Department of Computer Science

Model Checking Large Nuclear Power Plant Safety System Designs

Jussi Lahtinen

Aalto University

DOCTORAL DISSERTATIONS

Aalto University publication series
DOCTORAL DISSERTATIONS 159/2016
VTT SCIENCE 133

Model Checking Large Nuclear Power Plant Safety System Designs

Jussi Lahtinen

A doctoral dissertation completed for the degree of Doctor of Science (Technology) to be defended, with the permission of the Aalto University School of Science, at a public examination held at the lecture hall T2 of the school on 7 October 2016 at 12.

Aalto University
School of Science
Department of Computer Science

Supervising professor
Assoc. Prof. Keijo Heljanko

Thesis advisor
Assoc. Prof. Keijo Heljanko

Preliminary examiners
Assoc. Prof. Jiří Barnat, Masaryk University, Czech Republic
Prof. Lars M. Kristensen, Bergen University College, Norway

Opponent
Prof. Gerald Lüttgen, University of Bamberg, Germany

Aalto University publication series
DOCTORAL DISSERTATIONS 159/2016

VTT SCIENCE 133

ISBN 978-952-60-6959-3 (printed)
ISBN 978-952-60-6958-6 (pdf)
ISSN-L 1799-4934
ISSN 1799-4934 (printed)
ISSN 1799-4942 (pdf)
http://urn.fi/URN:ISBN:978-952-60-6958-6

ISBN 978-951-38-8448-2 (printed)
ISBN 978-951-38-8447-5 (pdf)
ISSN-L 2242-119X
ISSN 2242-119X (printed)
ISSN 2242-1203 (pdf)
http://urn.fi/URN:ISBN:978-951-38-8447-5

Unigrafia Oy
Helsinki 2016

Finland

NORDIC ECOLABEL
441 697
Printed matter

Aalto University

Abstract

Aalto University, P.O. Box 11000, FI-00076 Aalto www.aalto.fi

Author
Jussi Lahtinen

Name of the doctoral dissertation
Model Checking Large Nuclear Power Plant Safety System Designs

Publisher School of Science

Unit Department of Computer Science

Series Aalto University publication series DOCTORAL DISSERTATIONS 159/2016

Field of research Theoretical Computer Science

Manuscript submitted 4 May 2016 **Date of the defence** 7 October 2016

Permission to publish granted (date) 4 July 2016 **Language** English

☐ **Monograph** ☒ **Article dissertation** ☐ **Essay dissertation**

Abstract

Digital instrumentation and control (I&C) systems are increasingly being used for implementing safety-critical applications such as nuclear power plant safety systems. The exhaustive verification of these systems is challenging, and verification methods such as testing and simulation are typically insufficient. Model checking is a formal method for verifying the correctness of a system design model. The requirements of the system are formalised using temporal logic, and the behaviour of the system model is exhaustively analysed with respect to these formal specifications. The method is very effective in finding hidden design errors.

Model checking is computationally very demanding, and thus one of the challenges in applying model checking is its scalability. This dissertation discusses the verification of larger systems implementing multiple functions using model checking. First of all, this dissertation presents methodology for modelling safety system designs, and describes a simple abstraction technique for models of these systems that utilises modular over-approximating abstractions. Furthermore, the dissertation presents the development of an iterative abstraction refinement algorithm for the purpose of automatically finding an abstraction level suitable for verification. This dissertation also studies hardware failures, and creates an extension of the safety system modelling methodology that enables the analysis of fault-tolerance properties in large many-redundant system assemblies. The methodology follows closely the conventions of probabilistic risk assessment (PRA), and serves as a first step for further integration between model checking and PRA. Finally, this work presents the development of a test set generation technique based on model checking that utilises the structure of function block diagram (FBD) programs.

The results of this work have a high significance to safety because the developed techniques can be used to verify the correctness of safety system designs used in nuclear power plants. The work has also improved the scalability and applicability of model checking, and can be seen as part of a continuum toward larger plant-level models and toward new all-encompassing safety analysis approaches.

Keywords model checking, automation, nuclear, PLC, Function Block Diagram, fault-tolerance, instrumentation and control I&C, iterative abstraction refinement, compositional minimization, formal verification, safety system, structure-based testing, test generation

ISBN (printed) 978-952-60-6959-3 **ISBN (pdf)** 978-952-60-6958-6

ISSN-L 1799-4934 **ISSN (printed)** 1799-4934 **ISSN (pdf)** 1799-4942

Location of publisher Helsinki **Location of printing** Helsinki **Year** 2016

Pages 230 **urn** http://urn.fi/URN:ISBN:978-952-60-6958-6

Aalto-yliopisto

Tiivistelmä

Aalto-yliopisto, PL 11000, 00076 Aalto www.aalto.fi

Tekijä
Jussi Lahtinen

Väitöskirjan nimi
Ydinvoimaloiden laajojen turva-automaatiojärjestelmien mallintarkastus

Julkaisija Perustieteiden korkeakoulu

Yksikkö Tietotekniikan laitos

Sarja Aalto University publication series DOCTORAL DISSERTATIONS 159/2016

Tutkimusala Tietojenkäsittelyteoria

Käsikirjoituksen pvm 04.05.2016 **Väitöspäivä** 07.10.2016

Julkaisuluvan myöntämispäivä 04.07.2016 **Kieli** Englanti

☐ **Monografia** ☒ **Artikkeliväitöskirja** ☐ **Esseeväitöskirja**

Tiivistelmä

Monet turvakriittiset sovellukset kuten ydinvoimaloissa käytetyt turva-automaatiojärjestelmät perustuvat yhä enenevässä määrin digitaaliseen ohjelmoitavaan tekniikkaan. Tällaisten digitaalisten järjestelmien verifiointi on erittäin haastavaa, eivätkä perinteiset menetelmät kuten testaus ja simulointi usein kykene saavuttamaan täydellistä kattavuutta. Mallintarkastus on formaali menetelmä, jota käytetään järjestelmän verifioinnin apuvälineenä. Mallintarkastuksessa järjestelmän toiminnalliset vaatimukset muodostetaan aikalogiikan lauseiden avulla, ja vaatimusten täyttyminen tarkastetaan käymällä systemaattisesti läpi kaikki järjestelmästä laaditun mallin käyttäytymiset. Menetelmä on erittäin tehokas löytämään piileviä suunnitteluvirheitä.

Mallintarkastus on laskennallisesti vaativa menetelmä, ja eräs menetelmän soveltamiseen liittyvä haaste on sen skaalautuvuus. Tässä väitöstyössä tutkitaan mallintarkastuksen soveltamista useiden alijärjestelmien muodostamien laajojen kokonaisuuksien verifiointiin. Työssä on luotu metodologiaa turvajärjestelmien mallintamiseen, sekä tehty mallinnustavan kanssa yhteensopiva modulaarinen abstraktiomenetelmä, joka perustuu moduulien yliapproksimatiivisiin abstraktioihin. Lisäksi väitöstyössä on kehitetty iteratiivinen tekniikka, jonka tarkoituksena on etsiä verifiointiin sopiva abstraktiotaso automaattisesti. Työssä on myös tutkittu laitteistovikojen mallintamista ja kehitetty turvajärjestelmien mallinnuksen kanssa yhteensopiva mallinnustekniikka, joka mahdollistaa järjestelmän vikasietoisuuteen liittyvien ominaisuuksien tarkastelemisen laajoissa moniredundanttisissa järjestelmissä. Tekniikka mukailee todennäköisyysperusteisen riskianalyysin (PRA, probabilistic risk assessment) käyttämiä tapoja jäsentää vikaantumiseen liittyviä ongelmia, ja on täten myös askel kohti näiden kahden menetelmän syvempää integraatiota. Viimeiseksi, työssä on kehitetty mallintarkastusta hyödyntävä tekniikka, jonka avulla voidaan automaattisesti luoda joukko testejä toimilohkokaavion (function block diagram, FBD) rakenteen perusteella.

Väitöstyön tulokset ovat tärkeitä turvallisuuden kannalta, sillä kehitettyjä tekniikoita voidaan käyttää varmistamaan turva-automaatiojärjestelmien suunnittelun virheettömyys. Väitöstyön myötä mallintarkastuksen skaalautuvuus ja käyttökelpoisuus ovat myös parantuneet. Työn voi nähdä osana jatkumoa kohti yhä suurempia laitostason malleja, ja uusia kokonaisvaltaisia tapoja analysoida turvallisuutta.

Avainsanat mallintarkastus, automaatio, ydinvoima, toimilohko, vikasietoisuus, abstraktio, formaali verifiointi, turvajärjestelmä, rakenteellinen testaus, testigenerointi

ISBN (painettu) 978-952-60-6959-3 **ISBN (pdf)** 978-952-60-6958-6

ISSN-L 1799-4934 **ISSN (painettu)** 1799-4934 **ISSN (pdf)** 1799-4942

Julkaisupaikka Helsinki **Painopaikka** Helsinki **Vuosi** 2016

Sivumäärä 230 **urn** http://urn.fi/URN:ISBN:978-952-60-6958-6

Contents

Preface

Model checking has been an interest of mine for quite a long time. I did my Master's thesis in 2007 on model checking of safety instrumented systems using timed automata but I had been introduced to model checking during my university studies even before that. Even though I officially enrolled for postgraduate studies as late as 2010, the actual research work presented in this dissertation started already around 2008 when I was hired as a research scientist at VTT.

I would like to express my gratitude to the people and organisations that have supported me throughout my doctoral work. To start with, the funding of the work was provided primarily by three different SAFIR (Finnish National Research Program on Nuclear Safety) programmes: SAFIR2010, SAFIR2014, and SAFIR2018. The funding of my work within the programmes comes from the Finnish state nuclear waste management fund (VYR, Valtion Ydinjätehuoltorahasto) collected annually from the Finnish utilities Fortum Oyj, Teollisuuden Voima Oyj, and Fennovoima Oy. Research in the SAFIR programmes is also funded by VTT Oy. I have additionally received a personal grant from the Fortum Foundation, and direct funding in the form of work hours from VTT. All funding of this work is gratefully acknowledged.

I am indebted to my thesis advisor Assoc. Prof. Keijo Heljanko who has been extremely active throughout the entire process. I am certain that this dissertation would not yet be finished without his continuous support. Keijo managed to find time for frequently occurring private tutoring sessions, and encouraged me to continue with the work at times when the problems seemed too overwhelming.

I would also like to thank my preliminary examiners for their constructive feedback that helped me improve the contributions of my work. I am honoured to have Prof. Gerald Lüttgen from the University of Bamberg

as my opponent in the public defence of my dissertation.

My workplace has also supported my research work, and has given me plenty of time to finalise this dissertation. I owe thanks to Dr. Jari Hämäläinen for hiring me, and for being enthusiastic about research work related to model checking in the early phases of my research career. I would also like to thank my former team leader Dr. Juhani Hirvonen for encouraging me during the early phases of my career, as well as my current team leader Dr. Juha Kortelainen, Research Professor Dr. Tommi Karhela and Head of Research Area Dr. Riikka Virkkunen for their support during the final phases of the writing process.

I am grateful for the contributions of my co-authors. In addition to Assoc. Prof. Keijo Heljanko who was already mentioned, I worked with many researchers. Prof. Ilkka Niemelä was involved in the writing of Publication I and Publication II. His comments were insightful as always, and the influence of the conversations we had was significant. I am also grateful to my other co-authors Janne Valkonen, Kim Björkman, Tuomas Kuismin and Juho Frits for their efforts and helpful comments. I owe special thanks to Janne Valkonen who has arranged many opportunities for me to focus on my research, and has helped me advance my work in countless other ways as well. I would also like to thank Antti Pakonen, Teemu Tommila, Dr. Jan-Erik Holmberg and Tero Tyrväinen who have provided helpful remarks on many occasions. Numerous representatives of Fortum, TVO and the Finnish Radiation and Nuclear Safety Authority (STUK) and other organisations have also been very supportive towards my work. I want to especially thank Martti Välisuo, Olli Hoikkala, Mauri Viitasalo, Mika Koskela, Minna Tuomainen and Erik Lönnqvist.

Finally, I am very grateful for love and support from my family, friends and relatives. Katri and Akseli, thank you for your patience, and all the healthy distractions you have provided.

Espoo, August 8, 2016,

Jussi Lahtinen

List of Publications

This thesis consists of an overview and of the following publications which are referred to in the text by their Roman numerals.

I Jussi Lahtinen, Janne Valkonen, Kim Björkman, Juho Frits, Ilkka Niemelä and Keijo Heljanko. Model checking of safety critical software in the nuclear engineering domain. *Reliability Engineering & System Safety*, Vol. 105, p.104 – 113, Elsevier, September 2012. (`http://dx.doi.org/10.1016/j.ress.2012.03.021`)

II Jussi Lahtinen, Kim Björkman, Janne Valkonen, Ilkka Niemelä. Emergency diesel generator control system verification by model checking and compositional minimization. In *8th Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS 2012)*, Znojmo, Czech Republic. Antonín Kučera, Thomas A. Henzinger, Jaroslav Nešetřil, Tomás Vojnar and David Antoš (Eds), p. 49 – 60, NOVPRESS, October 2012, ISBN 978-80-87342-15-2.

III Jussi Lahtinen, Tuomas Kuismin and Keijo Heljanko. Verifying large modular systems using iterative abstraction refinement. *Reliability Engineering & System Safety*, Vol. 139, p. 120 – 130, Elsevier, July 2015. (`http://dx.doi.org/10.1016/j.ress.2015.03.012`)

IV Jussi Lahtinen. Verification of fault-tolerant system architectures using model checking. In *1st International Workshop on Development, Verification and Validation of Critical Systems (DEVVARTS)*, Lecture Notes in Computer Science, Vol. 8696, p. 195 – 206, Springer, September 2014. (`http://dx.doi.org/10.1007/978-3-319-10557-4_23`)

V Jussi Lahtinen. Automatic test set generation for function block based systems using model checking. In *9th International Conference on the Quality of Information and Communications Technology (QUATIC 2014)*, Guimarães, Portugal, p. 216 – 225, IEEE, September 2014. (http://dx.doi.org/10.1109/QUATIC.2014.15)

Author's Contribution

Publication I: "Model checking of safety critical software in the nuclear engineering domain"

The author of this dissertation is the main author of Publication I. The author has developed the model-checking methodology described in the paper together with Valkonen, Björkman, Frits, Niemelä, and Heljanko. The case study systems have been modelled by Björkman and Frits. The author's involvement is also in providing the discussion and conclusions of the paper.

Publication II: "Emergency diesel generator control system verification by model checking and compositional minimization"

The author of this dissertation is the main author of Publication II. The idea of using this approach was from Niemelä and the author. The author was the main developer of the compositional minimization technique described in this paper together with Niemelä, and the development was supported by Björkman and Valkonen. The author has modelled the emergency diesel generator control system used in Publication II as a case study, and applied the developed technique in practice.

