-axis shows the number of test cases (m) *additional* to the minimal solution ($n+1$). The labels show the decision number and the contained conditions as presented in [25]. All the solutions are have less than $2n$ test cases, as the maximum observed for m is 6 while the range of number of conditions is 6 to 14. Figures 5 shows that most solutions are much closer to the minimal size (to the left) than to the worst case.

Another challenge which is not directly related to our approach but to MC/DC is the coupled and masked conditions where it is difficult to get a full MC/DC coverage with masked condition. For example the decision D10 in the TCAS II benchmark has two conditions (b and h) which are masked. Out of the nine conditions, hence only seven are retained in the roBDD and we compute our minimal solution accordingly.

For the complex example D15 in the 20 TCAS II decisions, our algorithm takes on average 0.7 seconds (incl. time for constructing the BDD) for a single run on an Intel(R) Core(TM) i7-7700 CPU @3.60GHz Linux machine with 64 GB RAM. From the proposed heuristics, we recommend the longest paths with reuse as Boolean number($\mathcal{H}_{LPB}$) as it shows high performance both in terms of high percentage of $n+1$ solutions and short time to compute the solutions compared to the rest of the heuristics.

5 Related work

Automatic test data generation approaches were proposed in [16,4,36] and are based on greedy or meta-heuristic search strategy. They use search algorithms to extract test paths from the control flow graph of a program, then invoke an SMT solver to generate test data [16] and afterwards reduce the test-suite with a greedy algorithm. The drawback for this approach is that often infeasible paths are selected, resulting in significant wasted computational effort. We did not investigate test *data* generation here, only boolean inputs to a single decision.

Kitamura et al. [25] and Yang et al. [37] use a SAT solver to construct minimal MC/DC test suites. That is, the MC/DC criterion is encoded in a single query, and the solver produces a suitable assignment for test case inputs if it exists, or times out. In contrast to the exhaustive nature of SAT queries which may lead to timeouts, our approach delivers a single answer in much less time, but may require repetition to find an optimal solution.

Some of their results do not satisfy UC-MC/DC in some cases, and generate test cases only for Masking MC/DC. There are also some conditions which are reported as infeasible, while the MC/DC pairs for those conditions can be found. For example in [25], decisions 6 and 8 of the TCAS II benchmarks have test suites with 3 and 4 test cases for 8 and 9 conditions respectively which cannot satisfy MC/DC.

A study of enhanced MC/DC coverage criterion for software testing based on n-cube graphs and gray code is presented in [8]. It is an exhaustive approach that takes input as a Boolean expression, builds the n-cube graph, and deduces test cases from all vertices of the graph. Their test cases selection is based on the weight of each test case in a similar way that we calculate the reuse factor of a path. The main difference is that they have to construct the n-cube graph which have the same effect as exhaustive traversal of a truth table and the resulting size of the test suite is not minimal.

Gay et al. [14,15], developed a technique to automatically generate test cases using model checkers for masking MC/DC. Using the JKind model checker, they produce a list of all test inputs and then select the desired test cases while preserving the coverage effectiveness. Their test suite reduction algorithm used to reduced the original test-suite does not guarantee to find the smallest set. They tested their approaches on different real-world avionics systems where they achieved an average MC/DC coverage of 67.67%.

Comar et al. [12] discussed MC/DC coverage in terms of BDD coverage. They examine the set of distinct paths through the BDD that have been taken based on the control flow graph. Based on BDDs they investigated the formalization and comparison of MC/DC to object branch coverage, but the test cases selection is out of their scope. We extend the formalization and definitions of MC/DC in terms of BDDs in the context of test cases selection.

The roBDDs have been used in [22,17] for test cases generation, and highlight the properties and benefits of roBDDs, however, MC/DC was not considered as coverage criterion. Like our approach, their greedy approach incrementally selects a pair of paths where only one condition changes for every condition.

6 Conclusion and Future works

We presented a heuristics-based approach for generating test cases for a Boolean decision (given as roBDD) that satisfy the MC/DC criterion. We evaluate our approach on the TCAS II Benchmark and results shows that we frequently find solutions which are equal or close to the minimal number of test cases without expensive back-tracking.

Our approach is sensitive to variable ordering in the BDD as each order yields a different roBDD. We obtained MC/DC solutions of size $n + 1$ by performing few permutations of conditions in a decision for all tested decisions. We present also the other possible solutions which show full MC/DC coverage. In general, our solutions have a size ranging from $n + 1$ to $2n$, with a high percentage of size $n + 1$ or $n + 2$ solutions, where even the latter, although not optimal, may be acceptable to a user. We proposed different heuristics and compared their properties. All our heuristics perform better than $\mathcal{H}_{RR}$. $\mathcal{H}_{LPB}$ and $\mathcal{H}_{LMMB}$ out-perform all other heuristics with 10 times (50%) having highest percentage of $n + 1$ solutions. We recommend $\mathcal{H}_{LPB}$ since it is 10% better than $\mathcal{H}_{LMMB}$.

For the future work we plan to extend our algorithm so that we support data input coverage where conditions are not abstracted, which requires taking constraints into consideration. We will also attempt to integrate our test case generation algorithm into our MC/DC measurement tool and model[2,3]. Although the experimental data shows that we always find an optimal solution, it remains open if this is a general property of our approach.

References

1. Adacore. Technical report on OBC/MCDC properties. Technical report, Couverture project, 2010.
2. Faustin Ahishakiye, Svetlana Jakšić, Volker Stolz, Felix D. Lange, Malte Schmitz, and Daniel Thoma. Non-intrusive MC/DC measurement based on traces. In *International Symposium on Theoretical Aspects of Software Engineering (TASE)*, pages 86–92. IEEE, 2019.
3. Faustin Ahishakiye, José I. Requeno Jarabo, Lars Michael Kristensen, and Volker Stolz. Coverage analysis of net inscriptions in coloured Petri net models. In *International Conference on Verification and Evaluation of Computer and Communication Systems (VECoS)*, pages 68–83. Springer, 2020.
4. Zeina Awedikian, Kamel Ayari, and Giuliano Antoniol. MC/DC automatic test input data generation. In *Annual Conference on Genetic and Evolutionary Computation Conference (GECCO)*, pages 1657–1664. ACM, 2009.
5. Matteo Bordin, Cyrille Comar, T. Gingold, Jérôme Guitton, Olivier Hainque, and Thomas Quinot. Object and source coverage for critical applications with the COUVERTURE open analysis framework. In *European Congress Embedded Real Time Software and Systems (ERTS)*, pages 1–9, 2010.
6. Certification Authorities Software Team (CAST). Rationale for accepting masking MC/DC in certification projects. *Technical Report: Position Paper CAST-6*, 2001.
7. Certification Authorities Software Team (CAST). What is a “Decision” in application of Modified Condition/Decision Coverage (MC/DC) and Decision Coverage (DC)? *Technical Report: Position Paper CAST-10*, 2002.
8. Jun-Ru Chang and Chin-Yu Huang. A study of enhanced MC/DC coverage criterion for software testing. In *Annual International Computer Software and Applications Conference (COMPSAC)*, pages 457–464, 2007.
9. John J. Chilenski. An investigation of three forms of the Modified Condition Decision Coverage (MC/DC) criterion. Technical report, Office of Aviation Research, 2001.
10. John J. Chilenski and Steven P. Miller. Applicability of Modified Condition/Decision Coverage to software testing. *Software Engineering Journal*, 9(5):193–200, 1994.
11. Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.
12. Cyrille Comar, Jérôme Guitton, Olivier Hainque, and Thomas Quinot. Formalization and comparison of MC/DC and object branch coverage criteria. In *European Congress Embedded Real Time Software and Systems (ERTS)*, pages 1–10, 2011.
13. Christopher R. Drake. PyEDA: Data structures and algorithms for electronic design automation. In *Python in Science Conference (SciPy)*, 2015.
14. Gregory Gay, Ajitha Rajan, Matt Staats, Michael Whalen, and Mats P. E. Heimdahl. The effect of program and model structure on the effectiveness of MC/DC test adequacy coverage. *ACM Transactions on Software Engineering and Methodology*, 25(3), July 2016.
15. Gregory Gay, Matt Staats, Michael Whalen, and Mats P. E. Heimdahl. The risks of coverage-directed test case generation. *IEEE Transactions on Software Engineering*, 41(8):803–819, 2015.
16. Kamran Ghani and John A. Clark. Automatic test data generation for multiple condition and MC/DC coverage. In *International Conference on Software Engineering Advances (ICSEA)*, pages 152–157, 2009.

17. Hongfang Gong, Junyi Li, and Renfa Li. CTFTP: A test case generation strategy for general Boolean expressions based on ordered binary label-driven Petri nets. *IEEE Access*, 8:174516–174529, 2020.
18. Axel Halin, Alexandre Nuttinck, Mathieu Acher, Xavier Devroey, Gilles Perrouin, and Benoit Baudry. Test them all, is it worth it? Assessing configuration sampling on the JHipster Web development stack. *Empirical Software Engineering*, 24(2):674 –717, 2019.
19. Sylvain Hallé, Edmond La Chance, and Sébastien Gaboury. Graph methods for generating test cases with universal and existential constraints. In *International Conference on Testing Software and Systems (ICTSS)*, pages 55–70. Springer, 2015.
20. Alan J. Hu. Formal hardware verification with BDDs: an introduction. In *IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*, volume 2, pages 677–682. IEEE, 1997.
21. James A. Jones and Mary J. Harrold. Test-suite reduction and prioritization for Modified Condition/Decision Coverage. *IEEE Transactions on Software Engineering*, 29(3):195–209, 2003.
22. Akram Kalaee and Vahid Rafe. An optimal solution for test case generation using ROBDD graph and PSO algorithm. *Quality and Reliability Engineering International*, 32(7):2263–2279, 2016.
23. Susanne Kandl and Sandeep Chandrashekar. Reasonability of MC/DC for safety-relevant software implemented in programming languages with short-circuit evaluation. *Computing*, 97(30):261–279, Mar 2015.
24. Sekou Kangoye, Alexis Todoskoff, and Mihaela Barreau. Practical methods for automatic MC/DC test case generation of Boolean expressions. In *IEEE AUTOTESTCON*, pages 203–212. IEEE, 2015.
25. Takashi Kitamura, Quentin Maissonneuve, Eun-Hye Choi, Cyrille Artho, and Angelo Gargantini. Optimal test suite generation for Modified Condition Decision Coverage using SAT solving. In *Computer Safety, Reliability, and Security*, pages 123–138. Springer, 2018.
26. Christoph Meinel and Thorsten Theobald. *Algorithms and Data Structures in VLSI Design*. Springer-Verlag, Berlin, Heidelberg, 1st edition, 1998.
27. Jim Newton and Didier Verna. A theoretical and numerical analysis of the worst-case size of reduced ordered binary decision diagrams. *ACM Transactions on Computational Logic*, 20(1), January 2019.
28. Frederic Pothon. DO-178C/ED-12C versus DO-178B/ED-12B: Changes and Improvements. Technical report, AdaCore, 2012. available at https://www.adacore.com/books/do-178c-vs-do-178b.
29. Sherief Reda and Ashraf M. Salem. Combinational equivalence checking using Boolean satisfiability and binary decision diagrams. In *Design, Automation and Test in Europe. Conference and Exhibition (DATE)*, pages 122–126. IEEE, 2001.
30. Leanna Rierson. *Developing Safety-Critical Software: A Practical Guide for Aviation Software and DO-178C Compliance*. CRC Press, 2013.
31. Gregory Tassey. The economic impacts of inadequate infrastructure for software testing, 2002.
32. Sergiy Vilkomir and Jonathan Bowen. Reinforced Condition/Decision Coverage (RC/DC): A new criterion for software testing. In *International Conference of B and Z Users*, volume 2272 of *LNCS*, pages 291–308. Springer, 2002.
33. Elaine Weyuker, Tarak Goradia, and Ashutosh Singh. Automatically generating test data from a Boolean specification. *IEEE Transactions on Software Engineering*, 20(5):353–363, 1994.

34. Elaine J. Weyuker, Stewart N. Weiss, and Dick Hamlet. Comparison of program testing strategies. In *Symposium on Testing, Analysis, and Verification (TAV)*, pages 1–10. ACM, 1991.
35. James Worrell. Logic and Proofs-Binary Decision Diagrams. Available at https://www.cs.ox.ac.uk/people/james.worrell/lec5-2015.pdf.
36. Tianyong Wu, Jun Yan, and Jian Zhang. Automatic test data generation for unit testing to achieve MC/DC criterion. In *International Conference on Software Security and Reliability (SERE)*, pages 118–126. IEEE Computer Society, 2014.
37. Ling Yang, Jun Yan, and Jian Zhang. Generating minimal test set satisfying MC/DC criterion via SAT based approach. In *Annual ACM Symposium on Applied Computing (SAC)*, pages 1899–1906. ACM, 2018.

PAPER D

COVERAGE VISUALIZATION AND ANALYSIS OF NET INSCRIPTIONS IN COLOURED PETRI NET MODELS

Faustin Ahishakiye, José Ignacio Requeno Jarabo, Lars Michael Kristensen, Volker Stolz

Journal of Innovations in Systems and Software Engineering (ISSE), March, 2022 (Submitted, under review).

Coverage Visualization and Analysis of Net Inscriptions in Coloured Petri Net Models

Faustin Ahishakiye[1*], José Ignacio Requeno Jarabo[2†], Lars Michael Kristensen[1†] and Volker Stolz[1†]

[1*]Computer Science, Electrical Engineering, and Mathematical Sciences, Western Norway University of Applied Sciences, Inndalsveien 28, Bergen, 5063, Norway.
[2]Information Systems and Computing, Complutense University of Madrid, C/Prof. José García Santesmases, 9, Madrid, 28040, Spain.

*Corresponding author(s). E-mail(s): fahi@hvl.no; Contributing authors: jrequeno@ucm.es; lmkr@hvl.no; vsto@hvl.no;
†These authors contributed equally to this work.

Abstract

High-level Petri nets such as Coloured Petri Nets (CPNs) are characterised by the combination of Petri nets and a high-level programming language. In CPNs and CPN Tools, the inscriptions (e.g., arc expressions and guards) are specified using Standard ML. The application of simulation and state space exploration for validating CPN models traditionally focusses on behavioural properties related to net structure, i.e., places and transitions. This means that the net inscriptions are only implicitly validated, and the extent to which their sub-expressions have been covered is not made explicit. This paper extends our previous work for coverage analysis of programming languages via net inscriptions of CPN models. In particular, we upgrade the CPN Tools library responsible for annotating, instrumenting and collecting the evaluation of Boolean conditions for determining the coverage criteria based on model executions (runs). The library now automates most of the instrumentation parts that were done manually before, and integrates the reports of the coverage analysis inside the CPN Tools GUI. We evaluate our approach on new publicly available CPN models.

Keywords: Coverage analysis, MC/DC, CPN model, Testing

1 Introduction

Coverage analysis is important for programs in relation to fault detection. Structural coverage criteria are required for software safety and quality design assurance [1], and low coverage indicates that the software product has not been extensively tested. Two common metrics are statement- and branch coverage [2], where low coverage concretely indicates that certain instructions have never actually been executed. Coloured Petri Nets [3] and CPN Tools [4] have been widely used for constructing models of concurrent systems with simulation and state space exploration (SSE) being the two main techniques for dynamic analysis. CPN model analysis is generally concerned with behavioural properties related to boundedness, reachability, liveness, and fairness properties. This means that the main focus is on structural elements such as places, tokens, markings (states), transitions and transition bindings. Arc expressions and guards are only implicitly considered via the evaluation of these net inscriptions taking place as part of the computation of transition enabling and occurrence during model execution. This means that design errors in net inscriptions may not be detected as we do not obtain explicit information on for instance whether both branches of an if-then-else expression on an arc have been covered.

We argue that from a software engineering perspective, it is important to be explicitly concerned with quantitative and qualitative analysis of the extent to which net inscriptions have been covered. Our hypothesis is that the coverage criteria used for traditional source code can also be applied to the net inscriptions of CPN models. Specifically, we consider the modified condition decision coverage (MC/DC) criterion. MC/DC is a well-established coverage criteria for safety-critical systems, and is required by certification standards, such as the DO-178C [5] in the domain of avionic software systems. In the context of MC/DC, a *decision* is a Boolean expression composed of subexpressions and Boolean connectives (such as logical conjunction). A *condition* is an atomic (Boolean) expression. According to the definition of MC/DC [6, 2], each condition in a decision has to show an independent effect on that decision's outcome by: (1) varying just that condition while holding fixed all other possible conditions; or (2) varying just that condition while holding fixed all other possible conditions that could affect the outcome. MC/DC is a coverage criterion at the condition level and is recommended due to its advantages of being sensitive to code structure, requiring few test cases ($n + 1$ for n conditions), and it is the only criterion that considers the independence effect of each condition.

Coverage analysis for software is usually provided through dedicated instrumentation of the software under test, either by the compiler, or additional tooling, such as binary instrumentation. Transferring this to a CPN model under test, our aim is to combine the execution of a CPN model (by simulation or SSE) with coverage analysis of SML guard and arc expressions. Within CPN Tools, there is no coverage analysis of the SML expressions in a CPN model. This means that to record coverage data for a CPN model under test, it is necessary to instrument the Boolean expressions such that the truth-values of individual conditions are logged in addition to the overall outcome of the decision. Our approach to instrumentation makes use of side-effects by outputting intermediate results of conditions and decisions, which we then process to obtain the coverage verdict. No modifications to the net structure of the CPN model are necessary. Furthermore, the instrumentation has little impact on model execution so that it does not delay the simulation and SSE.

In this article, we extend our approach for coverage analysis of net inscriptions in CPN models [7] with the following new contributions:

1. We automate our instrumentation: it takes as input the original CPN model and produces an instrumented model where the Boolean expressions in guards and arcs are transformed into a form that emits log entries that are collected for coverage analysis. The automatic instrumentation also processes the definition of auxiliary SML functions, which were not considered in our manually instrumented solution.
2. We integrated a coverage visualization in CPN model such that a tester can observe which guard and arc expressions are covered or not. The covered parts are highlighted in green whereas the uncovered parts are shown in red with a possibility to see coverage percentage for each decision.
3. We test more CPN models and gather coverage statistics for seven additional CPN models publicly available.