Publication III: "Verifying large modular systems using iterative abstraction refinement"

The author of this dissertation is mainly responsible for writing Publication III. The author has created the fictional example system used in the work, and modelled both of the analysed case study systems. The

concrete iterative abstraction refinement technique developed in Publication III was created together with Kuismin and Heljanko. The software described in the work was developed together with Kuismin.

Publication IV: "Verification of fault-tolerant system architectures using model checking"

The author of this dissertation is solely responsible for Publication IV.

Publication V: "Automatic test set generation for function block based systems using model checking"

The author of this dissertation is solely responsible for Publication V.

List of author's other related publications

The author of this dissertation has also contributed to the following publications that are closely related to nuclear domain system verification even though they are not part of the dissertation.

Kim Björkman, Juho Frits, Janne Valkonen, Jussi Lahtinen, Keijo Heljanko, Ilkka Niemelä, and Jari J Hämäläinen. Verification of safety logic designs by model checking. In *Sixth American Nuclear Society International Topical Meeting on Nuclear Plant Instrumentation, Control, and Human-Machine Interface Technologies, (NPIC & HMIT 2009)*, pages 5–9. American Nuclear Society (ANS), 2009.

Antti Pakonen, Teemu Mätäsniemi, Jussi Lahtinen, and Tommi Karhela. A toolset for model checking of PLC software. In *IEEE 18th Conference on Emerging Technologies & Factory Automation (ETFA)*, pages 1–6, September 2013.

Jussi Lahtinen, Janne Valkonen, Kim Björkman, Juho Frits, and Ilkka Niemelä. Model checking methodology for supporting safety critical software development and verification. In *European Safety and Reliability Conference, ESREL2010*, pages 2056–2063, September 2010.

Kim Björkman, Jussi Lahtinen, Tero Tyrväinen, and Jan-Erik Holmberg. Coupling model checking and PRA for safety analysis of digital I&C systems. In *The International Topical Meeting on Probabilistic Safety Assessment and Analysis (PSA 2015)*, pages 384–392. American Nuclear Society (ANS), 2015.

Antti Pakonen, Jussi Lahtinen, Veli-Pekka Kuutti, and Tommi Karhela. Integrating model checking with safety-critical I&C software design. In *7th International International Topical Meeting on Nuclear Plant Instrumentation, Control, and Human-Machine Interface Technologies, (NPIC & HMIT 2010)*, pages 1729–1740. American Nuclear Society (ANS), 2010.

Jussi Lahtinen, Mika Johansson, Jukka Ranta, Hannu Harju, and Risto Nevalainen. Comparison between IEC 60880 and IEC 61508 for certification purposes in the nuclear domain. In Erwin Schoitsch, editor, *Computer Safety, Reliability, and Security*, volume 6351 of *Lecture Notes in Computer Science*, pages 55–67. Springer Berlin Heidelberg, 2010.

Hannu Harju, Jussi Lahtinen, Jukka Ranta, Risto Nevalainen, and Mika Johansson. Software safety standards for the basis of certification in the nuclear domain. In *Quality of Information and Communications Technology (QUATIC), 2010 Seventh International Conference on the*, pages 54–62, Sept 2010.

1. Introduction

1.1 Background and research environment

This dissertation addresses the problem of verifying the logical correctness of nuclear power plant safety automation systems using a formal method called model checking.

Nuclear power plants have three main safety functions: (1) to control the reactivity of the reactor; (2) to remove heat from the reactor core; and (3) to confine radioactive material, to shield against radiation and control of planned radioactive releases, and to limit accidental radioactive releases [116].

The main safety functions are achieved partly by automation systems that can, e.g., shut down the reactor when needed or pump water to the reactor in order to increase heat removal. In modern nuclear power plants, these automation systems are mostly computer-based digital systems that read measurements, and decide when a set of actuators (e.g., pumps or valves) should be actuated. Furthermore, in order to achieve high reliability, multiple redundant subsystems as well as several diverse systems operating, e.g., using different physical principles, are typically used. These kinds of design solutions decrease the effect of hardware failures on the operation of the safety functions. The correct operation of these systems is also dependent on the correctness of their software. Design errors in software may occur simultaneously in all redundant subsystems as a common cause failure, leading to erroneously inhibited actuation signals or spurious commands from the safety function in the worst case.

Ensuring the correctness of the systems and their operational logic is of paramount importance, as failures of the systems can be potentially catastrophic. Traditionally used techniques include testing and simu-

lation. These are very useful techniques, but they are limited to covering only a small subset of all possible behaviours of the system, and can thus only detect the presence of errors, not their absence. Safety-critical systems, such as the safety automation systems used in nuclear power plants, however, require absolute assurance of the correctness of the systems. Another more formal approach is deductive verification, which uses axioms and proof rules to prove the correctness of a system. The method is time-consuming and can only be performed by experts with considerable experience. None of these methods alone is capable of exhaustive verification with reasonable effort.

Model checking [65] is a formal method that does analyse the system behaviour exhaustively. A model of the system is first built, similarly as in simulation, to be used in the analysis. However, unlike simulation or testing, interesting system scenarios need not be specified or processed individually. Instead, the requirements of the system are formalised and the correctness of the system model with respect to these formal requirements is then exhaustively verified using efficient algorithms. The scalability of the model-checking technique has constantly increased due to improvements in computer technology, and more efficient algorithms. The first significant algorithmic advancement was the introduction of symbolic Binary Decision Diagram (BDD) -based model checking [147]. Later, techniques based on mapping the model-checking problem into a propositional satisfiability (SAT) problem were developed (see, e.g., [25]), further improving the scalability of the method. Most recently, induction-based algorithms and the invention of the Property-Directed Reachability (PDR) algorithm [39] have yet again improved the performance of the method.

Model checking has been adopted in many safety-critical areas, and areas where a failure can lead to substantial economic loss. These technological advancements have already enabled the verification of realistic-sized safety-critical systems using model checking. For example, in the aviation domain model checking has been used for verifying flight critical software [153], for detecting situations where the systems is in a different mode than that assumed by the operator [182], and for verifying the coordination protocol for an automated air traffic control system [218]. In the space domain, NASA has applied model checking to the verification of a multi-threaded operating system used in the Deep Space-1 spacecraft [100], and developed a model-checking tool for Java programs [101]. Model-checking tools for C/C++ programs exist as well. The explicit-state

model checker DiVinE [14], for instance, supports model checking of multithreaded C/C++ programs. Model checking has also been heavily used for verifying the correctness of microprocessors at Intel [85] and at IBM [19], and to support operating system software development at Microsoft [12, 11]. Another application area of model checking is protocol verification, see, e.g., [106] and [125].

In the nuclear domain, Instrumentation & Control (I&C) system development relies typically on qualified automatic code generators, simulations and testing. Model-checking methods are not widely used in the nuclear context, even though tools such as SCADE (Safety Critical Application Development Environment) [22] offer limited support for formal analysis including model checking. Most similar to this dissertation, is the application of model checking to Korean nuclear power plant automation systems. [211, 214, 123]

The Finnish nuclear industry uses many different approaches to verify the correctness of systems. Typically, model checking is not used by the system designers. Instead, model checking analyses are performed as part of independent verification commissioned by the Finnish Radiation and Nuclear Safety Authority (STUK) or by the power utilities. This arrangement, in which the correctness of an existing system design needs to be independently demonstrated is the starting point of this dissertation.

The practice of applying model checking in the nuclear domain is not yet wide-spread, and the focus of such analysis has been in the verification of small limited case studies. This dissertation intends to find efficient and practical ways of applying model checking in the nuclear context, and to better integrate the method into the system development. This dissertation discusses how model checking can aid system verification on a larger scale, i.e., how larger systems implementing multiple safety functions can be efficiently verified, and how the method can also benefit plant-level fault tolerance analyses. The main benefit of this work is that it enables the exhaustive verification of the correctness of safety-critical system designs, which is unachievable in large-scale systems using testing and simulation. The problem of large-scale system verification by model checking in the context of nuclear power plants has not previously been addressed in the literature.

1.2 Objectives and scope

The general research problem of large-scale system verification is divided into four more specific research questions.

Model checking can become computationally challenging on larger system models. This is due to state explosion, i.e., the number of states in the model grows exponentially as the size of the model increases. Modelling solutions and the use of abstraction techniques can significantly improve on the feasibility of model-checking analysis.

Research Question 1 (RQ1): How can modelling and abstraction techniques be used to enable the model checking of larger nuclear domain automation systems?

Model checking involves a lot of manual work that makes the use of the method expensive and prone to human error. One part of modelling work that often requires human interaction is finding a suitable abstraction level for the model. Especially in large models it can be difficult to find a level of abstraction that is both computationally feasible and accurate enough for actual verification.

Research Question 2 (RQ2): How can a suitable abstraction level of the system model be found automatically?

Nuclear power plants are designed to be tolerant to failures. Traditionally the fault-tolerance of the systems and of the plant is analysed using methods that exclude the detailed operational logic of the safety automation systems. A model-checking approach could provide a more exhaustive plant-level fault-tolerance analysis if hardware failures and their effects were integrated with models of the safety automation systems.

Research Question 3 (RQ3): How can plant-level models be created that cover both the detailed operational logic of multiple automation systems and the hardware failures related to these systems?

The safety system software used in nuclear power plants need to be rigorously tested. The tests performed on the software need to include specification-based tests as well as structure-based tests. The software is often designed using the function block diagram (FBD) language [110], and the FBD programs are automatically translated into code. It is more intuitive to design tests based on the function block diagrams instead of the computer-generated code. However, no well-established methods exist

Table 1.1. The relationships between research questions and the publications

	Pub. I	Pub. II	Pub. III	Pub. IV	Pub. V
RQ1: How can modelling and abstraction techniques be used to enable the model checking of larger nuclear domain automation systems?	X	X		X	
RQ2: How can a suitable abstraction level of the system model be found automatically?			X		
RQ3: How can plant-level models be created that cover both the detailed operational logic of multiple automation systems and the hardware failures related to these systems?				X	
RQ4: Can model checking be used to support structure-based test design of function block diagrams?					X

for structure-based test design based on function block diagrams.

Research Question 4 (RQ4): Can model checking be used to support structure-based test design of function block diagrams?

The relationships between research questions and the publications that address these questions is illustrated in Table 1.1.

The contributions of each paper and the relations between the publications are illustrated in Figure 1.1. A short description of each publication and its contribution follows.

Publication I develops basic methodology for modelling safety automation systems. The methodology presented in this dissertation is the result of collaboration between many researchers. The main methodological contributions of Publication I are the use of a free environment model, and the generic techniques for abstracting the scan cycle of safety systems.

Publication I also discusses the model-checking process and work practices that should be followed to better support the safety demonstration and licensing of these systems. Two case studies are presented in which design errors were found using model checking. Publication I also identifies issues hindering the integration of model checking to system development.

The contribution of Publication II is an abstraction method for large modular systems, enabling simple abstraction of a system. An emergency diesel generator control system is used as a case study. Diesel generators are typically used at nuclear power plants to provide backup power in case of emergencies. The studied control system is made up of several interconnected subsystems. A design error discovered using model checking is presented in the paper.

Publication III defines a technique for determining a suitable abstraction level automatically that is based on the methodology of Publication I and Publication II. The novel contribution of Publication III is a technique based on iterative abstraction refinement, as well as the use of multiple model-checking engines in parallel. The technique is tested by applying it on two case study systems. The results show that in most cases the technique is able to find proofs of correctness more efficiently than traditional model-checking approaches.

The contribution of Publication IV is an extension of the methodology of Publication I to hardware failures allowing larger models to be built, in which also fault-tolerance properties can be analysed. The methodology follows closely the conventions of probabilistic risk assessment (PRA) [16], and serves as a first step for further integration between these two approaches.

The contribution of Publication V is an automatic test set generation technique to support structure-based testing. In this technique, model checking is used to generate the concrete test cases. The resulting test set is efficient in the sense that the number of test cases is small and the tests are concise while having maximum coverage according to a structure-based criterion.

This dissertation focuses on the verification of application software used in nuclear power plants to implement safety functions and safety-related functions. Any other software such as platform software or software related to data handling, transmission, or any software not relevant to safety is out of scope of this dissertation. In addition to application soft-

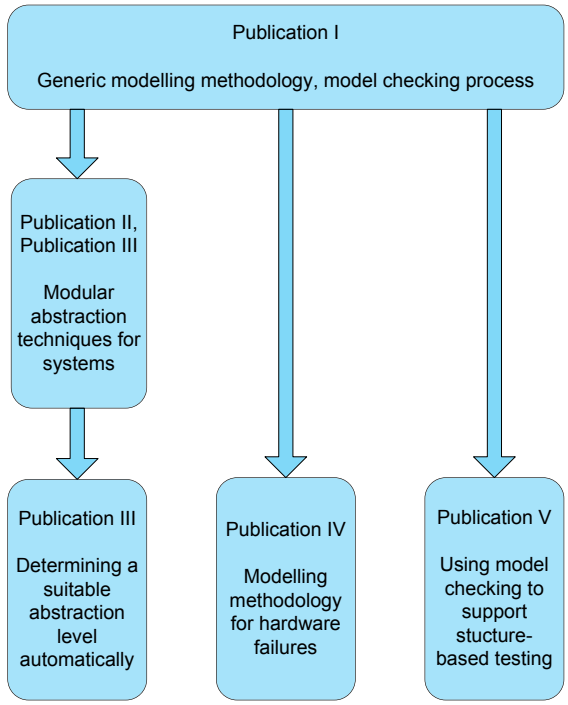


Figure 1.1. The contribution of each publication of this dissertation

ware, hardware components and failures related to safety systems are considered in Publication IV.

The dissertation focuses on verifying the logical correctness of the design of safety systems. Only system designs in the form of FBD programs are considered to be in the scope of this work. It is assumed that the FBD design language follows IEC 61131-3 [110]. Validating the final implementation against the original design of the system is also excluded in this work, and tools performing, e.g., code generation based on FBD programs are assumed to be correct.

Any asynchronous behaviour and data transmission delays are also left out of scope. Throughout this work it is assumed that the correct timing of the systems is separately verified, and that asynchronous anomalies due to, e.g., clock drift are appropriately handled by the automation platform.

The capability of model checking to exhaustively analyse a system is also based on the assumption that the functional requirements of the system used for the analysis are complete and correct. Problems related to the coverage, correctness, and formalisation of requirements are out of scope in this dissertation.

1.3 Research process and dissertation structure

The research of this dissertation includes both experimental and theoretical research. A significant part of the research is the development of a new methodology for: (1) modelling safety system designs; (2) modelling hardware failures; and (3) abstraction methods of modular systems.

The research also involves technique development. First, an automatic, iterative technique is developed for obtaining a suitable abstraction level of the model. Secondly, a technique for test set generation is developed. Both research tasks also include software tool development. Finally, the research work includes several case studies, in which the feasibility of the developed techniques and methodologies is evaluated by modelling both fictitious and real systems.