The remainder of this paper is organised as follows. In Section 2, we introduce the MC/DC coverage criterion in more detail. In Section 3, we present our approach to deriving coverage data and show how to instrument guard and arc expressions to collect the required coverage data. In Section 4 we consider the post-processing of coverage data. We demonstrate the application of our library for coverage analysis on publicly available CPN models in Section 5. In this section, we also evaluate our approach with respect to overhead in execution, and discuss our findings. Section 6 discusses related work, and we present our conclusions including directions for future work in Section 7. Our coverage analysis library, the instrumented example models, the Python code to instrument, produce reports and graphs, and documentation is available at https://github.com/selabhvl/cpnmcdctesting.

2 Coverage Analysis and MC/DC

When considering CPN models, we will be concerned with coverage analysis of guard and arc coverage of expressions. A guard expression is a list of Boolean expressions all of which are required to evaluate to true in a given transition binding for the transition to be enabled. We refer to such Boolean expressions as *decisions*. Similarly, an if-then-else expression on an arc will have a decision determining whether the then- or the else-branch will be taken. Decisions are constructed from *conditions* and Boolean operators.

Definition 1 (Condition, Decision) A **condition** is a Boolean expression containing no Boolean operators except for the unary operator NOT.

A **decision** is a Boolean expression composed of conditions and zero or more Boolean operators. It is denoted by $\mathrm{D}(c_1, c_2, \cdots, c_i, \cdots, c_n)$, where c_i, $1 \leq i \leq n$ are conditions.

As an example, we may have a guard (or an arc expression) containing a decision of the form $D = (a \wedge b) \vee c$, where a, b, and c are conditions. These conditions may in turn refer to the values bound to the variables of the transition. The evaluation of a decision requires a *test case* assigning a *value* $\in \{0, 1, ?\}$ to the conditions of the decision, where ? means that a condition was not evaluated due to short-circuiting. Short-circuit means that the right operand of the *and*-operator (&&/$\wedge$) is not evaluated if the left operand is false, and the right operand of the *or*-operator (||/$\vee$) is not evaluated if the left operand is true.

Depending on the software safety level (A-D) which is assessed by examining the effects of a failure in the system, different structure coverage criteria are required. *Statement* coverage for software levels A-C, *branch/decision* coverage for software levels A-B, and MC/DC for software level A [2]. Statement coverage is considered inadequate because it is insensitive to some control structures. Both statement- and branch coverage are completely insensitive to the logical operators ($\vee$ and $\wedge$) [8]. The criteria taking logical expressions into consideration have been defined [1]. These are *condition coverage* (CC), where each condition in a decision takes on each possible outcome at least once true and once false during testing; *decision coverage* (DC) requiring only each decision to be evaluated once true and once false; and *multiple condition coverage* (MCC) which is an exhaustive testing of all possible input combinations of conditions to a decision. CC and DC are considered inadequate due to ignorance of the independence effect of conditions on the decision outcome. MCC requires 2^n tests for a decision with n inputs. This results in exponential growth in the number of test cases, and is therefore time-consuming and impractical for many test cases.

To address the limitations of the coverage criteria discussed above, *modified condition/decision coverage* (MC/DC) is considered and is required for safety critical systems such as in the avionics industry. MC/DC has been chosen as the coverage criterion for the highest safety level software because it is sensitive to the complexity of the decision structure [6] and requires only $n+1$ test cases for a decision with n conditions [1, 9]. In addition, MC/DC coverage criterion is suggested as a good candidate for model-based development (MBD) using tools such as Simulink and SCADE [10]. Thus, our model coverage analysis is based on MC/DC criterion. The following MC/DC definition is based on DO-178C [2]:

Definition 2 (Modified condition/decision coverage) A program is MC/DC covered and satisfies the MC/DC criterion if the following holds:

- every point of entry and exit in the program has been invoked at least once,
- every condition in a decision in the program has taken all possible outcomes at least once,
- every decision in the program has taken all possible outcomes at least once,
- each condition in a decision has shown to independently affect that decision's outcome by: (1) varying just that condition while holding fixed all other possible conditions, or (2) varying just that condition while holding fixed all other possible conditions that could affect the outcome.

The coverage of program entry and exit in the Definition 2 is added to all control-flow criteria and is not directly connected with the main point of MC/DC [11]. The most challenging and discussed part is showing the independent effect, which demonstrates that each condition of the decision has a defined purpose. The item (1) in the definition defines the unique cause MC/DC and item(2) has been introduced in the DO-178C to clarify that so-called *Masked MC/DC* is allowed [12, 5]. Masked MC/DC means that it is sufficient to show the independent effect of a condition by holding fixed only those conditions that could actually influence the outcome. Thus, in our analysis, we are interested in evaluation of expressions by checking the independence effect of each condition.

Example 1 Consider the decision $D = (a \wedge b) \vee c$. Table 1a presents all eight possible test cases (combinations) for MCC. The MC/DC pairs column for example, $c(1,2)$ specifies that from test case 1 and 2 we can observe that changing the truth value

TC	a	b	c	D	MC/DC pairs
1	0	0	0	0	
2	0	0	1	1	c(1,2)
3	0	1	0	0	
4	0	1	1	1	c(3,4)
5	1	0	0	0	
6	1	0	1	1	c(5,6)
7	1	1	0	1	a(3,7), b(5,7)
8	1	1	1	1	

(a) MCC test cases

TC	a	b	c	D	MC/DC pairs
1	0	?	0	0	
2	1	1	?	1	a(1,2)
3	1	0	0	0	b(2,3)
4	0	?	1	1	c(1,4)

(b) Selected MC/DC test cases

Table 1: MCC and selected MC/DC test cases for decision $D = (a \wedge b) \vee c$

of c while keeping the values of a and b, we can affect the outcome of the decision. Comparing MCC to MC/DC in terms of the number of test cases, there are seven possible MC/DC test cases (1 through 7) that are part of an MC/DC pair, where condition c is represented by three MC/DC pairs of test cases. However, for a decision with three conditions, only four (i.e., $n+1$) test cases are required to achieve MC/DC coverage as shown in Table 1b, where '?' represents the condition that was not evaluated due to short-circuiting.

3 Instrumentation of CPN models

In this section, we describe our instrumentation approach on an example CPN model, and highlight the salient features of our coverage analysis library. Our overall goal is that through simulation or SSE, we instrument and (partially) fill a truth-table for each decision in the net inscriptions of the CPN model. Then, for each of these tables, and hence the decisions they are attached to, we determine whether the model executions that we have seen so far satisfy the MC/DC coverage criteria. If MC/DC is not satisfied, either further simulations are necessary, or if the state space is exhausted, developers need to consider the reason for this short-coming, which may be related to insufficient exploration as per a limited set of initial markings, or a conceptual problem in that certain conditions indeed cannot contribute to the overall outcome of the decision.

3.1 MC/DC coverage for CPN models

MC/DC coverage (or any other type of coverage) is commonly used with executable programs: which decisions and conditions were evaluated by the test cases, and with which result. Specifically, these are decisions *from the source code* of the system (application) under test. Of course, a compiler may introduce additional conditionals into the code during code generation, but these are not of concern. CPN Tools already reports a primitive type of coverage as part of simulation (the transition and transition bindings that have been executed) and the state space exploration (transitions that have never occurred). These can be interpreted as variants of state- and branch coverage.

Hence, we first need to address what we want MC/DC coverage to mean in the context of CPN models. If we first consider guard expressions on transitions, then we have two interesting questions related to coverage: if there is a guard, we know from the state space (or simulation) report whether the transition has occurred, and hence whether the guard expression has evaluated to true. However, we do not know if during the calculation of enabling by CPN Tools it ever has been false. If the guard had never evaluated to false, this may indicate a problem in the model or the requirements it came from, since apparently that guard was not actually necessary. Furthermore, if a decision in a guard is a complex expression, then as per MC/DC, we would like to see evidence that each condition contributed to the outcome. Neither case can be deduced from the state space report or via the CTL model checker of CPN

Tools as the executions only contain transition bindings that have occurred, and hence cases where the guard has evaluated to true.

3.2 Automated Instrumentation of Net Inscriptions

In the following, we describe how we instrument the guards on transitions such that coverage data can be obtained. We developed an automated instrumentation based on the `.cpn` XML file of CPN Tools in combination with an SML parser. Arc expressions are handled analogously. Guards in a CPN model are written following the general form of a comma-separated list of Boolean expressions (decisions):

$$[bExp_0, \ldots, bExp_n]$$

A special case is the expression

$$var = exp$$

which may have two effects: if the variable `var` is bound already via a pattern in another expression (arc or guard) of the transition, then this is indeed a Boolean equality test (decision). If, however, `var` is not bound via other expressions, then this assigns the value of `exp` to the variable `var` and does not contribute to any guarding effect.

We consider general Boolean expressions which may make use of the full feature set of the SML language for expressions, most importantly Boolean binary operations, negation, conditional expressions with if-then-else and function calls. Simplified, we handle:

$\langle bExp \rangle ::=$ `not` $\langle bExp \rangle \mid \langle var \rangle \mid$ f $\langle exp \rangle_0 \ldots \langle exp \rangle_n$
$\quad \mid \langle bExp \rangle$ `andalso` $\langle bExp \rangle \mid \langle bExp \rangle$ `orelse` $\langle bExp \rangle$
$\quad \mid$ `if` $\langle bExp \rangle$ `then` $\langle bExp \rangle$ `else` $\langle bExp \rangle$
$\quad \mid$ `let` ... `in` $\langle bExp \rangle$ `end`

Function symbols `f` cover user-defined functions as well as (built-in) relational operators such as $<, =$; we do not detail the overall nature of arbitrary SML expressions, but refer the reader to [13] for a comprehensive discussion. The automatic instrumentation also processes the definition of SML functions in the body of the `.cpn` XML file, which were not considered in our manually instrumented solution. We do not provide instrumentation to measure coverage of pattern matching in function definitions and **`case`** expressions.

SSE or simulation of the model is not in itself sufficient to determine the outcome of the overall expression and its subexpressions: guards are not explicitly represented, and we only have the event of taking the transition in the state space, but no value of the guard expressions. Hence, we need to rely on side-effects during model execution to record the intermediate results. Our key idea is to transform every subexpression and the overall decision into a form which will use SML's file input/output to emit a log-entry that we can collect and analyse. The coverage statistics is calculated from the logged entries through a Python script that is easy to reuse in other contexts.

Listing 1: Expressions

```
datatype condition =
   AND of condition * condition
   OR of condition * condition
   NOT of condition
   ITE of condition * condition
                    * condition
   AP of string * bool;
```

Listing 2: Evaluation function

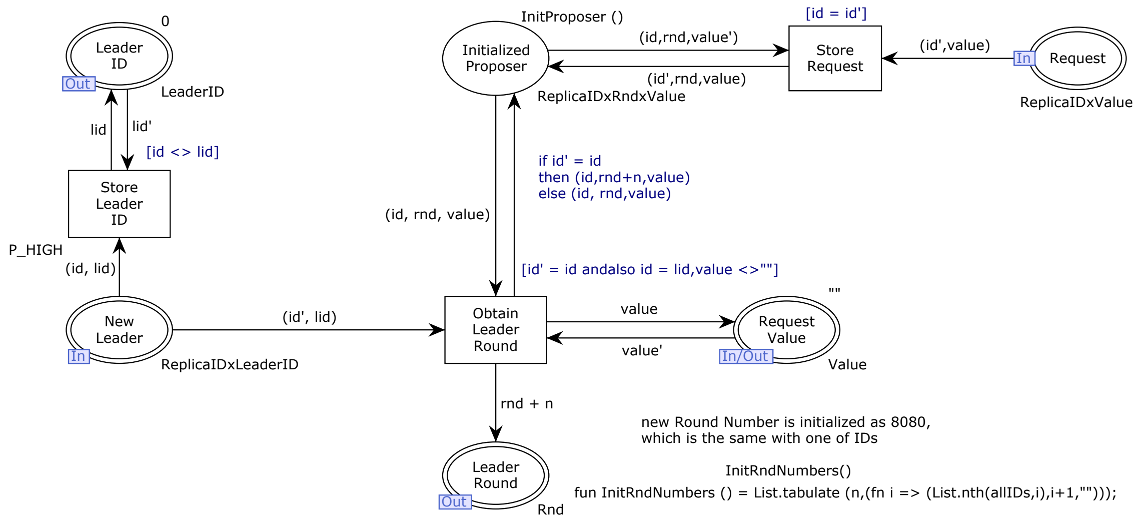
```
fun eval (AP (cond,v))=([(cond, SOME v)],v)
    eval (OR (a,b)) = let
val (ares,a') = eval a;
val (bres,b') = eval b;
in
(ares^^bres, a' orelse b')
end
...
fun EXPR (name,expr) : bool = [ ... ]
```

For the necessary instrumentation, a transformation of guard and arc expressions, we essentially create an interpreter for Boolean expressions: when guards are checked (in a deterministic order due to SML's semantics from left to right), we traverse a term representation of the Boolean expression and output the intermediate results. The Boolean expressions that are found in the definition of the SML functions will also trigger log messages during the SSE or simulation of the model.

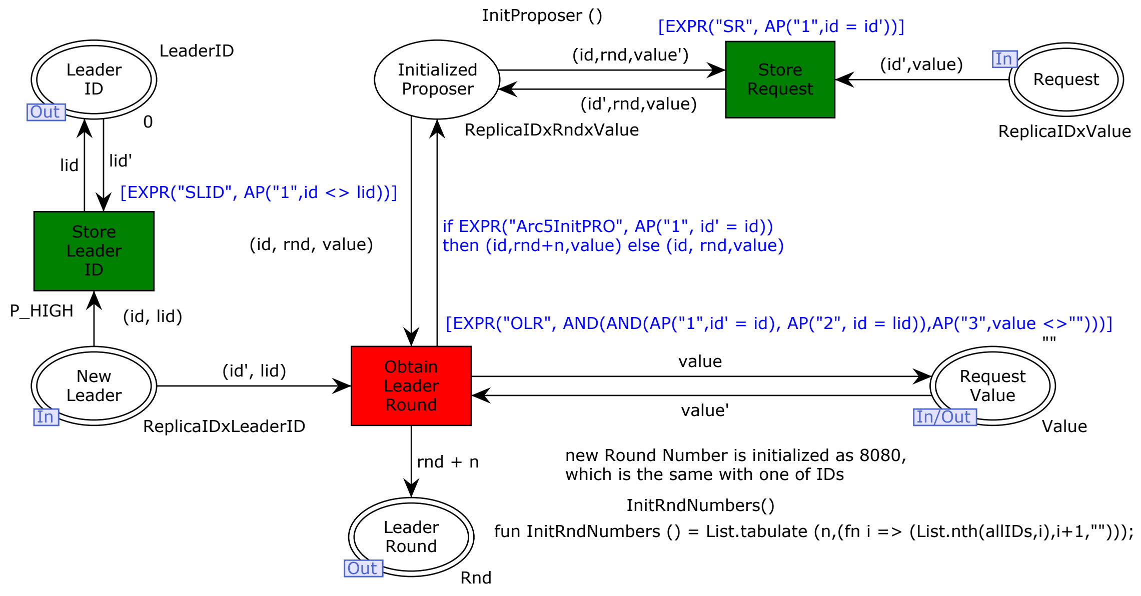
We have designed a data type (see Listing 1) that can capture the above constructs, and define an evaluation function (see Listing 2) on it. As we later need to map coverage reports back to code, for overall expressions `EXPR` and atomic proposition `AP` we introduce a component of type string that allows this identification. The evaluation function `eval` collects the result of intermediate evaluations in a list data structure, and the `EXPR` function (implementation not shown) turns this result into a single Boolean value that is used in the guard, and as a side-effect outputs the truth value outcome for individual conditions. As an example, if we consider a guard: `a>0` **`andalso`** `(b` **`orelse`** `(c=42));` then we can transform this guard in a straight forward manner into `EXPR("Gid", AND(AP("1", a>0), OR(AP("2", b), AP("3", c=42)))).` It is important to notice that this does not give us the (symbolic) Boolean expressions, as we still leave it to the standard SML semantics to evaluate the `a>0`, while abstractly we refer to the AP as a condition named "1". We elide expression- and proposition names for clarity in the text when not needed.

Any subexpression must be *total* and not crash and abort the model execution. A short-circuiting evaluation needs to explicitly incorporate the **`andalso`** or **`orelse`** operator and becomes more verbose, hence e.g. `x=0` **`orelse`** `(y/x >0.0)` becomes `OR (AP("O1", x=0)) (AP("O2", y/x > 0.0)).`

We can likewise apply the transformation to Boolean expressions in arc expressions: any Boolean expression is transformed into



(a) Original model


InitProposer ()
[EXPR("SR", AP("1",id = id'))]
LeaderID
Leader ID
Out
0
lid
lid'
[EXPR("SLID", AP("1",id <> lid))]
Store Leader ID
P_HIGH
(id, lid)
New Leader
In
ReplicaIDxLeaderID
(id', lid)
Initialized Proposer
(id,rnd,value')
(id',rnd,value)
ReplicaIDxRndxValue
Store Request
(id',value)
In
Request
ReplicaIDxValue
(id, rnd, value)
if EXPR("Arc5InitPRO", AP("1", id' = id))
then (id,rnd+n,value) else (id, rnd,value)
[EXPR("OLR", AND(AND(AP("1",id' = id), AP("2", id = lid)),AP("3",value <>"")))]
""
Obtain Leader Round
value
value'
Request Value
In/Out
Value
rnd + n
new Round Number is initialized as 8080,
which is the same with one of IDs
InitRndNumbers()
fun InitRndNumbers () = List.tabulate (n,(fn i => (List.nth(allIDs,i),i+1,"")));
Leader Round
Out
Rnd

(b) Model after instrumentation and visualization

Fig. 1: Paxos [14]: Guard and arc expressions before and after instrumentation

EXPR(...(AP ("A$_n$",bExp))...), resulting for example in the transformation of **if** bexp1 **orelse** bexp2 **then** e1 **else** e2 into
if EXPR("E1",OR(AP ("1",bexp1), AP ("2",bexp2))) **then** e1 **else** e2.

Figure 1 shows the sub-modules contained in the Paxos CPN model [14] called the initial Proposer and it is associated to the *InitProposer* substitution transition. It initializes Proposers to obtain a new leader, and receive a client request for consensus. Then, the value of the current round number of the leader and the value of the received client request will be presented on the port places as tokens, respectively [14].