This dissertation consists of five publications and a comprehensive summary of the work. The summary part is structured as follows. Chapter 1 describes the background of the work, the objectives and scope of the work, and the research process. Chapter 2 introduces the model-checking method and the model-checking algorithms most relevant to this work. Chapter 3 briefly discusses nuclear domain system design and verification. Related work is in Chapter 4. Chapter 5 summarises the main results of Publication I, and presents modelling methodology for FBD programs. Chapter 6 describes the contents of Publication II and Publication III, and presents the iterative abstraction refinement technique for large modular systems. In addition to the technique as described in Publication III, an extension of the technique that covers also liveness properties is presented. Chapter 6 also presents a comparison of the developed technique against another implementation of the Property-Directed Reachability (PDR) algorithm. Chapter 7 discusses the results of Publication IV related to hardware failure modelling and analysis of fault-tolerance, and describes a novel concept-level approach for coupled use of probabilistic risk assessment (PRA) and model checking that was not included in Publication IV. Chapter 8 presents the test set generation technique developed in Publication V. Finally, Chapter 9 concludes the dissertation by summarising the answers to the research questions, and discusses the implications of the work, as well as recommendations for future work.

2. Model checking

Model checking is a formal method used for verifying the correctness of systems. The effectiveness of traditional verification methods such as testing and simulation is dependent on the coverage of the test cases and the coverage of the executed simulation traces. In many complex systems the traditional verification methods cannot provide exhaustive analysis due to the sheer number of test cases or simulation traces required, and thus certain errors may remain hidden. A particular error is found only if a scenario capturing such an error is explicitly specified as part of the test design.

Another traditional technique is deductive verification (see, e.g., [34]), which uses axioms and proof rules to verify the correctness of a system. The technique is also known as interactive theorem proving. This kind of approach requires manual interaction, and can be difficult to use in practice.

Model checking is an automatic method that is also exhaustive. In model checking, test cases need not be explicitly defined. Instead, the functional requirements of the system are formalised, and a software tool ensures that all behaviour related to that property is analysed.

2.1 General model-checking process

Model checking is defined in Publication I as an iterative work process; see Figure 2.1. The process starts with the definition of the scope and boundaries of the analysis. This is done by analysing the overall design of the system and deciding which parts of the system behaviour should be included in the model.

In the model construction step a model of the system is built based on design documentation. The model can, for example, use propositional logic

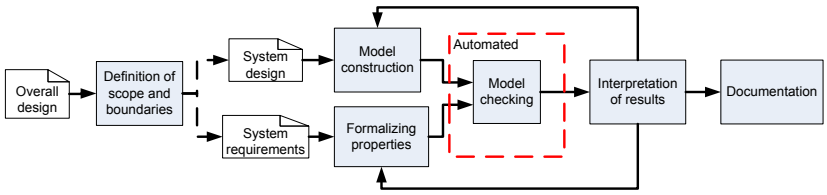


Figure 2.1. Model-checking process

to describe valid transitions, and a state machine model can then be automatically generated based on these constraints. The model can also specify the state machine model explicitly. In practice, these models are quite similar to simulation models. It is important to both define the interface between the model and its environment, and decide the appropriate abstraction level.

The system requirements need to be formalised using some exact notation such as temporal logic [170]. In temporal logic it is possible to express system behaviour in terms of time. Linear Temporal Logic (LTL) and Computation Tree Logic (CTL) [65] are examples of temporal logics. The formalisation of system requirements can be demanding, especially as the requirements are typically not precise enough to be directly translated into temporal logic. Typically they need to be divided into several sub-requirements each of which can be separately formalised. Requirement formalisation goes hand in hand with modelling since the requirements are based on the selected level of abstraction, and the interface between the model and the environment.

When the model and the formalised properties exist, the model-checking tool automatically checks whether a property f is true in a model M at its initial state s_0, formally denoted as $M, s_0 \models f$. If this is true the model-checking tool reports that the formalised property is true. If the property does not hold, the tool generates a concrete counterexample.

The next step is interpretation of the results of the model checker. The counterexamples output by the model checker need to be manually analysed to see whether the counterexample describes an actual error in the system, or if the counterexample is caused by incorrect assumptions in modelling, the level of abstraction chosen in the model, or an error in the model or a specification. If the model or a specification is incorrect, these parts need to be revised and the model-checking phase is executed again. This iterative nature of the model-checking process significantly improves the quality of models because the errors made in the modelling phase are

typically found when the counterexamples are interpreted.

The last model-checking phase is documentation. Items that are typically documented include the functional description of the system, the verified properties both in natural language and in temporal logic, and the results of model checking for each property. The findings (potential errors) need to be precisely documented so that the cause of the erroneous behaviour can be identified.

2.1.1 System modelling

In order to properly analyse the system it must be described in a way that captures the system state and the possible changes in that state through time. In practice, various high-level languages such as the modelling language of NuSMV [51], Promela [106], and Petri nets [157] are used for modelling. In the model-checking literature, however, Kripke structures (see, e.g., [65]) have become the standard representation for formulating verification algorithms in a modelling language independent way. In a Kripke structure, *a state* captures the value of every variable at a particular time instant. *Transitions* between states may exist depicting the possible ways a system can change its state as time progresses. The set of pairs forming the transitions is called the transition relation. A sequence of states that is according to the transition relation is called a *path*.

Definition 2.1. *A Kripke structure is defined as* $M = (S, S_0, R, L)$ *where* S *is the set of states,* $S_0 \subseteq S$ *is the set of initial states,* $R \subseteq S \times S$ *is the transition relation, and* $L : S \to 2^{AP}$ *is the labelling function, where* AP *is the set of atomic propositions.*

The labelling function describes the relations between states and the values of atomic propositions in these states. If a state is labelled with an atomic proposition it means that the proposition is true in that state. Conversely, the absence of that label means that the atomic proposition is false in that state.

A finite path of a Kripke structure M is a finite sequence of states $p = s_0, s_1, s_2, \ldots, s_n$ such that $(s_i, s_{i+1}) \in R$ for all $0 \leq i < n$, and $s_0 \in S_0$. An infinite path of M is a sequence of states $p = s_0, s_1, s_2, \ldots$ such that $(s_i, s_{i+1}) \in R$ for all $i \geq 0$, and $s_0 \in S_0$. Consequently, a finite word w of M is a finite sequence of labels $w = x_0, x_1, x_2, \ldots, x_n$ such that $x_i = L(s_i)$ for all $0 \leq i \leq n$, for a finite path $p = s_0, s_1, s_2, \ldots, s_n$ of M. An infinite word is a sequence of labels $w = x_0, x_1, x_2, \ldots$ such that $x_i = L(s_i)$

for all $i \geq 0$, for an infinite path $p = s_0, s_1, s_2, \ldots$ of M. The temporal logic LTL that is used in this dissertation is concerned with infinite paths. In this chapter, we extend deadlocking states of the Kripke structure by adding self-loops, and this gives the semantics of LTL to finite paths. In the following examples and definitions the terms *path* and *word* will refer to infinite paths and infinite words respectively.

Example 2.1. *A small Kripke structure is shown in Figure 2.2. It has four states: s, t, u, and v. The state s is the only initial state depicted using an incoming edge with no source state. The structure has four transitions: from s to t, from t to u, from u to v, and from v to itself. The labelling of the structure is such that $L(s) = \{P\}, L(t) = \{Q\}$, $L(u) = \{P, Q\}$, and $L(v) = \{Q\}$. A possible path in the example is $p = s, t, u, v, v, v, \ldots$.*

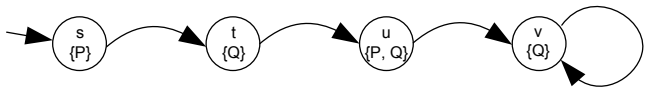


Figure 2.2. An example Kripke structure

2.2 Temporal logic

In order to reason about the behaviour of systems represented as Kripke structures, a formalism for describing sequences of transitions is needed. Temporal logic is such a formalism that extends classical logic in a way that makes reasoning about timelines possible. Temporal logic is needed as classical logic cannot capture properties such as "event A eventually happens."

In model checking, temporal logic is used for formalising functional requirements of the system as properties that the model should have. Traditionally properties are classified as *safety* or *liveness* properties. The safety-liveness classification was originally proposed by Lamport in [135]. Other more refined classifications of temporal properties exist as well, see, e.g., [145], or [53].

Intuitively, safety properties assert that "nothing bad happens," while liveness properties state that "something good" eventually happens in the system. Safety properties are such that if the property can be violated, a finite counterexample exists that demonstrates how the "bad state" is

reached. Counterexamples of liveness properties are always infinite sequences, in which the "good thing" does not happen.

In this dissertation, requirements are formalised using either state invariants or Linear Temporal Logic (LTL). State invariant properties express that a specified condition holds for all reachable states. A state invariant consists of a single formula of propositional logic that must be true in all initial states, and the satisfaction of the formula must be invariant under all states that can be reached from the initial states. Many safety properties are invariant properties. However, all safety properties are not invariants as safety properties may have requirements on finite path fragments instead of reachable states only. It is possible to reduce safety property verification to invariant checking by extending the model with a construct that recognizes these finite path fragments. A translation algorithm from LTL safety properties to finite automata is presented in [136]. The algorithm is based on the work of Kupferman and Vardi [126] and essentially reduces LTL safety checking to invariant checking.

There are also other well-known techniques for mapping safety properties to invariants. One example is the use of a subset of a temporal logic such as past LTL for which there is a construction based on history variables for reducing the property into an invariant. [102]

2.2.1 Linear temporal logic

There are two main branches of temporal logics: branching time and linear logics. Branching time logics see the future as a tree-shaped structure, in which many different futures exist any one of which may be realised. Linear logics, such as Linear Temporal Logic (LTL), on the other hand perceive all possible system behaviours as a set of paths. The descriptions of LTL syntax and semantics below mostly follow the notations in [65].

Given a set of atomic propositions AP, the set of LTL formulas can be inductively defined as follows:

- If $p \in AP$, then p is an LTL formula,
- If Φ is and LTL formula, then $\neg\Phi$, and $\mathbf{X}\ \Phi$ (next state) are LTL formulas,
- If Φ_1 and Φ_2 are LTL formulas, then $\Phi_1 \vee \Phi_2$, and $\Phi_1\ \mathbf{U}\ \Phi_2$ (until) are LTL formulas.

Logical shorthands such as $\wedge$, $\rightarrow$, $\leftrightarrow$, True, and False can be used as well. The temporal operator $\mathbf{X}$ (next state) requires that the property holds at

the next state of the path. The temporal operator **U** (until) is such that a formula Φ_1 **U** Φ_2 holds on a path where (1) Φ_1 is continuously true until Φ_2 becomes true; and (2) Φ_2 must eventually become true at some point. In addition to the temporal operators **X** and **U** the following shorthand temporal operators can also be used:

- Finally: **F** Φ $=$ True **U** Φ, holds when a property eventually becomes true at some future state of the path;
- Globally: **G** Φ $=$ $\neg$ **F** $\neg\Phi$, requires that the property holds at every future state of the path;
- Release: Φ_1 **R** Φ_2 $=$ $\neg(\neg\Phi_1$ **U** $\neg\Phi_2)$, requires that either Φ_2 is always true, or it is continuously true until and including the point where Φ_1 becomes true; and
- Weak until: Φ_1 **W** Φ_2 $=$ $(\Phi_1$ **U** $\Phi_2)$ $\vee$ **G** Φ_1, requires that either Φ_1 is always true, or it is continuously true until Φ_2 becomes true.

Formally, the semantics of LTL formulas can be defined with respect to an infinite word $w \in (2^{AP})^\omega$ where $w = x_0, x_1, x_2, \ldots$. The suffix of w starting at x_i is denoted as w^i. The relation $\models$ is defined inductively as follows:

$w^i \models p$ iff $p \in x_i$ for $p \in$ AP,

$w^i \models \neg\Phi$ iff $w^i \not\models \Phi$,

$w^i \models \Phi_1 \vee \Phi_2$ iff $w^i \models \Phi_1$ or $w^i \models \Phi_2$,

$w^i \models$ **X** Φ iff $w^{i+1} \models \Phi$, and

$w^i \models \Phi_1$ **U** Φ_2 iff $\exists j \geq i$ such that $w^j \models \Phi_2$ and $w^n \models \Phi_1$ for all $i \leq n < j$.

The formal semantics of temporal formulas can be interpreted with respect to paths of a Kripke structure. When a set of paths is considered, the LTL formula has to be true on all paths to be true. Therefore, an LTL formula Φ holds in a Kripke structure M if and only if $w \models \Phi$ for every word w of M. If a formula Φ does not hold for M there is a word $w = x_0, x_1, x_2, \ldots$ such that $w \models \neg\Phi$. Such a word w is called a counterexample to Φ.

2.3 NuSMV model checker

The model-checking tool primarily used throughout this work is NuSMV[1] [57, 84]. NuSMV is a state-of-the-art open-source model-checking tool that can be used for modelling many kinds of systems. The input language of NuSMV is intended for describing finite-state machine models that operate in discrete time. Finite data types such as Boolean variables, enumerative variables, and integers with a limited range are supported.

2.3.1 Modelling language

NuSMV models are collections of variable declarations and assignments defining the valid initial states and transition relations over these variables. Non-deterministic transitions are also supported. In addition, the modelling language uses, e.g., macro definitions, and module hierarchies.

The modules of NuSMV are simply encapsulated collections of declarations. Once a module has been defined, instances of it can be created as many times as necessary. Modules are associated with a list of parameters, and can contain instances of other modules. In a NuSMV model, a single *main* module must exist that has no parameters. This is the main module of the model that is evaluated by the interpreter.

In NuSMV, specifications can be expressed using state invariants or Linear Temporal Logic (LTL). NuSMV also supports specification languages that have not been used in this work. Such languages are, e.g., Computation Tree Logic (CTL) [65] and Property Specification Language (PSL) [115].

An example NuSMV model is in Listing 2.1. The main module of the model declares a single Boolean variable `var1`, and creates an instance `detector1` of another module `EdgeDetector` with the Boolean variable `var1` as a parameter. The main module also has two macro definitions for the outputs of `detector1`: `rising_edge` and `falling_edge`. The module `EdgeDetector` detects rising edges and falling edges of its Boolean parameter `input`. It declares a Boolean variable `previnput` whose initial value is set to false in the `ASSIGN` section of the module. The transition relation regarding the variable is defined using the `next` expression. In this case it is defined so that at the next time step the value of `previnput` is equal to the value of the parameter `input` at the current time point. The result is that after the initial time point `previnput`

[1] NuSMV version 2.5.4 was used. `http://nusmv.fbk.eu/`

always holds the value of `input` at the previous time point.