The "InitProposer" module is one of several modules of the Paxos model, and the arc and guard expressions in the other modules were transformed in a similar manner. The figure also illustrates how, after evaluating coverage data, we indicate full coverage by colouring the guard green, otherwise red.

4 Post Processing of Coverage Data

We now discuss the coverage analysis which is performed via post-processing of the coverage data recorded through the instrumentation. We did not implement the MC/DC coverage analysis in SML directly. Rather, we feed individual observations about decision outcomes and their constituent conditions into a Python tool that computes the coverage results. This allows us to reuse the backend in other situations, without being SML or CPN specific.

4.1 Coverage Analysis

The general format from the instrumentation step is a sequence of colon-delimited rows, where each triple in a row captures a single decision with the truth values of all conditions in a fixed order and the outcome. As an example, see Script 4.1. The name (stemming from the first argument to an `EXPR` above) is configurable and should be unique in the model; and derived from the name of the element (guard or arc) the expression is attached to. This makes it easy to later trace coverage results for this name back to the element in the model. We recommend to derive the name from the element (guard or arc) the expression is attached to. This makes it easy to later trace coverage results for this name back to the element in the model, and for the user to navigate to the sub-module containing the element should they desire to do so.

Script 4.1: Log decisions

```
...
a3:01:0
t42:01110:0
t42:01011:1
...
```

Script 4.2: Decisions evaluation table

```
 ...
 Returna19
 0001       0
 0010       0
 0101       0
 0110       0
 1001       1
 1101       1
 1110       1
 ...
MCDC covered? False
R{1:[(0001, 1001), (0101, 1101), (0110,
1110)], 2:[], 3:[], 4:[]}
```

Script 4.1 shows that the decision "*t*42" was triggered twice, possibly on a guard which did not enable the transition (outcome indicating false), after which the exploration choose different transition bindings which resulted in a changed outcome of the 3rd and 5th condition in this decision and an overall outcome of true. We chose to print the binary representation instead of, e.g., a slightly shorter integer value to facilitate casual reading of the trace. Also, this allows us to enforce the correct number of bits that we expect per observation, corresponding to the number of conditions in the decision, which mitigates against instrumentation- or naming-mistakes.

Our Python tool parses the log file and calculates coverage information. It prints the percentage of decisions that are MC/DC and branch covered in textual mode and in GNU Plot syntax (see charts in Figure 3). The output contains individual reports in the form of the truth tables for each decision, which summarizes the conditions that are fired during the execution of the CPN model, and sets of pairs of test cases per condition that show the independence effect of that condition.

In the case that the decision is not MC/DC covered, the information provided by the Python script helps to infer the remaining valuations of the truth tables that should be evaluated in order to fulfil this criteria. In the example in Script 4.2, the first condition (left-most column in the table) has multiple complementary entries where the expression only varies in one bit (e.g., rows 0001 and 1001) and the output changes (0 to 1). The `R` set shows three such pairs for condition 1, but no complementary entries at all are found in the truth table for conditions 2, 3 and 4, and hence indicated as empty sets `[]` by Python. This information can then be used by developers to drill down into parts of their model, e.g. through simulation, that have not been covered adequately yet.

4.2 Combining Coverage Data from Multiple Runs

Coverage- or testing frameworks rely on their correct use by the operator, only a sub-class of tools such as fuzzers are completely automated. Our central `mcdcgen()` function only explores the state space for the current configuration as determined by the initial markings. Compared to regular testing of software, this corresponds to providing a single input to the system under test.

It is straightforward to capture executions of multiple runs of the Petri net: our API supports passing initialisation functions that reconfigure the net between runs. However, as there is no standardised way of configuring alternative initial markings or configurations in CPN Tools, the user has to actively make use of this API. In the default configuration, only the immediate net given in the model is evaluated, and no further alternative configurations are explored.

As an example, we show in Listing 3 how we make use of this feature in the MQTT-model, where alternative configurations were easily discoverable for us: the signature of MC/DC-generation with a simple test-driver is `mcdcgenConfig = fn : int*('a→'b)*'a list*string→unit`, where the first argument is a timeout for the SSE, the second is a function with side-effects that manipulates the global configurations that are commonly used in CPN Tools to parameterise models, the next argument is a list of different configurations, followed by the filename for writing results to.

Listing 3: MC/DC tool invocation

```
use (cpnmcdclibpath^"config/simrun.sml");
(* Invocation with default settings (no timeout) *)
mcdcgen("path/to/mqtt.log");
```

```
(* Invocation without timeout; base model + 2 configurations *)
mcdcgenConfig(0, applyConfig,[co1,co2],"path/to/mqtt3.log");
```

This function will always first evaluate the initial model configuration, and then have additional runs for every configuration. Internally, it calls into CPN Tools' `CalculateOccGraph()` function for the actual SSE. Hence the first `mcdcgen`-invocation in Listing 3 will execute a full SSE without timeout, whereas the second `mcdcgenConfig`-invocation would produce three subsequent runs logged into the same file, again without a default timeout. The test-driver can easily be adapted to different scenarios or ways of reconfiguring a model. Alternatively, traces can also be produced in separate files that are then concatenated for the coverage analysis.

Coverage Visualization in CPN Model

To visualize the coverage information in a graphical CPN model, we provide another Python script which parses the CPN model and changes the colour of guards in the CPN model based on coverage data. We take both the original model under test and the coverage results as input arguments and produce a new model where covered arcs and transitions are highlighted in green, whereas the uncovered parts are highlighted in red.

5 Evaluation on Example Models

In this section, we provide experimental results from an evaluation of our approach to model coverage for CPNs. We present the results of examining eleven (11) non-trivial CPN models from the literature that are freely available as part of scientific publications: a model of the Paxos distributed-consensus algorithm [14], a model of the MQTT publish-subscribe protocol [15], three models for distributed constraint satisfaction problem (DisCSP): weak-commitment search (WCS), asynchronous backtracking (ABT) and synchronous backtracking (SBT) algorithms [16], a complex model of the runtime environment of an actor-based model (CPNABS) [17], a reactor control system for a nuclear power plant (RCS-NPP) model and Niki T34 Syringe driver model [18]. In addition, we have tested four CPN models for test case generation from natural language requirements (NatCPN) [19]: nuclear power plant (NPP) model, turn indicator system (TIS) model, priority command (PC) model and vending machine (VM) model. All models come with initial markings that allow state space generation, in the case of MQTT, T34PIM and DisCSP complete, and incomplete in the case of Paxos, NatCPN and CPNABS.

5.1 Experimental Setup

Figure 2 gives an overview of our experimental setup. Initially, we have the original CPN model under test and we instrument it by transforming SML expressions into a form that as a side-effect prints how conditions were evaluated and the overall outcome of the decision (cf. Section 3). Second, we run the

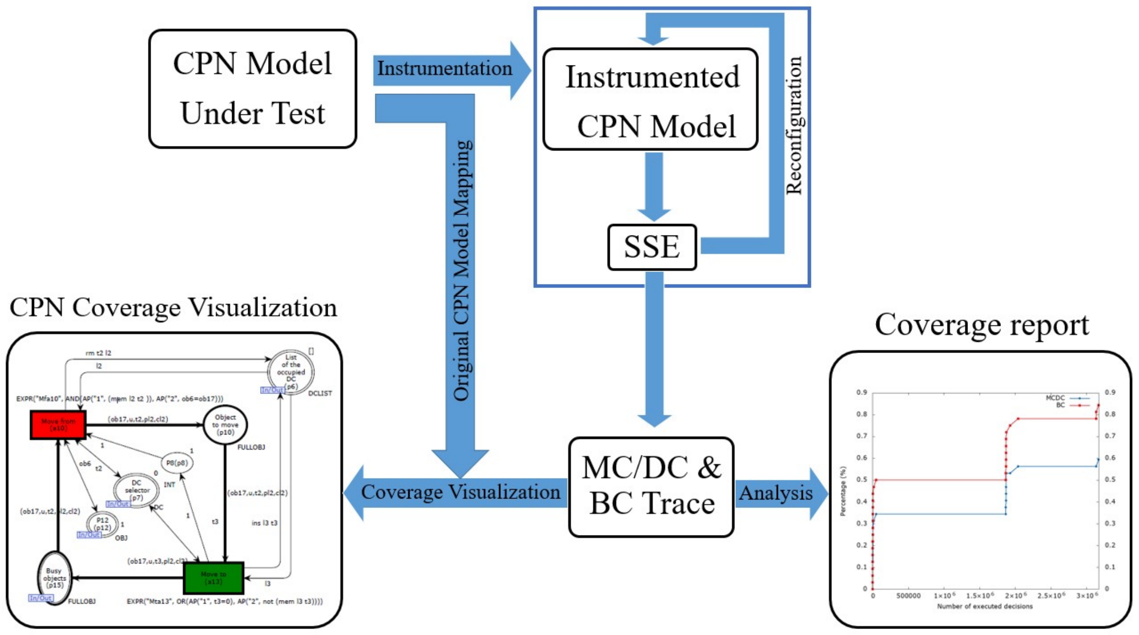


Fig. 2: Experimental setup for Coverage analysis for CPN models

SSE on the instrumented model and then reconfigure the configuration (initial marking) with any additional initial configurations if they are obvious from the model. As the side effect of SSE, we run the MC/DC generation which gives as output a log file containing the information of evaluations of conditions in arcs expressions and guards and the decision outcome. Finally, we run the MC/DC analyser (cf. Section 4) that determines whether each decision is MC/DC-covered or not. In addition, it reports the branch coverage (BC), by checking if each of the possible branches in each decision has been taken at least once.