Finally, in the `DEFINE` section of `EdgeDetector` two macro definitions `rising` and `falling` are described using simple `case` structures. The definition `rising` is true whenever the parameter `input` was false in the previous time point, and is true at the current time point. Similarily, `falling` is true whenever the parameter `input` is currently false, and was is true at the previous time point. An example of a state invariant specification in the format of NuSMV is given on line 10. An equivalent LTL specification is given on line 11.

```
1  MODULE main
2  VAR
3    var1 : boolean;
4    detector1 : EdgeDetector(var1);
5
6  DEFINE
7    rising_edge := detector1.rising;
8    falling_edge := detector1.falling;
9
10   INVARSPEC (! rising_edge);
11   LTLSPEC G (! rising_edge);
12
13 MODULE EdgeDetector(input)
14 VAR
15   previnput : boolean;
16 DEFINE
17   rising :=
18     case
19       ! previnput : input;
20       TRUE : FALSE;
21     esac;
22   falling :=
23     case
24       ! input : previnput;
25       TRUE : FALSE;
26     esac;
27 ASSIGN
28   init(previnput) := FALSE;
29   next(previnput) := input;
```

Listing 2.1. An example NuSMV model

2.4 Employed algorithms

NuSMV implements many different model-checking algorithms. With regard to the algorithms used in this dissertation, the BDD-based symbolic invariant checking (see Section 2.6.1) and BDD-based symbolic LTL model checking (see Section 2.8.1) are both implemented in NuSMV. For a detailed description of NuSMV's implementation of LTL model checking, see, e.g., [178].

NuSMV also employs many SAT-based model-checking algorithms. The implementation of the k-induction algorithm (described in Section 2.7) in NuSMV follows the description of Eén and Sörensson [78].

NuSMV does not have any implementation of the PDR algorithm nor any implementation of liveness-to-safety reductions. The iterative abstraction refinement technique described in Chapter 6 uses both techniques. For PDR model checking, the NuSMV model is translated into the format required by the model-checking tool ABC/ZZ [80] that has an implementation of PDR. The models used in the ABC/ZZ tool are in the AIGER input format. The AIGER [28] format is based on And-Inverter Graphs (AIGs) and it basically corresponds to a low level desription of a Boolean circuit. The liveness-to-safety reduction is performed using standalone software that implements the state-recording translation as described in [186].

The PDR algorithm and the liveness-to-safety reduction are both implemented in the nuXmv [50] tool, which is an extension to NuSMV. However, the license conditions of nuXmv prevent free commercial use. Since the research in this dissertation intends to support verification work typically performed in customer projects, the nuXmv tool was not utilised.

2.5 Symbolic model checking

The first model-checking techniques were simultaneously developed by two different research groups: Clarke and Emerson [63], and Quielle and Sifakis [172]. The developed algorithms were originally *explicit-state* techniques in which the system states are individually represented and enumerated, and the algorithms operated directly on the Kripke structure. These early explicit-state techniques suffered severely from the state explosion problem. State explosion means that the number of states in the model grows exponentially as the size of the model increases. As an ex-

ample, a system with n Boolean variables can have 2^n states in the worst case. Even though some explicit state model checkers such as SPIN [106] and DiVinE [14] have been quite succesful, the explicit-state techniques can become quite infeasible in large models such as the system models studied in this dissertation that involve a lot of free input variables, and have state spaces with a high branching degree. There are also interesting new approaches halfway between symbolic and explicit-state model checking, see, e.g., [156]. These approaches have not been studied in this dissertation, and determining whether the approaches are applicable to this domain is left for future work.

In spite of many different approaches, the state explosion problem has remained a formidable challenge. One of the most successful approaches has been symbolic model checking, in which states are not individually represented, and the algorithms operate on sets of states and sets of transitions instead.

The symbolic encoding of the states and transitions of a Kripke structure is demonstrated here using the system described in Example 2.1. The example Kripke structure is defined as:

$$
\begin{aligned}
&S = \{s, t, u, v\},\\
&S_0 = \{s\},\\
&R = \{(s,t), (t,u), (u,v), (v,v)\}, \text{ and}\\
&L(s) = \{P\}; L(t) = \{Q\}; L(u) = \{P, Q\}; L(v) = \{Q\}.
\end{aligned}
$$

The different states of the system need to be symbolically encoded. Since there are four states in the system, all states can be represented using two bits: x_0 and x_1. The encoding used here maps s to $\neg x_0 \wedge \neg x_1$, t to $\neg x_0 \wedge x_1$, u to $x_0 \wedge \neg x_1$, and v to $x_0 \wedge x_1$.[2] This encoding allows us to represent sets of states using Boolean functions such that each Boolean function evaluates to true exactly for the states it represents. For example, the set of states in which Q is true can be represented by the Boolean formula $x_0 \vee x_1$. Other labels of the system can be represented in a similar manner.

For symbolically representing the transitions of the system, two sets of state variables are needed. The starting state of a transition is encoded as $x = (x_0, x_1)$ and the target state of the transition is encoded using primed copies of the variables: $x' = (x'_0, x'_1)$. The symbolic transition relation RS is then a formula such that $(s, s') \in R$ if and only if $x = (x_0, x_1)$ is the sym-

[2] An alternative way to express this encoding is to use a bit vector of length 2, so that the states s, t, u, v are mapped to vectors $00, 01, 10, 11$ respectively.

bolic representation of s, and $x' = (x'_0, x'_1)$ is the symbolic representation of s', and $RS(x, x')$ evaluates to true.

In the example system, the total symbolic transition relation is: $RS = \{(\neg x_0 \wedge \neg x_1 \wedge \neg x'_0 \wedge x'_1) \vee (\neg x_0 \wedge x_1 \wedge x'_0 \wedge \neg x'_1) \vee (x_0 \wedge \neg x_1 \wedge x'_0 \wedge x'_1) \vee (x_0 \wedge x_1 \wedge x'_0 \wedge x'_1)\}$

For model checkers, variants of Binary Decision Diagrams (BDDs) [43] have traditionally been used for representing the state sets and the transition relation. A BDD is a directed acyclic graph that has two sink nodes labelled 0 and 1 representing the Boolean values 0 ($False$) and 1 ($True$). Each non-sink node is labelled with a Boolean variable v. The non-sink nodes have two child nodes called *low child* and *high child*, and the edges to these nodes are labelled with 0 and 1. An edge from v to its low child represents an assingment of v to 0. Similarly, an edge from v to its high child represents an assignment of v to 1. An Ordered Binary Decision Diagram (OBDD) is a BDD in which the input variables are ordered, and every path from a source node to a sink node follows the variable ordering. A Reduced Ordered Binary Decision Diagram (ROBDD) is a BDD in which all isomorphic subgraphs have been merged, and nodes that have isomorphic child nodes have been removed. ROBDDs are typically used as they are very compact, and have a canonical representation. Many basic operations can be performed very efficiently on ROBDDs. An example ROBDD is illustrated in Figure 2.3. When compared to explicit representation of state spaces, representing sets of states by their characteristic functions symbolically makes it possible to verify systems that have several orders of magnitude more states [44].

In addition to BDD-based symbolic model checking, techniques based on propositional satisfiability (SAT) solving are also often used, see, e.g., [25]. The propositional satisfiability problem [29] can be defined as follows. Let $V = x_1, x_2, \ldots x_n$ be a finite set of Boolean variables. A Boolean variable x_i or its negation $\neg x_i$ is called a literal. A clause is a disjunction of literals, and a SAT instance is a conjunction of clauses. A valuation is a function that assigns each variable in V to a Boolean value. A literal x_i is satisfied if x_i is assigned a Boolean value 1. A literal $\neg x_i$ is satisfied if x_i is assigned a Boolean value 0. A clause is satisfied when at least one of its literals is satisfied. A SAT instance is satisfied when all of its clauses are satisfied. The SAT problem is to decide whether a valuation that satisfies the SAT instance exists, and it is the canonical NP-complete decision problem [29].

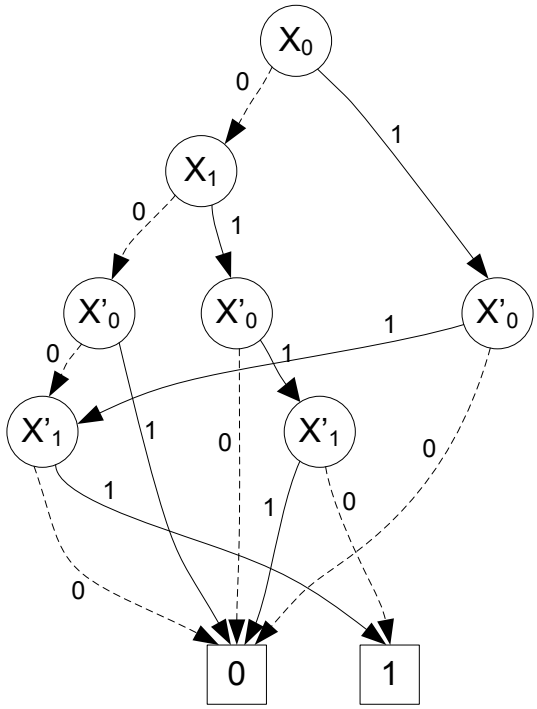


Figure 2.3. A Reduced Ordered Binary Decision Diagram for the transition relation of the example system, $RS = \{(\neg x_0 \wedge \neg x_1 \wedge \neg x'_0 \wedge x'_1) \vee (\neg x_0 \wedge x_1 \wedge x'_0 \wedge \neg x'_1) \vee (x_0 \wedge \neg x_1 \wedge x'_0 \wedge x'_1) \vee (x_0 \wedge x_1 \wedge x'_0 \wedge x'_1)\}$. A variable ordering of x_0, x_1, x'_0, x'_1 is applied.

2.6 Symbolic invariant checking

As was mentioned in Section 2.2, safety property checking can be reduced to the checking of state invariant properties. Therefore, in the context of safety property verification this dissertation only discusses model-checking algorithms that can be used to verify state invariants. These algorithms, together with an appropriate reduction, can be used to check all safety properties. Three invariant checking algorithms are covered. BDD-based invariant checking [74, 57] and the PDR algorithm [39] use an approach based on inductive invariants, while the more general k-induction algorithm [189] builds on Bounded Model Checking (BMC) principles, and extends traditional BMC to also proving properties.

2.6.1 BDD-based invariant checking

The basic method for verifying invariant properties is to prove that a state in which the invariant property is false cannot be reached from the initial

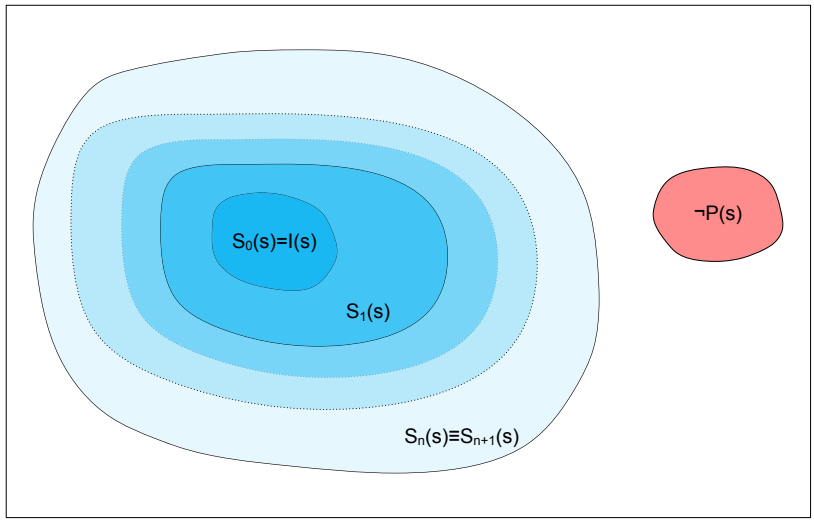


Figure 2.4. The forward method checks whether a bad state $\neg P(s)$ can be reached from the initial states.

state of the model. The basic forward variant of this method, see, e.g., [74], starts from the initial states and iteratively applies the transition relation to the forward direction to create sets of states that can be reached from the initial states. A description of this technique follows.

Assume that $I(s) = S_0(s)$ is a formula encoding the initial states of a system, $R(s, s')$ is a formula encoding the transition relation of the system, and $P(s)$ is a formula encoding the states that satisfy the invariant property, where s and s' are sets of states.

The forward method starts with the set of initial states $I(s)$, and checks whether the formula $I(s) \wedge \neg P(s)$ is satisfiable. If the formula is not satisfiable, the algorithm calculates a new set of states by forward traversal of the transition relation: $S_1(s) := I(s) \vee \exists s' : I(s') \wedge R(s', s)$. For arbitrary n the formula $S_n(s)$ can then be recursively calculated from $S_{n+1}(s) := S_n(s) \vee \{s' | \exists s'' : S_n(s'') \wedge R(s'', s')\}$, for $n \geq 0$.

At each recursive step the algorithm checks whether the formula $S_n(s) \wedge \neg P(s)$ is satisfied. If it is, then the satisfying truth assignment represents a counterexample to the invariant property. If the formula remains unsatisfied, the algorithm increases n until a fix-point is reached, i.e., $S_{n+1} \equiv S_n(s)$. This fix-point corresponds to the reachable state space of the system. If the formula $S_n(s) \wedge \neg P(s)$ is true in this fix-point state, the invariant property is true in the system.

In the alternative backward calculating method (see, e.g., [148]) the bad states are the starting point, and reachable states are calculated in the

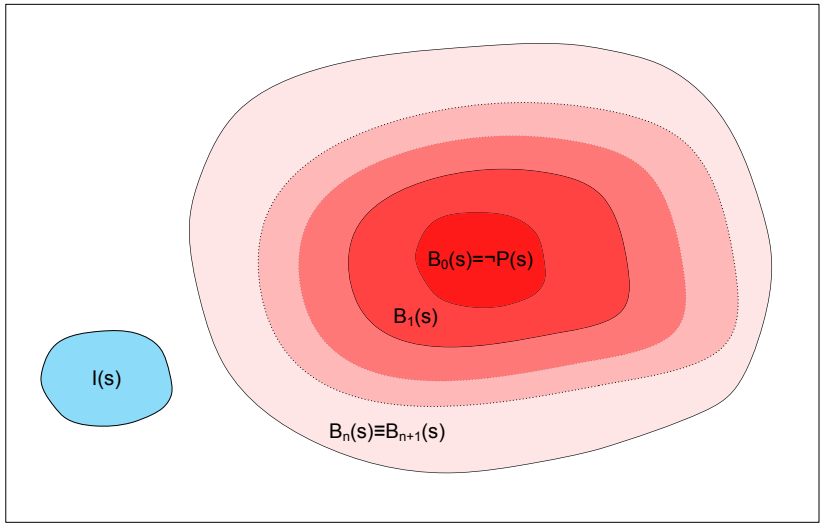


Figure 2.5. The backward method checks starts from a bad state $\neg P(s)$ and checks whether an initial state $I(s)$ of the system can be reached by backward traversal of the transition relation.

backwards direction. The invariant property is unsatisfied if an initial state can be reached, and satisfied if a fix-point state is reached that does not include initial states. In the backward method the bad state $\neg P(s)$ is denoted as B_0. $B_{n+1}(s)$ is then the set of states from which a bad state can be reached in $n+1$ steps, calculated recursively from $B_{n+1}(s) := B_n(s) \vee \{s' | \exists s'' : B_n(s'') \wedge R(s', s'')\}$, for $n \geq 0$. On each iteration the algorithm checks whether an initial state was reached, i.e., whether $I(s) \wedge B_n(s')$ is satisfied. If the formula $I(s) \wedge B_n(s')$ is satisfied, the satisfying truth assignment again represents a counterexample to the invariant property.