Furthermore, we visualize the coverage information in the CPNs models taking as input the original CPN model and the results of how conditions and decisions are MC/DC evaluated. This results in the coloured CPN model where the covered parts are coloured in green and the uncovered parts are presented in red. Figure 1(a) shows a CPN model structure of an original model and Figure 1(b) shows the instrumented CPN model after coverage analysis where covered and uncovered parts are highlighted. Table 2 presents the summary of the percentage of how much the tested CPN models are MC/DC and BC covered. The percentage is calculated as the number of covered conditions over the total number of conditions in case of MC/DC and the ratio of covered decisions/branches and the total number of decisions/branches.

5.2 Experimental Results

Table 2 presents the experimental results for the eleven example models [17, 15, 14, 16, 18, 19]. For each model, we consider the number of executed decisions (second column) in arcs and guards. Column *Model decisions* refers to the number of decisions that have been instrumented in the model. The number of decisions observed in the model and in the log-file may deviate in case some of the decisions are never executed, in which case they will not appear in the

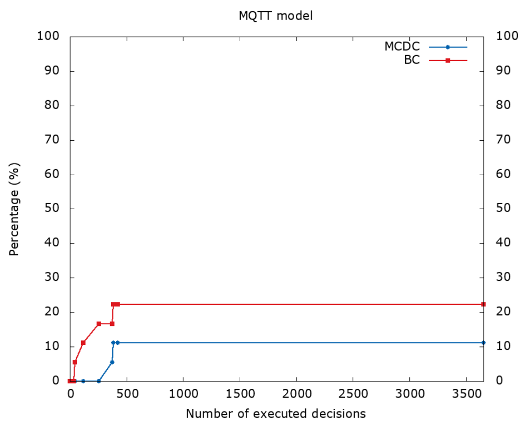
Table 2: MC/DC coverage results for example CPN models

CPN Model	Executed decisions	Model decisions	Non-trivial decisions	MC/DC (%)	BC (%)	Simulation status
Paxos	2,281,466	27	11	37.03	40.74	incomplete
MQTT-timeout	3,654	18	14	11.11	22.22	incomplete
MQTT-notimeout	1,828,751	23	19	21.73	65.22	complete
CPNABS	1,386,642	32	13	59.37	88.88	incomplete
DisCSP WCS	140680	9(2)	5	57.14	57.14	complete
DisCSP SBT	7686	7	3	57.14	57.14	complete
DisCSP ABT	604055	7	5	57.14	57.14	complete
NPP	194,481	13	13	53.84	92.3	incomplete
PC	8,677,800	10	9	90	90	incomplete
TIS	10,789,149	19	19	52.94	73.68	incomplete
VM	4,444	8	7	25	50	incomplete
T34PIM	3,644,768	23	8	69.56	82.6	complete

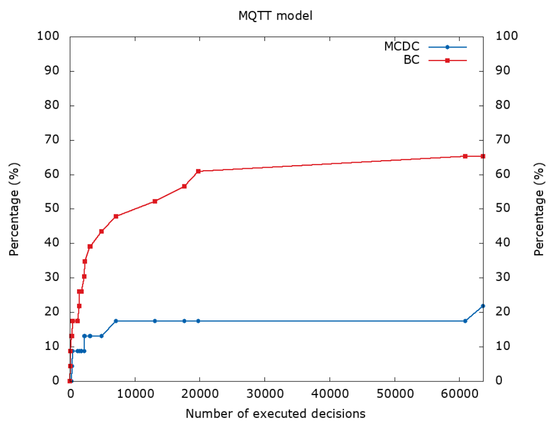
log file. We indicate them in brackets if during our exploration we did not visit, and hence log, each decision at least once. In the case of DisCSP, there are two guard decisions which were never executed. The column *Non-trivial decisions* gives the number of the decisions (out of all decisions) that have at least two conditions in the model, as they are the interesting ones while checking independence effect. If a decision has only one condition, it is not possible to differentiate MC/DC from DC. Columns MC/DC(%) and BC(%) present the coverage percentage for the CPN models under test. We record the ratio of covered decisions over the total number of decisions. Due to the large (maybe infinite) state space, we set the timeout to 600 seconds: in most models, running longer SSE do not increase the coverage metrics in terms of the number of arcs and guards expression executed.

5.3 Discussion of Results

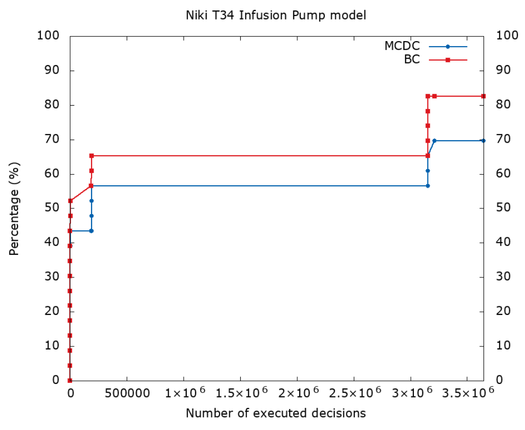
MC/DC is covered if all the conditions show the independence effect on the outcome. BC is covered if all the branches are taken at least once. This makes MC/DC a stronger coverage criterion compared to BC, which we will also see in the following graphs. Figure 3 shows the coverage results as the ratio of covered decisions and the number of executed decisions in guards and arcs for both MC/DC and BC. The plots show that the covered decisions increase as the model (and hence the decisions) is being executed. Note that the x-axis does not directly represent execution time of the model: the state space explorer prunes states that have been already visited (which takes time), and hence as the SSE progresses the number of expressions evaluated per time unit will decrease. In case an expression was executed with the same outcome, the coverage results do not increase, since those test cases have already been explored. Our instrumentation does not have a significant impact on the execution time of the model. Considering the time taken for the full SSE of the finite state models, for instance DisCSP model, both without and with instrumentation, it takes 212.346 seconds versus 214.922 seconds respectively. It is around 1% of overhead which is the cost for the instrumentation.



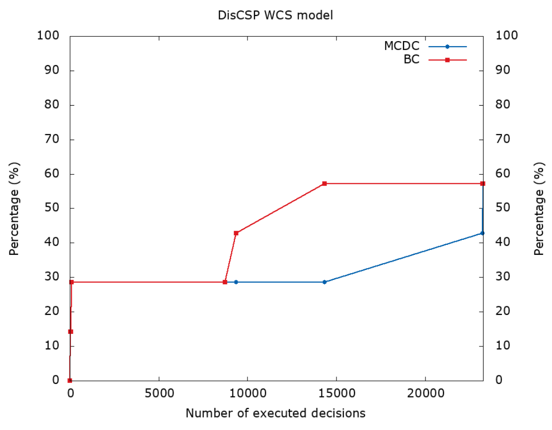
(a) MQTT model with timeout



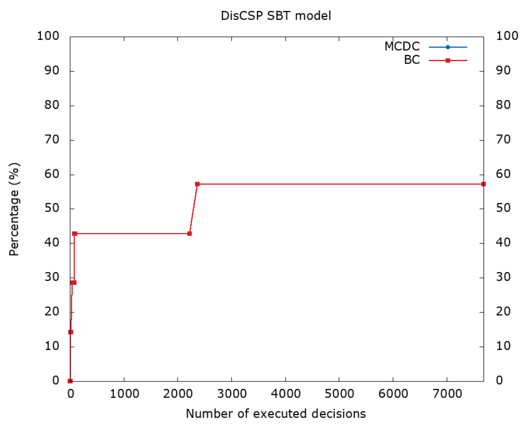
(b) MQTT model without timeout



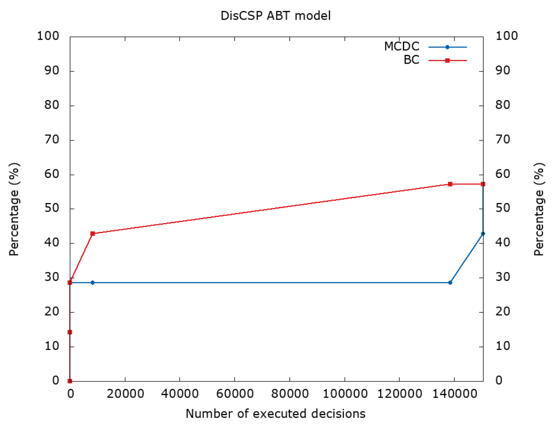
(c) T34PIM model



(d) DisCSP WCS model



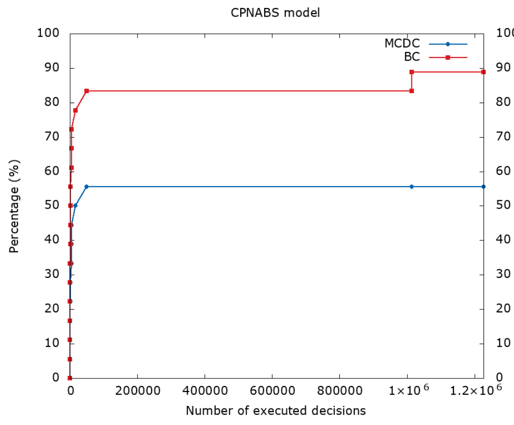
(e) DisCSP SBT model



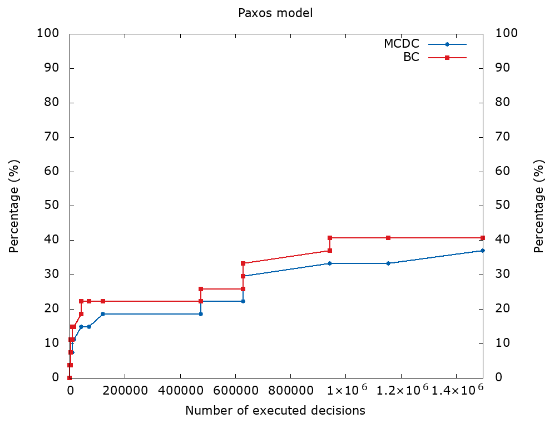
(f) DisCSP ABT model

Fig. 3: MC/DC and BC versus number of executed decisions: finite models

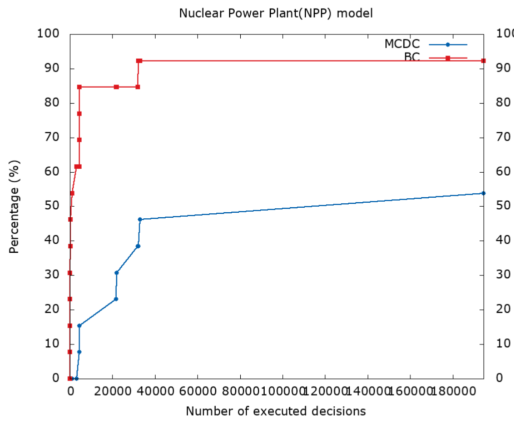
The CPNABS model and T34PIM model have many single condition (trivial) decisions, and their coverage percentage are higher compared to other models. The Paxos model has less than a half of its decisions covered for both BC and MC/DC with a small percentage difference. The VM model and MQTT with timeout have also less percentage in coverage and both have a high number of non-trivial decisions, which puts more weight on having a suitable test-suite to achieve good MC/DC coverage. In addition, we considered



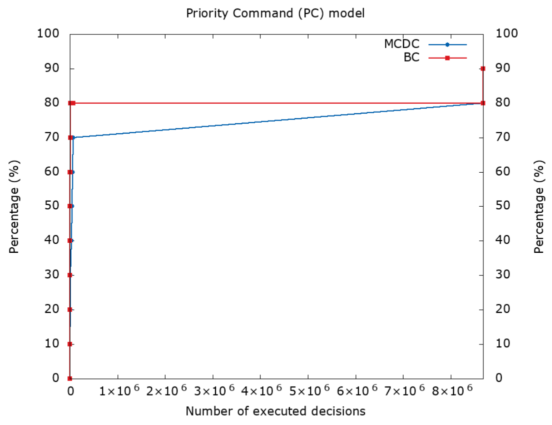
(a) CPN ABS model



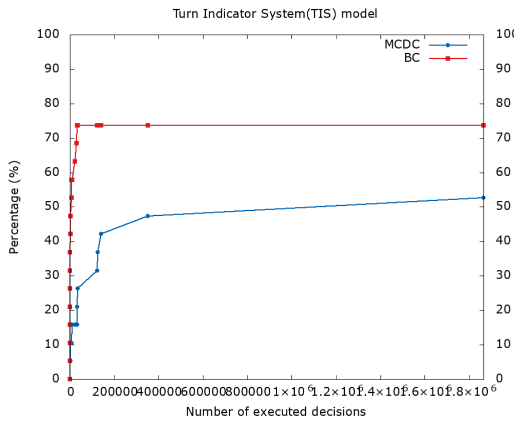
(b) Paxos model



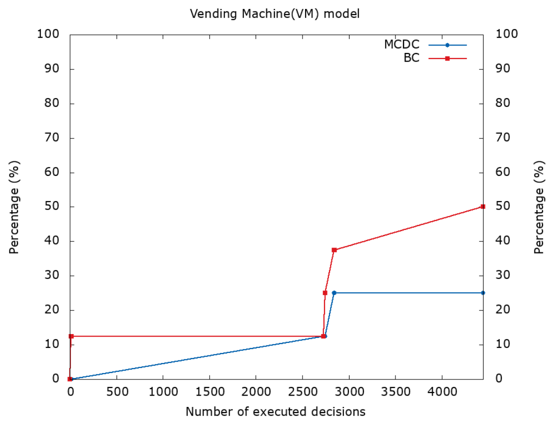
(c) NPP model



(d) PC model



(e) TIS model



(f) VM model

Fig. 4: MC/DC & BC versus number of executed decisions: Incomplete models

additional configurations without timeout for the SSE in the MQTT model and compared the coverage metrics when the configurations are set to timeout. As shown in Figure 3(a)&(b), the MC/DC and BC percentage increased from 11.11% to 21% and 22.22% to 65.22% respectively. It is interesting to observe the quality differences of the curves for the tested models. Some of the tested models have less than half of their decisions covered. This should attract the attention of developers and they should assess whether they have tested their models enough, as these results indicate that there is something that might be considered doubtful and require to revisit their test-suite. Two factors affect the coverage percentage results presented for these models:

1. The tested models had no clear test suites; they might be lacking test cases to cover the remaining conditions. Depending on the purpose of each model, some of the test cases may not be relevant for the model or the model may not even have been intended for testing. This could be solved by using test case generation for uncovered decisions (see our future work).
2. The models might be erroneous in the sense that some parts (conditions) in the model are never or only partially executed due to a modelling issue, e.g. if the model evolved and a condition no longer serves any purpose or is subsumed by other conditions. For example in the DisCSP model, there are two decisions which were never executed, and we cannot tell if this was intentionally or not without knowing the goal of the developers.

A main result of our analysis of the example models is that none of the models (including those for which the state space could be fully explored) have full MC/DC or BC. This confirms our hypothesis that code coverage of net inscriptions of CPN models can be of interest to developers, such as revealing not taken branches of the if-then-else arc expressions, never executed guard decisions, conditions that do not independently affect the outcome and some model design errors. Our results show that even for full SSE, we may still find expressions that are not MC/DC covered. Assuming that the model is correct, improving coverage then requires improving the test suite. This confirms the relevance and added value of performing coverage analysis of net inscriptions of CPN models over the dead places/transitions report as part of the state space generation. A natural next step in a model development process would be for the developers to revisit the decisions that are not MC/DC covered and understand the underlying reason. For the models that we have co-published, we can indeed confirm that the original models were not designed with a full test-suite in mind, neither from the initial configuration, nor through embedded configurations like for example the MQTT model.

6 Related Work

Coverage analysis has attracted attention in both academic and industrial research. Especially the MC/DC criterion is highly recommended and commonly used in safety critical systems, including avionic systems [5]. However, there is a limited number of research addressing model-based coverage analysis. Ghosh [20] expresses test adequacy criteria in terms of model coverage and explicitly lists *condition coverage* and *full predicate coverage criterion* for OCL predicates on UML interaction diagrams, which are semantically related to CPNs in that they express (possible) interactions. Test cases were not automatically generated. In [21], the authors present an automated test generation technique, MISTA (Model-based Integration and System Test Automation) for integrated functional and security testing of software systems using high-level Petri nets as finite state test models. None of the above works addressed structural coverage analysis such as MC/DC or BC on CPN models.

MC/DC is not a new coverage criterion. Chilenski [9] investigated three forms of MC/DC including Unique-Cause (UC) MC/DC, Unique-Cause + Masking MC/DC, and Masking MC/DC. Moreover, other forms of MC/DC have been discussed in [22]. More than 70 papers were reviewed and 54 of them discussed MC/DC definitions and the remaining were only focusing on the use of MC/DC in faults detection. We presented in [23], a tool that measures MC/DC based on traces of C programs without instrumentation.

Simulink [24] supports recording and visualising various coverage criteria including MC/DC from simulations via the Simulink Design Verifier. It also has two options for creating test cases to account for the missing coverage in the design. Test coverage criteria for autonomous mobile systems based on CPNs ware presented by Lill et al. in [25]. Their model-based testing approach is based on the use of CPNs to provide a compact and scalable representation of behavioural multiplicity to be covered by an appropriate selection of representative test scenarios fulfilling net-based coverage criteria. Simão et al. [26] provide definitions of structural coverage criteria family for CPNs, named CPN Coverage Criteria Family. These coverage criteria are based on checking if all-markings, all-transitions, all-bindings, and all-paths are tested at least once. Our work is different from the above presented work in that we are analysing the coverage of net inscriptions (conditionals in SML decisions) in CPN models based on structure coverage criteria defined by certification standards, such as DO-178C [2].

7 Summary and Outlook

We have extended our earlier proof of concept [7] and the supporting software tool to measure MC/DC and branch coverage (BC) of SML decisions in CPN models. There are three main contributions in this paper: 1) We provide a library and automated annotation mechanism that intercept evaluation of Boolean conditions in guards and arcs in SML decisions in CPN models, and record how they were evaluated; 2) we compute the conditions' truth assignments and check whether or not particular decisions are MC/DC-covered in the recorded executions of the model; and 3) we collect coverage data using our library from eleven publicly available CPN models and report whether they are MC/DC and BC covered.