Finally, a combination of forward and backward invariant checking is also possible. In this third variant the algorithm calculates both predicates $S_n(s)$ and $B_n(s)$ simultaneously with increasing values of n. On each step the algorithm checks whether a state exists that is in both $S_n(s)$ and $B_n(s)$. If such a state s' exists, it means that it can be reached from the initial state in n steps, and that a bad state can be reached from s' in n steps. The satisfying truth assignment of $S_n(s) \wedge B_n(s)$ represents a counterexample to the invariant property P. The recursive calculation is again terminated when such a state cannot be found, and either of the predicates $S_n(s)$ and $B_n(s)$ reach a fix-point.

2.6.2 Property directed reachability

Bradley suggested an entirely new model-checking approach originally titled IC3 (Incremental Construction of Inductive Clauses for Indubitable Correctness) [39]. Later in [77] the technique was improved upon, and renamed as Property Directed Reachability (PDR). PDR is a SAT-based technique that avoids the unrolling of the transition relation, and instead attempts to reach a proof by solving several smaller SAT-problems. The technique has been found to be very efficient in practice. For example, Brayton's initial implementation of the algorithm won third place in the Hardware Model Checking Competition HWMCC'10 (2010) [26]. In later competitions, various PDR implementations have been incorporated also in other submissions. The description of the algorithm here follows mostly the notations of the original approach [39], while the improvements of [77] are also discussed. In the description below, the transition relation over current and next-state variables of the model is denoted $T(s, s')$. The set of initial states is denoted $I(s)$, and the set of property states is denoted $P(s)$. For convenience, the dependence of T, I, and P on the state variables s and s' is omitted below. For a more detailed explanation of inductive generalisation of states, see [77]. For a proof of the termination of the original algorithm, see [39].

PDR is an algorithm for verifying safety properties. In particular, the intention of the PDR algorithm is to create an inductive invariant that proves the original state invariant property P under examination. This invariant is a Boolean formula called $Proof$ that has the following properties:

- Proof property 1: $Proof$ is satisfied in the initial states of the system $I(s)$;
- Proof property 2: $Proof$ is inductive, i.e., if a state s satisfies the invariant, all successors of s also satisfy the invariant, $Proof \wedge T \implies Proof'$; and
- Proof property 3: $Proof$ is such that it implies the examined property P, $Proof \implies P$.

The algorithm operates on sets of clauses F_i, that over-approximate the set of states reachable from the initial states of the system in i transitions. The sets of clauses $F_0, \ldots F_k$, also called *frames*, are such that the

following four properties are always satisfied:

- Frame property 1: $F_0 = I$.
- Frame property 2: $F_i \implies F_{i+1}$, for $0 \leq i < k$.
- Frame property 3: Every state that can be reached in one transition from a state satisfying F_i satisfies F_{i+1}, i.e., $F_i \wedge T \implies F_{i+1}$, for $0 \leq i < k$.
- Frame property 4: The invariant P is satisfied in all sets F_i, $F_i \implies P$, for $0 \leq i \leq k$.

The algorithm works iteratively. Initially it starts with just one set of clauses $F_0 = I$, and first checks whether the negation of the property P is satisfied in it. If it is, the initial state of the system is a bad state. Otherwise, it enters the main loop of the algorithm.

On each execution of the main loop the algorithm checks whether a bad state can be reached in one transition from the highest frame by solving $F_k \wedge T \wedge \neg P$, using a SAT solver. If the formula is satisfiable the SAT-solver returns a predecessor of a bad state m that satisfies it. The state m is then generalised to a *cube*[3] s.

Next the algorithm attempts to remove s from F_k by trying to find out if the cube is blocked by a previous frame by checking the satisfiability of $F_{k-1} \wedge T \wedge s'$, where s' denotes the next-state encoding of s. If the formula is satisfiable, a satisfying cube t can be extracted, and the algorithm tries to similarly block this new cube in the previous frame F_{k-2}. Eventually a cube will be blocked by some frame, or the initial frame F_0 is reached. If the initial frame (initial state) cannot block the previous cube, the algorithm has found a counterexample to the property P. If, however, at some point a cube can be blocked in some frame F_i, the algorithm adds a negation of that cube to frames F_j, where $j \leq i$, so that frame property 2 remains satisfied.

Once the formula $F_k \wedge T \wedge \neg P$ becomes unsatisfiable in the main loop, the algorithm adds a new frame $F_{k+1} = P$, and the propagates the formulas in the frames to their successors whenever this can be done without violating the frame properties. For each clause $c \in F_i$ the algorithm checks whether $F_i \wedge T \wedge \neg c$ is satisfiable. If the formula is unsatisfiable, c is added to frame

[3] A cube is defined as a conjunction of literals. A literal is defined as a variable or its negation. The cube s is such that it describes a set of states, and it is created by removing irrelevant literals from the bad state m. In PDR, this reduction is done by ternary simulation.

F_{i+1}.

The algorithm terminates with a proof when two consecutive frames become identical, i.e., $F_i \equiv F_{i+1}$, for some $0 \leq i < k$. Since all of the Frame properties are true, we can see that F_i satisfies the properties of a proof. Proof property 1 is implied by Frame properties 1 and 2; Proof property 2 is implied by the fact that $F_i = F_{i+1}$; and Proof property 3 is implied by Frame property 3.

One of the particular benefits of the PDR algorithm is that the proof given by the algorithm is an actual Boolean formula. The $Proof$ formula serves as a certificate of the correctness of the model-checking process since the inductive invariance of the $Proof$ formula can be independently checked using another tool.

2.7 k-induction

One alternative to the BDD-based symbolic model-checking technique is called Bounded Model Checking (BMC) [25]. In BMC, the general model-checking problem is not directly addressed. Instead, a more restricted bounded problem is analysed instead. The basic idea of BMC is to ask whether a counterexample of length k can be found. This problem is then translated into a propositional satisfiability (SAT) problem that can then be solved using efficient SAT solvers.

Let M be a Kripke structure of a system, and P be an invariant that is being verified against the system. A path of M is encoded by unrolling its transition relation k times. This means that $k + 1$ copies of the state variables are created: one copy for each time point. The transition relation T always holds for state variables of two consecutive time points. The initialised finite paths of M of length k are then defined as:

$$[\![M, k]\!] := I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1})$$

A formula that encodes the initialised finite paths of M of length k that lead to a state in which the invariant P is false can be defined as:

$$[\![M, k, \neg P]\!] := I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \neg P(s_k)$$

In the traditional BMC approach, the value of the bound k is initially 0. If a counterexample to the model-checking problem with a bound cannot be found, the value of k is increased. If the examined property is true, the

value of k is in theory increased until a *completeness threshold*, i.e., a separately computable bound that guarantees that the property holds over all infinite paths of the model, is met. In practice, the BMC method is not complete, since the completeness thresholds are typically very large on non-trivial models.

The technique called k-induction [189, 78] combines the BMC approach with mathematical induction in order to verify invariant properties. In k-induction, the transition relation T is unrolled as well, and the counterexamples are found using a formula similar to the BMC approach. In addition to this, k-induction uses a second SAT formula for finding a proof that no counterexamples exist. The description of the k-induction technique follows the notation of the paper by Eén and Sörensson [78].

The k-induction approach starts with a k value of 0 and increases k whenever no counterexample has been found and the property has not been proved. When formulas generated for high values of k are used, the k-induction algorithm always assumes that the reasoning on lower k values has been inconclusive.

The main idea of k-induction is to create a k-step inductive proof. The base case of the induction proof states that a bad state cannot be reached from an initial state in k steps. The induction step of the proof then states that if P has been true for k consecutive time points, it will always be true at time point $k + 1$. If both the base case and the induction step are true for some k, it is possible to inductively deduce that P is true in the system.

Finally, termination and completeness of the k-induction technique can be guaranteed by restricting the analysis to unique paths in the induction step. Unique paths are such that all states on the path are unique, i.e., there are no loops in the path. In finite state systems, there necessarily exists an upper bound for the length of such a path, ensuring the termination of the approach.

The base case, the induction step, and the formula restricting the induction step to non-looping paths are defined below. The formulas are such that their unsatisfiability is used when proving the correctness of P.

$$\text{Base}_k := I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \neg P(s_k)$$

$$\text{InductionStep}_k := \bigwedge_{i=0}^{k} T(s_i, s_{i+1}) \wedge \bigwedge_{i=0}^{k} P(s_i) \wedge \neg P(s_{k+1})$$

$$\text{Unique}_k := \bigwedge_{i=0}^{k-1} \bigwedge_{j=i+1}^{k} (s_i \neq s_j)$$

The k-induction method with the checks for non-looping paths is a complete method for verifying state invariants. Similar complete methods for more descriptive temporal logics exist as well. The technique described in [27] combines k-induction with a Büchi automata-based BMC encoding for LTL with past operators and a reduction from liveness properties to safety properties, resulting in a complete method.

2.8 Model-checking techniques for liveness properties

LTL can be used to express liveness properties. In this section, two different approaches for checking liveness properties expressed using LTL are briefly described. BDD-based symbolic LTL model checking is described in Section 2.8.1, and an approach based on reducing liveness checking into safety checking is described in Section 2.8.2.

2.8.1 BDD-based symbolic LTL model checking

LTL model checking is based on the automata-theoretic approach [202], in which the negated LTL specification $\neg\phi$ is transformed into a generalised Büchi automaton denoted $A_{\neg\phi}$, and composed with the system model M forming $A_{M,\neg\phi}$. In this approach the model-checking problem reduces to checking for emptiness of the language of the composed system. In what follows, this well-known symbolic LTL model-checking technique is explained in detail following the notations used by Clarke, Grumberg, and Peled in [65] as well as the description of the technique by Rozier in [178].

Definition 2.1. *A Büchi automaton is a tuple $\langle \Sigma, S, S_0, \Delta, F \rangle$ where: Σ is a finite alphabet, S is a finite set of states, $S_0 \subseteq S$ is the set of initial states, $\Delta \subseteq S \times \Sigma \times S$ is the transition relation, and $F \subseteq S$ is the set of accepting states.*

A run r of a Büchi automaton over an infinite word $a_0, a_1, a_2, \cdots \in \Sigma^\omega$ is a sequence of states $s_0, s_1, s_2, \cdots \in S$ such that $s_0 \in S_0$, and $(s_i, a_i, s_{i+1}) \in \Delta$ for all $i \geq 0$. A run r is accepting if it visits a state in F infinitely often. The set of infinite words accepted by an automaton A is the language of $L(A) \subseteq \Sigma^\omega$. A language is empty when $L(A) = \emptyset$.

A generalised Büchi Automaton (GBA) can have multiple acceptance

sets $F_1, F_2, \ldots, F_n \in F$ and an infinite run is accepted by the automaton if it visits a state in each set infinitely often.

For every LTL formula ϕ, it is possible to construct a non-deterministic generalised Büchi automaton A_ϕ (also called tableau) such that the language accepted by A, $L_\omega(A_\phi)$ corresponds exactly to the runs described by ϕ, see, e.g., [140] for a proof.

The process of constructing the Büchi automaton $A_\phi = (\Sigma, S_\phi, S^0_\phi, \Delta_\phi, F_\phi)$ is as follows. Assume that the logical $\wedge$ operator has been eliminated by DeMorgan's law ($a \wedge b$ is equivalent to $\neg(\neg a \vee \neg b)$), and that the temporal operators **G**, **F**, **R**, and **W** have been eliminated from the formula (using the equivalences described in Section 2.2.1), and let AP be the set of atomic propositions in ϕ. The alphabet Σ of the Büchi automaton is 2^{AP}. In order to construct the set of states of the Büchi automaton, the set of elementary formulas $el(\phi)$ is first recursively defined.

- $el(p) = \{p\},$ if $p \in AP$
- $el(\neg\psi) = el(\psi)$
- $el(\psi_1 \vee \psi_2) = el(\psi_1) \cup el(\psi_2)$
- $el(\mathbf{X}\ \psi) = \{\mathbf{X}\ \psi\} \cup el(\psi)$
- $el(\psi_1\ \mathbf{U}\ \psi_2) = \{\ \mathbf{X}\ (\psi_1\ \mathbf{U}\ \psi_2)\} \cup el(\psi_1) \cup el(\psi_2)$

Each subset of $el(\phi)$ corresponds to a state $s \in S_\phi$ of the Büchi automaton A_ϕ. Each of these states is labelled with the elementary formulas it corresponds to. The labelling of state s is denoted $l(s)$. In addition to these states, a special state s^0_ϕ is also added to S_ϕ. The state s^0_ϕ is the initial state of the Büchi automaton, $S^0_\phi = s^0_\phi$.

In order to construct the transitions between the states, a function sat is first defined that associates each subformula of ϕ with the set of states that satisfies that subformula.

- $sat(\psi) = \{s \mid \psi \in s\}$ where $\psi \in el(\phi)$
- $sat(\neg\psi) = \{s \mid s \notin sat(\psi)\}$
- $sat(\psi_1 \vee \psi_2) = sat(\psi_1) \cup sat(\psi_2)$
- $sat(\psi_1\ \mathbf{U}\ \psi_2) = sat(\psi_2) \cup (sat(\psi_1) \cap sat(\mathbf{X}\ (\psi_1\ \mathbf{U}\ \psi_2)))$

The transition relation of the Büchi automaton must be such that each elementary formula labelled in a state is true in that state. In particular, if an elementary formula of the form $\mathbf{X}\ \psi$ is labelled in a state s, then ψ

should be satisfied by all successors of s. Also, if $\neg \mathbf{X}\, \psi$ is true in s, then none of the successors of s should satisfy ψ. Formally this property of the transition relation can be defined so that:

$$\Delta_x = \{(s, a, s') \in S_\phi \times AP \times S_\phi \mid \forall\, \mathbf{X}\, \psi \in el(\phi) : \big(s \in sat(\mathbf{X}\, \psi) \Leftrightarrow s' \in sat(\psi)\big) \text{ and } a = l(s') \cap AP\}$$

In addition to this, transitions are added from the special initial state s^0_ϕ to all states s' where $sat(\phi)$ is satisfied. This can be defined formally as follows:

$$\Delta_i = \{(s^0_\phi, a, s') \in S^0_\phi \times AP \times \in S_\phi \mid s' \in sat(\phi) \text{ and } a = l(s') \cap AP\}$$

The transition relation of the Büchi automaton A_ϕ is then:

$$\Delta_\phi = \Delta_x \cup \Delta_i$$

Finally, the acceptance sets of the Büchi automaton are constructed so that for each subformula of ϕ of the form $\psi_1\, \mathbf{U}\, \psi_2$, a new acceptance set $F_{\psi_1\, \mathbf{U}\, \psi_2} \in F_\phi$ is created containing all states labelled ψ_2, and all states labelled $\neg\psi_1\, \mathbf{U}\, \psi_2$.