We have tested more CPN models and have improved the usability of our instrumentation with respect to the previous release. Firstly, we automate the annotation mechanism that intercepts evaluation of Boolean conditions, which had to be done manually before. We also support the instrumentation of Boolean decision not only in the arcs and guards of CPN models, but also in any SML decision (e.g., function declarations). Secondly, the new release better integrates the coverage analysis tool with the graphical user interface CPN Tools, which supports a broad palette of visual options to indicate successful coverage of guards through colour based on different coverage criteria (MC/DC, BC,...). We leave the encoding of partial functions into delayed evaluation using so-called *thunks* in SML as future work since it did not pose a problem yet in our example models. Thunks wrap expressions into a constant function that needs to be called to trigger evaluation, and can hence be passed around safely as arguments. As an example, consider `[List.length xs > 0, hd xs]`, which is a valid chain of guards, but will crash in our instrumentation when the list `xs` is empty.

Our experimental results show that our library and post-processing tool can find how conditions were evaluated in all the net inscriptions in CPN models and measure MC/DC and BC. Results reveal that the MC/DC coverage percentage is quite low for the CPN models tested. This is interesting because it indicates that developers may have had different goals when they designed the model, and that the model only reflects a single starting configuration. We can compare this with the coverage of regular software: running a program will yield *some* coverage data, yet most programs will have to be run with many different inputs to achieve adequate coverage.

To the best of our knowledge, our approach is the first work on coverage analysis of CPN models based on BC and MC/DC criteria. This work highlighted that coverage analysis is interesting for CPN models, not only in the context of showing the covered guard and arcs SML decisions, but also the effect of conditionals in SML decisions on the model outcome and related potential problems.

Outlook.

Our general approach to coverage analysis presents several directions forward which would help developers get a better understanding of their models: firstly, while generating the full state space is certainly the preferred approach, this is not feasible if the state space is inherently infinite or too large. Simulation of particular executions could then be guided by results from the coverage and try to achieve higher coverage in parts of the model that have not been explored yet. However, while selecting particular transitions to follow in a simulation is straight-forward, manipulating the data space for bindings used in guards is a much harder problem and closely related to test case generation (recall the CPNs also rely on suitable initial states, which are currently given by developers). Making use of feedback between the state of the simulation and the state of the coverage would, however, require much tighter integration of the tools.

As for the measured coverage results, it would be interesting to discuss with the original developers of the models if the coverage is within their expectations. While on the one hand low coverage could indicate design flaws, on the other hand our testing may not have exercised the same state space as the original developers did: they may have used their model in various configurations, whereof the state of the `git` repository just represents a snapshot, or we did not discover all possible configurations in the model. In the future, we may also try to generate test-cases specifically with the aim to increase coverage.

References

[1] Hayhurst, K.J., Veerhusen, D.S., Chilenski, J.J., Rierson, L.K.: A Practical Tutorial on Modified Condition/Decision Coverage. Technical Report NASA/TM-2001-210876, NASA Langley Server (2001). https://dl.acm.org/doi/book/10.5555/886632

[2] Rierson, L.: Developing Safety-Critical Software: A Practical Guide for Aviation Software and DO-178C Compliance, 1st edn., pp. 13–46. CRC Press, USA (2013)

[3] Jensen, K., Kristensen, L.M.: Colored Petri Nets: A graphical language for formal modeling and validation of concurrent systems. Commun. ACM **58**, 61–70 (2015). https://doi.org/10.1145/2663340

[4] Jensen, K., Christensen, S., Kristensen, L.M., Michael, W.: CPN Tools (2010). http://cpntools.org/

[5] Pothon, F.: DO-178C/ED-12C versus DO-178B/ED-12B: Changes and Improvements. Technical report, AdaCore (2012)

[6] John J., C., Steven P., M.: Applicability of Modified Condition/Decision Coverage to software testing. Software Engineering Journal **9**(5), 193–200 (1994)

[7] Ahishakiye, F., Jarabo, J.R., Kristensen, L.M., Stolz, V.: Coverage Analysis of Net Inscriptions in Coloured Petri Net Models. In: Hedia, B.B., Chen, Y., Liu, G., Yu, Z. (eds.) 14th Intl. Conf. on Verification and Evaluation of Computer and Communication Systems (VECOS). LNCS, vol. 12519, pp. 68–83. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-65955-4_6

[8] Cornett, S.: Code Coverage Analysis. available at https://www.bullseye.com/coverage.html, Accessed 6 June 2020 (1996-2014)

[9] John J., C.: An investigation of three forms of the Modified Condition Decision Coverage (MC/DC) criterion. Technical report, Office of Aviation Research (2001)

[10] Heimdahl, M.P.E., Whalen, M.W., Rajan, A., Staats, M.: On MC/DC and implementation structure: An empirical study. In: Proc. of IEEE/AIAA 27th Digital Avionics Systems Conference, pp. 5–315313 (2008). https://doi.org/10.1109/DASC.2008.4702848

[11] Vilkomir, S., Bowen, J.: Reinforced condition/decision coverage (RC/DC): A new criterion for software testing. In: Proc. of ZB 2002:Formal Specification and Development in Z and B. LNCS, vol. 2272, pp. 291–308. Springer, Berlin, Heidelberg (2002). https://doi.org/10.1007/3-540-45648-1_15

[12] Certification Authorities Software Team (CAST): Rationale for Accepting Masking MC/DC in Certification Projects. Technical report, Position Paper CAST-6 (2001)

[13] Tofte, M.: Standard ML language. Scholarpedia **4**(2), 7515 (2009). https://doi.org/10.4249/scholarpedia.7515

[14] Wang, R., Kristensen, L.M., Meling, H., Stolz, V.: Automated test case generation for the Paxos single-decree protocol using a Coloured Petri Net model. J. Logical and Algebraic Methods in Programming **104**, 254–273 (2019). https://doi.org/10.1016/j.jlamp.2019.02.004

[15] Rodríguez, A., Kristensen, L.M., Rutle, A.: Formal Modelling and Incremental Verification of the MQTT IoT Protocol. In: Proc. of Trans. Petri Nets and Other Models of Concurrency. LNCS, vol. 11790, pp. 126–145. Berlin, Heidelberg (2019). https://doi.org/10.1007/978-3-662-60651-3_5

[16] Pascal, C., Panescu, D.: A Colored Petri Net model for DisCSP algorithms. Concurr. Comput. Pract. Exp. **29**(18), 1–23 (2017)

[17] Gkolfi, A., Din, C.C., Johnsen, E.B., Kristensen, L.M., Steffen, M., Yu, I.C.: Translating active objects into Colored Petri Nets for communication analysis. Science of Computer Programming **181**, 1–26 (2019). https://doi.org/10.1016/j.scico.2019.04.002

[18] Caesarea Medical Electronics: Niki T34 syringe pump instruction manual (2008). https://manuals.plus/cme/cme-niki-t34-stringe-pump-manual-pdf

[19] Silva, B.C.F., Carvalho, G., Sampaio, A.: Test case generation from natural language requirements using CPN simulation. In: SBMF. LNCS, vol. 9526, pp. 178–193. Springer, Berlin, Heidelberg (2015). https://doi.org/10.1007/978-3-319-29473-5_11

[20] Sudipto Ghosh, France, R., Braganza, C., Nilesh Kawane, Andrews, A., Orest Pilskalns: Test adequacy assessment for UML design model testing. In: Proc. of 14th Intl. Symp. on Software Reliability Engineering, ISSRE'03., pp. 332–343 (2003). https://doi.org/10.1109/ISSRE.2003.1251054

[21] Xu, D., Xu, W., Kent, M., Thomas, L., Wang, L.: An automated test generation technique for software quality assurance. IEEE Reliab. **64**(1), 247–268 (2015). https://doi.org/10.1109/TR.2014.2354172

[22] Paul, T.K., Lau, M.F.: A systematic literature review on Modified Condition and Decision Coverage. In: Proc. of the 29th Annual ACM Symp. on Applied Computing. SAC '14, pp. 1301–1308. Association for Computing Machinery, New York, USA (2014). https://doi.org/10.1145/2554850.2555004

[23] Ahishakiye, F., Jakšić, S., Stolz, V., Lange, F.D., Schmitz, M., Thoma, D.: Non-intrusive MC/DC measurement based on traces. In: Méry, D., Qin, S. (eds.) Intl. Symp. on Theoretical Aspects of Software Engineering, pp. 86–92. IEEE, Guilin, China (2019). https://doi.org/10.1109/TASE.2019.00-15

[24] Simulink: Types of Model Coverage. Accessed 06 April 2022. https://se.mathworks.com/help/slcoverage/ug/types-of-model-coverage.html

[25] Lill, R., Saglietti, F.: Model-based Testing of Cooperating Robotic Systems using Coloured Petri Nets. In: Proc. of SAFECOMP 2013 - Workshop DECS (ERCIM/EWICS Workshop on Dependable Embedded and Cyber-physical Systems) of the 32nd Intl. Conf. on Computer

Safety, Reliability and Security, Toulouse, France (2013). https://hal.archives-ouvertes.fr/hal-00848597

[26] Simão, A., Do, S., Souza, S., Maldonado, J.: A family of coverage testing criteria for Coloured Petri Nets. In: Proc. of 17th Brazilian Symposium on Software Engineering (SBES'2003), pp. 209–224 (2003)