If such a Büchi automaton is created for the negated specification $\neg\phi$, the resulting automaton $A_{\neg\phi}$ characterises all possible runs that violate the specification ϕ. When this automaton is composed with the system model M, the resulting automaton $A_{M,\neg\phi}$ characterises all possible runs of M that also violate ϕ.

The system model M is given as a Kripke structure $M = (S, S_0, R, L)$ over a set of atomic propositions AP must first be interpreted as a Büchi automaton. Again, a special initial state s^0_M must be added to the Büchi automaton. $A_M = (\Sigma, S_M, S^0_M, \Delta_M, F_M)$ is the corresponding Büchi automaton, where:

- $\Sigma = 2^{AP}$,
- $S_M = S \cup \{s^0_M\}$,
- $S^0_M = \{s^0_M\}$
- For all $s, s' \in S_M, a \in \Sigma : (s, a, s') \in \Delta_M$ if and only if $L(s') = a$ and $(((s, s') \in R)$ or $(s = s^0_M$ and $s' = s_0))$; and
- $F_M = S_M$.

Let $A_{\neg\phi} = (\Sigma, S_\phi, S^0_\phi, \Delta_\phi, F_\phi)$ be the automaton that corresponds to the negated LTL formula, and $A_M = (\Sigma, S_M, S^0_M, \Delta_M, F_M)$ be the automaton corresponding to the system model.

In the special case where all of the states of A_M are accepting ($F_M = S_M$) the automaton $A_{M,\neg\phi} = (\Sigma, S, S^0, \Delta, F)$ accepting the intersection of the languages $L(A_M) \cap L(A_{\neg\phi})$ can be defined by a simpler production construction method[4] [65]:

- $S = S_M \times S_\phi$,
- $S^0 = S^0_M \times S^0_\phi$,
- For all $s, s' \in S_M, t, t' \in S_\phi, a \in \Sigma : ((s,t), a, (s',t')) \in \Delta$ if and only if $(s, a, s') \in \Delta_M$ and $(t, a, t') \in \Delta_\phi$, and
- $F = F_M \times F_\phi$

If an accepting run can be found in $A_{M,\neg\phi}$ it is a counterexample of ϕ in M. If no such run can be found, i.e., $L(A_{M,\neg\phi}) = \emptyset$, it is concluded that ϕ holds in M, i.e., $M \models \phi$.

The product automaton $A_{M,\neg\phi}$ can also be viewed as a directed graph $G_{M,\neg\phi} = (V, E)$ where V is the finite set of states corresponding to the states of the automaton such that $V = S$, and $E \subseteq V \times V$ is the set of edges such that $(s, s') \in E$ whenever $(s, a, s') \in \Delta$ for any two states s and s', and for any alphabet $a \in \Sigma$. The paths of the graph then correspond to the computations of the automaton.

In order to perform model checking, strongly connected components (SCC) of the graph G can be exploited. A strongly-connected component is defined as a maximal subgraph such that any two states are mutually reachable. A single state that is not connected to itself is a trivial SCC.

Given a set of accepting sets F, a *fair* SCC is defined to be a such a nontrivial SCC that intersects each accepting set in F.

Every accepting run of a generalised Büchi automaton can be represented as a lasso-shaped path from an initial state to an accepting cycle. Thus, using the above defined constructs, the problem of finding whether the generalised Büchi automaton $A_{M,\neg\phi}$ has an accepting run can be reduced to finding a path to a fair strongly-connected component in the graph $G_{M,\neg\phi}$.

In symbolic LTL model checking, the model A_M, the LTL formula $A_{\neg\phi}$ and the product automaton $A_{M,\neg\phi}$ are encoded using reduced ordered

[4]Note that this product construction is not correct for arbitrary automata.

Algorithm 2.1 Symbolic algorithm detecting fair cycles [79]

```
1:  procedure CHECKFAIRCYCLES(F)
2:      Z' := ⊤
3:      repeat
4:          Z := Z'
5:          for each F_i ∈ F do
6:              Y := CheckEU(Z, F_i ∧ Z)
7:              Z' := Z' ∧ PreImage(Y)
8:          end for
9:      until Z' ⇔ Z
10:     return Z
11: end procedure
12: procedure CHECKEU(φ_1, φ_2)
13:     X := φ_2
14:     repeat
15:         X' := X
16:         X := X ∨ (φ_1 ∧ PreImage(X))
17:     until X' ⇔ X
18:     return X
19: end procedure
```

BDDs (see Section 2.5). Accepting runs are detected based on a symbolic algorithm that operates on sets of states, and sets of transitions. The first such algorithm was by Emerson and Lei [79]. The Emerson-Lei algorithm is based on μ-calculus (see, e.g., [147]) and the computation of fix-points by repeated pre-image computations.

A symbolic algorithm based on the original Emerson-Lei algorithm is outlined as pseudo-code in Algorithm 2.1. The algorithm operates on sets of states, i.e., the variables X, Y, Z, X', Z' are reduced ordered BDDs representing a set of states. The algorithm has two procedures. *CheckFairCycles* has as input the set of accepting states F, and it returns either an empty set or a state set in which each state can be reached from another state within the set, and all accepting conditions are met. It does this by greatest fix-point calculation. The value of Z is initially set to $\top$ (i.e., all of the states in the automaton) and the value of Z is iteratively updated until it converges to a fix-point.

The procedure *CheckEU* is used to compute a set of states from which a path exists such that the first parameter ϕ_1 is true until the second pa-

rameter ϕ_2 becomes true. It does this by calculating a least fix-point. $CheckFairCycles$ makes repeated calls to $CheckEU$ in order to compute states from which each of the accepting states can be reached. It then excludes the states that do not have any successors, as such states can not form a fair cycle. Both procedures of Algorithm 2.1 utilise a predicate $PreImage(X)$ that calculates the states that are backwards reachable from the state set X. Formally $PreImage(X)$ can be defined as:

$$PreImage(X) = \{s \in S | (s, a, s') \in \Delta \text{ for some } s' \in X\}.$$

If a fair cycle can be found using Algorithm 2.1, it remains to be shown that the corresponding cycle in M can be reached from the initial states. The detailed counterexample trace also has to be created. A path to the fair SCC can be calculated by starting from the initial states of M and calculating states reachable from them by repeatedly applying the transition relation until a state in the fair SCC is found or a fix-point state (all reachable states found) is reached. A similar reachability analysis can be used to construct a looping trace in the fair SCC such that all acceptance sets are visited. The counterexample trace is the combination of the path from the initial state to the SCC, and the loop inside that SCC visiting all of the acceptance sets.

Due to a doubly-nested fix-point operator, the Emerson-Lei algorithm operates in quadratic time. A theoretically better $\mathcal{O}(n \log n)$ algorithm is described in [35] but this alternative algorithm has been found to often perform worse in practice; see [174].

2.8.2 Liveness-to-safety reductions

In this dissertation (Publication III in particular) a liveness-to-safety reduction is employed as one way to check liveness properties. In Publication III, the state-recording translation method is used for this purpose.

The *state-recording translation* by Schuppan and Biere [24, 186] reduces liveness checking to safety checking in a monolithic way, i.e., so that only a single safety-checking query is needed to verify a liveness property. The state-recording translation can be applied for general LTL formulas. The product automaton $A_{M,\neg\phi}$ is built, similarly as was described in Section 2.8.1, and each acceptance set can be interpreted as a fairness constraint.

A counterexample to a liveness property is an infinite path where some-

thing good never happens. In a finite-state system this means that the counterexample is lasso-shaped, consisting of a finite prefix and an infinitely repeating loop.

The state-recording translation works by trying to guess the starting point of that loop. In practice, this translation extends the model with a separate observing part, and uses an oracle variable $save$ to determine the starting point of the loop. The translation saves copies of the state variable values to a second pair of state variables at this time point. Another variable $saved$ is used to indicate that the loop starting point has already been guessed in the past. A condition $looped$ is also created that is true when the current state equals the previously saved state, indicating that a lasso-shaped path has been found. The translations monitors the fulfilment of the acceptance sets of the liveness property using flag variables. It creates an additional state variable $fair_i$ for each acceptance set F_i that observes whether a state in the acceptance set is visited within the loop.

Using the auxiliary constructs described above, the liveness property can then be translated into an invariant property stating that no lasso-shaped path exists (where at the last state the state variable values are equal to the saved variable values) such that every acceptance set related to the liveness property is satisfied within the loop. Finally, the liveness property is true when no such paths exist in the model. If k is the number of acceptance sets, the invariant in the translated model corresponding to the original liveness property is of the form:

$$\neg \left(looped \bigwedge_{i=0}^{k} fair_i \right)$$

In addition to the state-recording translation, other liveness-to-safety reduction techniques exist as well. The k-liveness [60] and the acceptance-counting translation [90] describe essentially the same technique. This technique is based on the observation that an LTL property holds when there exists an integer N such that no run of $A_{M,\neg\phi}$ visits an accepting state N or more times. This technique tracks visits to accepting states using counter variables. If an upper bound for N is found, it deduces that the LTL property must hold.

Another approach by Bradley et al. [40] looks for fair strongly-connected components in the composition of the system model and the negated property $A_{M,\neg\phi}$ by making a series of separate reachability queries that are performed by a model-checking tool. In their approach, the queries are done using the PDR algorithm.

3. Nuclear instrumentation and control system development and verification

This chapter focuses on the development and verification of digital Instrumentation & Control (I&C) systems used in nuclear power plants. Section 3.1 covers the most essential design principles related to nuclear domain I&C systems, and Section 3.2 briefly discusses digital I&C system development related matters.

3.1 Nuclear power plant I&C systems

I&C systems enable and ensure the safe operation of nuclear power plants. In their most simple form, I&C systems measure the physical parameters of the plant, provide the HMI (Human-Machine Interaction) interface to the control room, and actuate devices such as pumps and valves when necessary. These control actions can either be fully automatic, or based on manual operator commands.

Nuclear power plants have strict requirements for safety and reliability. To meet these requirements, the *defence in depth* principle [109] is applied. Defence in depth is essentially the use of multiple successive defensive barriers to prevent the release of radioactive material into the environment. When successfully used, this design principle leads to systems in which no single human or mechanical failure can lead to a hazardous scenario. In I&C system design the defence in depth principle is achieved with a hierarchy of systems, and the use of *redundancy* and *diversity*.

For example, it is common to implement three levels of I&C systems: control systems, preventive protection systems, and protection systems. The intention of this arrangement is that preventive protection systems are activated when control systems fail to keep the plant processes within predetermined limits, and protection systems are activated if the control actions of the preventive protection systems are insufficient. The pro-

tection systems have a higher safety category, and they are physically separated from systems belonging to lower safety categories.

The reliability of individual systems is commonly increased by implementing several redundant subsystems that are all capable of performing the same safety function. This design approach is very effective against single failures in, e.g., measurements.

In addition to single failures, I&C system design also takes common cause failures (CCFs) into consideration. CCFs can occur when multiple components fail simultaneously due to the same cause such as, e.g., incorrect maintenance actions, loss of electrical power, environmental conditions (fire, flood, earthquake), or software design errors. In I&C system design, CCFs are prevented by the use of diversity. Diversity is the use of different means to perform the same function. In practice, diversity can be about measuring different physical parameters (e.g., temperature/pressure), using different physical operating principles (e.g., inserting control rods to the reactor to shutdown/injecting boron to the reactor to shutdown), and the use of different design organisations, different vendors or different underlying technologies.

Nuclear power plants have traditionally used conservative technology because of, e.g., the long lifetime of the plants, and due to the strict requirements to demonstrate the safety of the used technology. The first generation nuclear power plants relied on analogue technology, but these systems are becoming obsolete, and digital I&C systems have become increasingly more common. The existing analogue systems are currently being replaced with digital technology, and new plants rely primarily on digital I&C systems.

The use of digital I&C systems poses challenges for functional verification as these software-based systems often are more complex than traditional analogue systems, and cannot be exhaustively tested. Due to the use of defence in depth principles, design errors in these software-based systems typically cannot lead to any hazardous events in the plant. However, software errors can decrease the protection capabilities of the plant by inhibiting individual protection systems. Software errors can also have significant consequences to the plant operation and safety through spurious protection signals, and CCFs, rendering the detection of such errors essential.

3.2 Digital I&C system development

Digital I&C systems differentiate between the platform of the system, and application software that runs on the platform. The platform is an underlying computer system, containing both hardware and software, on which application programs can be executed.

The application program is typically executed using a Programmable Logic Controller (PLC), even though alternative technologies are possible. In this dissertation, it is assumed that the implementation of the platform is correct, and that the application program is PLC-based.

A PLC generally executes a series of instructions cyclically. During each scan cycle it reads inputs, executes a PLC program, and updates outputs based on the computation. In nuclear power plant applications a fixed scan cycle length is typically used.

The international standard IEC 61131-3 [110] defines five languages used for PLC programming: function block diagram (FBD), ladder diagram (LD), instruction list (IL), structured text (ST), and sequential function chart (SFC). This dissertation focuses on the commonly used FBD language. FBD is a graphical design language in which simple elementary function blocks such as AND, OR, or timers are represented as blocks. A function block diagram is built by connecting these function block instances to each other by drawing lines between input and output gates of the function blocks. An example function block diagram following the graphical notations used in this dissertation is illustrated in Fig. 3.1. An input of a diagram may be connected to an output of another diagram. Composite function blocks that consist of several interconnected function blocks may also be defined. Finally, an FBD program consists of a collection of function block diagrams.

One way of developing PLC based systems is to utilise automatic translators. Once a safety function has been implemented as an FBD program it can be automatically translated into a C program that again can be automatically compiled into PLC executable code, see Figure 3.2. This general PLC system development process is followed in several nuclear domain I&C platforms, e.g., AREVA's TELEPERMS XS [5], and Rolls-Royce's Spinline [176]. In these platforms, vendor-specific variants of the FBD language are used that do not follow IEC 61131-3. The actual languages used by the vendors are confidential. In this dissertation it is assumed that the design language follows IEC 61131-3, and it is noted

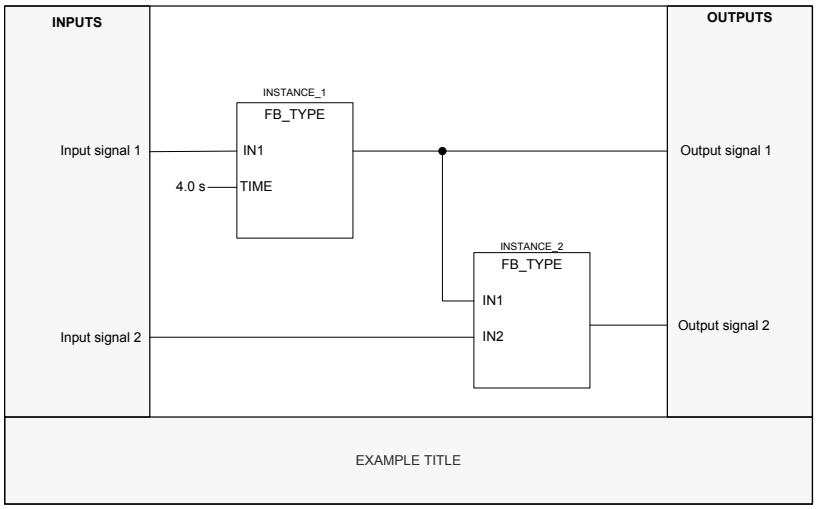


Figure 3.1. An example function block diagram (FBD)

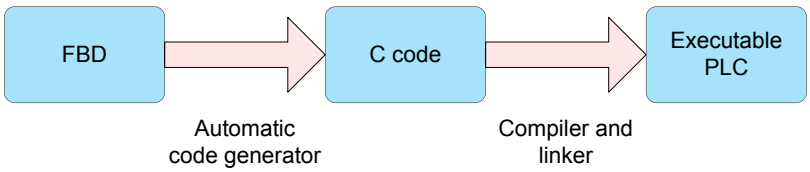


Figure 3.2. PLC system development

that the methodology described in Chapter 5 is also applicable to other vendor-specific FBD language variants.

AREVA uses the SPACE (Specification and Coding Environment) tool (see, e.g., [4]) for developing TELEPERM XS application software. The tool contains a qualified code generator with multiple years of operating experience. Verification of TELEPERM XS systems relies heavily on the correctness of the code generator since it reduces the need for testing the generated code. In the context of TELEPERM XS, the focus of formal verification has been in validating the code generator. The verification process of TELEPERM XS relies heavily on using the SIVAT (Simulation Validation Test Tool) [175] simulator tool, as well as separately performed formal checks to ensure the equivalence of functional requirements, the application program design and the automatically generated code.

In the Spinline platform, application software development is based on the use of SCADE (Safety Critical Application Development Environment) [22], and CLARISSE System and Software Development Environment (SSDE) (see, e.g., [176]). CLARISSE is a dedicated software work-

shop providing the software tools and libraries needed for Spinline system configuration and application software development. SCADE is an industrial tool developed by Esterel technologies.It is based on the synchronous LUSTRE [169] language, and it provides a block diagram formalism for designing the system with well-defined semantics, a design verifier that supports a variety of formal validation tasks, and an automatic tool for C code generation. The design verifier of SCADE is independently developed by Prover Technology.

3.2.1 Model checking using SCADE

The design verifier of SCADE supports two model checking based verification strategies: the proof strategy, and the debug strategy. The proof strategy is based on the use of BDDs, and k-induction. The debug strategy performs SAT-based bounded model checking. [1]

SCADE does not support the use of temporal logic for property formalisation. Only invariant properties can be checked. More complex properties can be checked by modelling the property as an observer using the same modelling language as is used for the design. Huhn et al. [107] have reported severe complexity problems in the SCADE design verifier, but manage to verify medium-sized industrial systems by applying various abstraction and model simplification techniques. Basold et al. [15] have also looked out for alternative verification methods for SCADE programs, but their method has not yet matched the verification capabilities of the design verifier included in SCADE. Wakankar et al. [204] describe the use of SCADE to verify a steam generator pressure control system used in a nuclear power plant.

4. Related work

This chapter reviews the related work with respect to nuclear domain system verification, and the techniques developed in Publication I, Publication II, Publication III, Publication IV, and Publication V. Section 4.1 presents work related to the application of formal methods for verifying nuclear domain systems. Section 4.2 covers model checking of programmable logic controllers. Section 4.3 discusses abstraction techniques and compositional verification techniques. Section 4.4 reviews iterative abstraction refinement approaches. Section 4.5 discusses the use of model checking for fault-tolerance analysis, and Section 4.6 discusses the use of model checking for automatic test generation. Finally, Section 4.7 briefly reviews model checking of full-scale and real-world systems using NuSMV.

4.1 Application of formal methods in the verification of nuclear power plant I&C systems

This section reviews different approaches used for applying model checking and formal methods in general to the verification of nuclear power plant I&C systems.

4.1.1 Use of formal methods in the Darlington nuclear power plant in Canada

The paper by Wassyng and Lawford [208] describes one of the earliest applications of formal methods in a nuclear context. In their approach, a tabular notation is used to describe both requirements and the design, and the correctness of the design is verified using a theorem prover. The approach was successfully applied in the design process of reactor shutdown system software used in the Darlington nuclear power plant in Canada.

4.1.2 Use of model checking in the Paks nuclear power plant in Hungary

Nemeth and Bartha [158, 159] have applied formal methods to the verification of a primary-to-secondary leaking (PRISE) safety procedure used in the Hungarian Paks nuclear power plant. The application software of the system is designed using FBDs, and implemented on the basis of TELEPERM XS system platform. Nemeth and Bartha modelled the FBDs using a Colored Petri Net (CPN) formalism, and proved that the PRISE procedure is initiated whenever a real accident occurs, and that the procedure is not activated when no real accidents have occurred. These properties were proved using state space analysis together with explicit state model checking. Additionally, primary circuit dynamics were also modelled using CPN, and the composition of this model and the PRISE procedure was analysed by simulation.

4.1.3 Formal verification of Korean nuclear power plant automation systems

Formal methods and model checking have been extensively used by researchers in the context of Korean nuclear power plant automation systems. In [123], a computer aided tool-set is described that supports the development of PLC-based systems. The development life-cycle is based on formal requirement specifications, system design using FBDs, and an automatic translation to C programs. A formal language based on both tabular notations and automata is used for requirement specification, and for synthesising FBD programs automatically, see [213].

Model checking and theorem proving are used to verify the correctness of the FBD programs, see [211, 214]. The model-checking approach is based on first translating the FBD programs to Verilog, which is one of the most common Hardware Description Languages (HDLs) used in the design of integrated circuits (IC). Verilog programs are then automatically translated into the input language of the Cadence SMV model checker. In their approach, the temporal specifications used for model checking are manually developed in cooperation with domain experts, and specified in LTL. The model-checking approach is demonstrated in [211, 214] using a nuclear power plant shutdown system as a case study. In the case study, several errors were found that were not noticed during manual inspections.

Verification activities based on equivalence checking are also used. In [211, 214] the VIS verification system is used to verify the behavioural equivalence between an FBD program and a modified version of the same FBD program. In [138], the translation from FBD to an ANSI-C program is verified using the model checker HW-CBMC [62].

4.1.4 Model checking in the Finnish nuclear domain

Model checking has been studied in the context of Finnish nuclear power plant automation since 2007. The method has been used in several research case studies, including an emergency cooling system of a nuclear reactor [199], an industrial arc protection system [199, 128, 124], a change-over switching unit for a busbar [33], an emergency diesel generator control system [131], a stepwise shutdown system [30], and an embedded control software of an uninterruptible power supply (UPS) [89]. Also, two fictive yet realistic case studies have been used: the fictive two-redundant system presented in Publication III, and the fictive nuclear power plant model used in Publication IV.

Model checking has also been used in the Finnish nuclear industry as an independent verification approach. The power company Fortum has utilised model checking in the verification application I&C software [167]. In the Olkiluoto 3 project VTT Technical Research Centre of Finland Ltd performed a model-checking analysis of two safety-critical systems: the Protection System (PS) and the Priority Actuation and Control System (PACS). Due to non-disclosure agreements, further information on the latter assignment is not available.

4.1.5 Other use of formal tools

In the nuclear industry, formal verification approaches have focused a lot on the correctness of automatic code generators, and on analysing generated code using static code analysis tools. ISTec GmbH[1] has developed a reverse engineering tool called RETRANS [86][152] that can be used to check the functional equivalence between generated source code and a FBD specification. The tool has been used in the verification TELEPERM XS systems used in nuclear plants in Bohunice (Slovakia), Paks (Hungary), and Beznau (Switzerland). [86]

[1]Institut für Sicherheitstechnologie (Istec GmbH). `http://www.istec-gmbh.com/`

Static code analysis tools such as PolySpace [75], Frama-C [71], and Astrée [70] have been used by Electricité de France (EDF) [2] to assess software used in nuclear power plants [162]. The techniques are currently being used to analyse the protection system of the EPR Flamanville nuclear power plant. The tools analyse the generated C code, and are used to prove program properties formally, and to detect run-time errors in the code.

The MALPAS [209] toolkit used for the static analysis of generated C code has also been used for analysing nuclear power plant software systems. The tool was used for assessing the Temelín nuclear power plant reactor protection systems in Czech Republic [215], and in the analysis of Sizewell B nuclear power plant primary protection system in the United Kingdom [207].

4.2 Model checking of programmable logic controllers (PLCs)

Model checking of PLC software has been addressed by several authors. Many of the approaches are based on the standard programming languages of IEC 61131-3. This section focuses on model-checking work done in the context of FBD programs. For approaches based on other IEC 61131-3 formalisms, see, e.g., [144], [210], [48], [184], and [41] for approaches based on Instruction List programs; [95] for an approach based on Structured Text; [139], and [108] for Sequential Function Chart approaches; and [177] for an approach based on Ladder Diagrams. For a more extensive survey on the application of model checking to the verification of PLC software, see, e.g., [163].

Several techniques have been developed for model checking FBD programs. Yoo et al. [211], [212], [117] define formal semantics for FBDs and develope translation rules from FBD to Verilog programs that are then automatically translated into the input language of the Cadence SMV model checker. This work is part of the Korean nuclear domain verification research discussed in Section 4.1.3.

Pavlovic and Ehrich [168] describe a process of verifying FBD programs using the NuSMV model checker. Their approach consists of three consecutive transformations, in which the graphical FBD program is first translated into textual format, then simplified by removing redundant circuit variables, and finally translated into the NuSMV modelling language. In

[2] Electricité de France (EDF). `http://www.edf.fr/`

this work the scan cycle of the PLC is implicitly modelled by using a program counter in the model to track the program execution.

Translations from FBD programs to timed automata exist as well. Silva and Barbosa [72] translate both the FBD program and the system specifications into timed automata. They then use a conformance testing tool called UPPAAL-TRON for verifying that the system implementation is according to its specification. Soliman et al. [190, 191, 192] have constructed timed automata models for a set of formally specified function blocks specifically designed to be used in safety applications, namely PLCopen safety function blocks. They have also created a prototype model transformer in Java. Enoiu et al. [83, 82] have created timed automata models based on FBD programs as well. This work is related to test generation and is also discussed in Section 4.6.

Finally, Nemeth and Bartha [158, 159] describe verification of FBD programs in the context of nuclear power plants using Colored Petri Nets (CPN). This work was also mentioned in Section 4.1.2.

The work described in this dissertation abstracts away from the PLC scan cycle, and program counters as in [168] are not used. The biggest difference to previous work is the systematic use of a modular hierarchy enabling larger systems to be easily modelled. In addition, the models in this work are built manually based on early design phase documentation of the system.

4.3 Abstraction and compositional verification

Abstraction is a technique for reducing the state explosion problem. Intuitively, a simplification of a model is called an abstraction, whereas adding more detail to the model is called a refinement. Abstractions can be thought as binary relations between two system models, in which states of the more concrete system are mapped to the states of the abstract system. If the abstract system allows for more behaviour than the concrete system, it is called an over-approximation. The state spaces related to over-approximations tend to be large but can be less complex to analyse by symbolic methods. An over-approximation can lead to spurious counterexamples because of unrealistic behaviour may be induced in the model by the abstraction. Under-approximations, on the other hand, reduce the state space by restricting the behaviour of the model, and can thus lead to false positives.

In this work an over-approximating abstraction technique called compositional minimisation [68] has been primarily utilised to simplify the verification task. The general idea of compositional minimization is that the system is divided into interconnected modules, and a subset of modules is replaced with abstract modules. These abstract modules, or *interface modules* have no intrinsic functionality, and the variables at the interfaces of the modules are completely non-deterministic, making the verification of the abstract system significantly more efficient.

Many other verification approaches based on the compositionality of the system have been developed. The general idea in these compositional verification approaches is that it is often possible to break down a large verification problem into simpler locally verifiable properties, the conjunction of which implies the result of the original problem. Typically, these approaches also assume that the system is composed of interconnected modules.

In assume-guarantee reasoning [181] [171], an assumption is made of the environment of a module, and verified separately. It is then checked whether a particular module fulfils a system property under this assumption. If the property is true in the individual module, it is also true in the whole system. Assume-guarantee reasoning generally requires a lot of human effort, as it is not trivial to come up with assumptions advancing the verification. In [69], a learning algorithm is described that creates assumptions of the environment model automatically, and improves these assumptions based on the results of model checking.

Circular reasoning [149] is another compositional verification approach, in which the correctness of each individual module is verified assuming that the environment of that module also behaves correctly. The circularity of the reasoning is then resolved using induction over time.

In some model-checking approaches a parallel composition of the components of the model is calculated in order to create a single global model that depicts the behaviour of the system as a whole. Some special-purpose model-checking algorithms exploit the compositionality of the model instead. For example, partitioned transition relations [45] and lazy parallel compositions [20] examine the transition relations of different components in the model separately, which can reduce the state space needed for verification.

4.4 Iterative abstraction refinement

The iterative abstraction refinement technique of Publication III is based on the generic verification strategy consisting of the classical four steps: (1) generating an initial abstraction; (2) model checking the property on the abstraction; (3) checking possible counterexamples on the concrete model; and (4) refining the abstraction when needed. This type of iterative abstraction refinement was first introduced by Kurshan et al. [127]. Their technique titled the *localisation reduction* uses non-deterministic abstractions on the variable level and relies heavily on the dependency graph of the variables in the model.

Many other variations to the generic iterative refinement loop have been suggested. Balarin and Sangiovanni-Vincentelli [10] describe a way to automate the iterative abstraction refinement process using a tool based on language containment. Das and Dill [73] have applied predicate abstraction in their version of counterexample guided refinement. In [66, 67], a SAT solver is used to validate abstract counterexamples, and a combination of sampling with Integer Linear Programming (ILP) and machine learning is used in the refinement step.

The Counter-Example Guided Abstraction Refinement (CEGAR) technique by Clarke et al. [61] uses a more general existential abstraction technique based on predicate abstraction that divides the variables into abstract variable clusters. In this approach the validity of the counterexamples is checked using symbolic BDD-based simulation. Refinements are created by inspecting the counterexample in order to locate variables that can be used for partitioning a variable cluster and making the counterexample infeasible.

The technique developed in this dissertation has been inspired by the original CEGAR technique but instead of using predicate abstraction and computing abstract transition relations using predicates, a light-weight approach based on module-level abstractions is used.

In the proof-based abstraction approach [151, 2] the counterexample itself is not used for checking the feasibility of the counterexample or in the refinement step. Instead, a SAT-solver is used to prove that counterexamples up to a certain bound k are not possible, and BDD-based techniques are used to prove the refined model. The idea is similar to the one used in this dissertation except that the feasibility checks are performed using k-induction.

There are also iterative techniques that focus on the compositionality of the system. In [219], an iterative refinement technique based on the modularity of the system has been developed in the context of asynchronic circuits. Refinements in the technique are based on parallel composition. Each component is refined iteratively by checking synchronizations among components with shared interfaces.

Compositionality and dependency analysis have also been used in the context of state/event models. State/event models consist of concurrent finite-state machines that have pairs of input events and output actions that are associated with the transitions of the machines. In [141], a compositional verification technique is presented for state/event models, in which only a few component-machines are initially considered, and if necessary, more component-machines are gradually included based on a dependency analysis of the structure of the system and traversal of the dependency graph. In [18] this technique is used for the verification of state/event models with a deep hierarchical structure. In this work the hierarchical structure is also exploited by reusing earlier reachability checks of superstates of the model to conclude the reachability of their substates.

The iterative abstraction refinement technique developed in this dissertation utilises multiple verification engines in parallel, similar to the generic portfolio-based approach, see, e.g., [193]. To make the parallel approach more diverse, a model translation toolchain is employed, in which the NuSMV models are translated into low level AIGER format models on-the-fly, enabling the utilisation of other competitive model checking tools such as the ABC/ZZ model checker used in the current implementation of the algorithm. The author of this dissertation is not aware of similar iterative abstraction refinement approaches in which the model-checking engines are run in parallel. Multiple different algorithms have previously been applied consecutively and in different phases of the abstraction refinement loop. For example, Glusman et al. [94] have used two different model checkers: a SAT-based technique to check and concretise spurious counterexamples, and a BDD-based model checker to verify the abstractions. Wang et al. [205] employ BDD-based model checking as well as a BDD-ATPG (automatic test pattern generation) hybrid algorithm for verification.

4.5 Fault-tolerance analysis using model checking

There are several previous approaches in which model checking is used to analyse system faults and fault-tolerance.

FSAP/NuSMV-SA [38] is a safety analysis platform in which a system model can be augmented with failure modes, and the fault-tolerance of the system can be analysed by injecting faults into the model and using model checking for verification. The platform also supports reliability analysis, and can, e.g., generate fault trees of the system automatically.

The SCADE system discussed in Section 3.2.1 has also been used in the context of analysing system failures using model checking. Joshi and Heimdahl [121] have analysed Simulink models using the SCADE design verifier. A verification approach based on fault injection is used, and the technique is applied to a simple wheel brake system case study. In [97], a formal safety analysis method called Deductive Cause-Consequence Analysis (DCCA) is integrated in the SCADE framework, allowing safety analysis to be performed using the SCADE design verifier.

Bieber et al. [23] use a combination of fault tree analysis and model checking for safety analysis of complex systems. In this work, the Altarica language [6] is used to model an AIRBUS A320 hydraulic system. The model is separately analysed using fault tree analysis techniques and model checking, and results from both approaches are used together to assess the fulfilment of safety requirements of the system. For other work combining model checking with fault tree analysis, see [122, 52, 161].

In [185], Schneider et al. show how model checking can be used to validate the fault-tolerance of a dual-redundant system for a spacecraft controller. The approach uses a fault injection scheme and the model checker Spin.

In [187, 188], an approach called Hierarchically Performed Hazard Origin and Propagation Studies (HiP-HOPS) is combined with model checking. HiP-HOPS is used to obtain information on component failures based on fault trees and failure modes and effects analysis (FMEA), and model checking is used to verify safety properties based on this information.

Finally, process algebra based approaches for formalising fault-tolerant systems have also been developed, see, e.g., [21] and [42].

The hardware failure modelling methodology described in this dissertation, in contrast to previous work, is about modelling large modular systems in which various different hardware faults can be postulated. The

methodology is built on top of the NuSMV modelling language, and is closely related to probabilistic risk assessment (PRA).

4.6 Automatic test generation using model checking

The classic way of using a model checker to generate test cases was first introduced by Callahan et al. [46] and Engels et al. [81]. In both approaches the negation of a specification is model checked in order to create a counterexample that represents a test case fulfilling the original specification. Since then, numerous variants of this generic test generation principle have been developed, see [88] for an extensive survey.

A few research groups have addressed automatic structure-based test generation in the context of function block diagrams. Enoiu et al. [83] have defined their own coverage criteria for function block diagrams, and utilise the model checker UPPAAL for test generation.

Jee et al. [119] have also developed an automatic test generation technique for FBD programs based on coverage metrics [120, 118] they had previously developed. The main difference of the technique when compared against the technique developed as part of this dissertation is that a Satisfiability Modulo Theories (SMT) solver is used to derive the concrete test cases instead of a model checker.

Automatic test generation has also been used in the SCADE framework. In [76], the tools GATEL and TCG (Test Case Generator) are used for generating test cases based on structure-based coverage metrics.

In Publication V, simple greedy heuristics are used for generating an efficient test set. Several other papers have addressed the efficiency of tests and the test generation process as well. Ammann et al. [3] reduce the size of the test set by removing duplicate test cases and detecting tests which are already a prefix to another test. Gargantini et al. [92] also detect generated tests that are prefixes to other tests, and use the SMV model checker to get the shortest possible test cases.

Hamon et al. [98] have developed a technique called *iterated extension* in which a model-checking tool is modified to search for extensions to previously found counterexamples. This approach reduces the time to generate very long test cases.

There are also techniques for generating efficient test sets that are not related to model checking. For example, automatic test pattern generators (ATPGs) use test compaction to reduce the overall size of the tests.

Test compaction can be applied to both combinatorial and sequential circuits. In combinatorial circuits, two test cases can be run simultaneously when the test vectors use non-conflicting logic values on the inputs. In sequential circuits the test sequences cannot be arbitrarily combined. Instead, the compatibility of the test sequences must be first analysed. See, e.g., Niermann et al. [160] for a description of different algorithms used for compacting a set of tests generated by a sequential circuit ATPG.

In an approach developed by Gargantini and Fraser [91], efficient test sets are generated in the context of Boolean form expressions. Their approach is based on hypothesising faults and applying a set of fault classes on the Boolean expressions. Test predicates are optimised using either a SAT solver or an SMT solver. In one of their optimisation strategies, test cases are designed so that several independent faults can be detected at once. Their work has similarities to what was done in Publication V, as the bounded model checking algorithm used for test generation also relies on satisfiability solvers. In addition, the test collection strategy for optimising the test generation process resembles the heuristics implemented in Publication V.

Automatic test generation approaches related to model checking exist for software code as well. For example, a combination of concrete and symbolic execution of Java programs is used in [93] for test generation in order to exercise a large number of paths in a program. This work is in the context of developing an automated testing environment for a software component providing separation assurance between multiple airplanes.

Fraser and Arcuri [87] have developed a genetic algorithm for test set generation in the context of software code. The algorithm tries to cover all coverage goals simultaneously while keeping the size of the test set as small as possible.

Test set efficiency is also discussed by Campos et al. [47]. They have tried to improve fault localization in software by generating a test set based on its fault detection capability. This is another aspect of test efficiency that can be very useful for debugging. The technique described in Publication V does not strive for improving fault localisation since it is more straight-forward to try to localise errors using traditional model checking based verification.

4.7 Applying NuSMV model checking to full-scale and real-world systems

This dissertation focuses heavily on the use of the NuSMV [51] model checker, (see [150] for an extensive tutorial). NuSMV has been applied to a variety of real-world problems.

In [142], NuSMV is used in the context of artificial intelligence (AI) to verify that multi-agent systems comply with their specifications. In particular, the paper focuses on the verification of temporal epistemic properties of the system.

In [96], a real-world hybrid system modelled as a Fluid Petri Net (FPN) is analysed, and the FPN model is automatically converted into a discrete model, whose specifications are defined using Computational Tree Logic (CTL) and verified using NuSMV.

Choi and Heimdahl [56] have verified safety-critical systems in the domain of aircraft control systems. They have verified a Flight Guidance System (FGS), a Flight management system (FMS), and a fictitious Altitude Switch System (ASW) using an abstraction technique they call domain reduction abstraction. In domain reduction abstraction the input domain (environment) of the system is simplified by a division of input variable values into equivalence classes. A representative set of data values for each equivalence class is selected using a linear/integer programming tool.

Miller [154] describes five succesful examples of using formal methods in the development of high-integrity systems. Three of the cases utilise NuSMV: the verification of Flight Control System (FCS 5000), the verification of the Adaptive Display and Guidance System (ADGS-2100), and verification of the Operational Flight Program (OFP). All three systems were initially modelled using Simulink, and a translation from Simulink models to NuSMV is utilised. In all three cases, several previously unknown errors were discovered (26 errors in the Flight Control System, 98 errors in the Adaptive Display and Guidance System, and 12 in the Operational Flight Program).

Miller et al. have also studied the early validation of system requirements [155]. In this work, a formal model of a Flight Guidance System was created based on the system's requirements. Functional and safety requirements of the system were then captured as natural language "shall" statements and manually translated into formal properties

over the model. The shall statements were then validated against the formal model using NuSMV.

In [49], the Computer Based Interlocking System (CBI) used for ensuring safe train movements at a railway station is verified using NuSMV. A search algorithm is first used to generate interlocking tables based on the system description, originally given in a domain specific language. The interlocking tables are then translated into a NuSMV model, and verified.

Several approaches related to safety analysis have also been used. In [195], a particular function of a flight management system is analysed by first performing a hazard analysis of the system, and then identifying general categories of errors by conducting a Fault Tree Analysis and a Failure Mode Effects Analysis. A list of safety requirements is then developed based on these error categories, and verified using model checking.

In [146], an extension of the NuSMV tool called nuXmv is used to formally compare different designs of an automated air traffic control system. As part of this work, the models of the alternative system designs are also extended with faults, and the resulting fault trees are analysed and compared with each other.

Bozzano et al. [37] perform a complete formal analysis of the AIR6110 wheel brake system including contract-based design, model checking and safety analysis. The different architecture-level designs of the system are recreated in a formal manner, the behaviours of the different design level models are automatically analysed and compared, and the functional correctness of the system is verified. Several formal techniques are used, including model checking using the nuXmv tool. In addition, fault trees and probabilistic reliability measures are automatically produced as part of the safety analysis.

5. Methodology for modelling FBD programs

This chapter summarises the main results of Publication I, and presents modelling methodology for FBD programs. The methodology focuses on using the modelling language of the NuSMV model checker, and it is the basis for the models and case studies discussed in Publication II, Publication III, Publication IV, and Publication V.

The methodology presented here is the result of collaboration between many researchers. The contribution of Publication I is in the abstractions used for scan cycle discretisation and environment modelling. The utilisation of a function block library, and the practical modular modelling conventions described in Section 5.6 (see also [165]) are based on earlier research work at VTT, and the author of this dissertation has no contribution on these aspects of the methodology.

Section 5.1 first discusses the scope of modelling. Sections 5.2, 5.3, 5.4, and 5.5 cover methodological issues related to time discretisation and the environment model. Section 5.6 presents a modular technique for modelling FBD programs in the modelling language of NuSMV. Section 5.7 briefly covers issues related to requirement formalisation. Section 5.8 discusses examples of errors found using model checking. Finally, Section 5.9 discusses the limitations and threats to validity of the modelling approach.

5.1 Scope of modelling

As was mentioned in Section 3.2, the application software of safety automation systems is commonly designed using vendor-specific derivatives of the FBD language, and the FBD programs are typically used as a basis for code generation. As far as this dissertation is concerned, it is assumed that the software performing the code generation is trusted, and is not

expected to have errors. The model-checking approach of this dissertation focuses solely on the FBD programs. Source code is thus not used as an input for modelling, even though analysing the source code could lead to more accurate results. This is mainly due to: (1) contractual difficulties in obtaining and using source code in a research case study; (2) function block diagrams, unlike source code, are available in early design phases where it is still possible to make changes with moderate cost and effort; and (3) superior analysability of the function block diagram formalism when compared to the source code.

Furthermore, the models described in this dissertation focus solely on the logical function of a system, and the hardware and system level aspects are left out. Only application software is in the scope of verification, and all operating system software, platform software, or any data transmission issues are excluded. It is also assumed that the function blocks operate as specified, and that the controller running the application software is running correctly, and there are, e.g., no race conditions possible. In Chapter 7 the scope of modelling is extended to cover also other system aspects such as hardware failures.

5.2 Environment model

The essential idea in environment modelling is to allow the environment to behave freely so that no behaviour is excluded due to wrong modelling choices. Thus, all variables of the environment are free variables that get their values non-deterministically at every point in time. This sort of over-approximation retains the truth value of universal properties such as LTL properties [65, Chapter 13]. This is because the over-approximated system has more behaviours than the real system, and the approximated system includes all the realistic behaviours as well. If some unsafe behaviour cannot be found in the over-approximated system, then the behaviour cannot be found in the concrete system either. On the other hand, in over-approximated systems false negatives (i.e., counterexamples that are due to the environment acting up in a way that is not realistic) are possible in the system verification. This means that the efforts required for analysing counterexamples and specifying the examined specifications are somewhat increased. Typically the false negatives can be ruled out by carefully stated specifications. Another thing to consider when making over-approximating abstractions is to make sure that the performed ab-

straction actually is an over-approximation.

5.3 Cyclic operation of the PLC

An FBD program is executed by repeatedly running a scan cycle. During a scan cycle the PLC reads the inputs of the FBDs, executes the logic elements (function blocks) within the FBDs, and updates the outputs. In safety-critical systems it is typical that the scan cycle is of constant length, and dynamic features associated with varying scan cycle lengths are not recommended.

The execution of the logic elements in an FBD means that all the individual function blocks read their inputs and update the outputs. The execution order of the function blocks may be explicitly ordered. However, an explicit execution order is not required by the IEC 61131-3 standard, and instead the order can be implicitly determined by following a set of rules given in the standard. The standard requires, for example, that a function block may not be evaluated before all of its inputs have been evaluated within a scan cycle.

5.4 Modelling of time and analogue variables

The model checker NuSMV operates in discrete time steps. Due to this, the modeller needs to decide what a single time step in the model corresponds to. This decision can have significant consequences on the behaviour of the system model, see the discussion in Section 5.9.

In the literature, three different approaches are typically used for modelling the scan cycle: explicit modelling, implicit modelling, and abstraction from the scan cycle. The explicit modelling focuses on the real time aspects of calculations performed during a scan cycle. The implicit modelling approch focuses on the individual instructions performed by the PLC during a scan cycle but the duration of the scan cycle is not of interest. Finally, models that abstract from scan cycles assume that the program execution is instantaneous. This approach is most useful when the environment of the model is much slower than the scan cycle length. [143]

The models used in this dissertation abstract away from the scan cycle. A single discrete time step of the model corresponds to executing the en-

tire scan cycle. It is assumed that the scan cycle length is constant, which is the case in typical nuclear domain software systems.

It is also assumed that communication between individual FBD programs is synchronous and instantaneous. In the final implementation of a system two FBD programs that communicate with each other via their inputs and outputs can be physically implemented on separate computers. In these cases certain data transmission protocols and delays may be involved in their communication. Typically it is the task of platform software to handle data transmission issues in such a way that they are invisible to application software. From the point of view of application software the assumptions are therefore reasonable and correspond to reality assuming the automation platform operates within specified timing requirements. These issues are left out of consideration and instead it is assumed that communication between all FBDs is fault-free, seamless, and that no transmission related delays occur.

Another modelling choice is to decide the length of the time step used in the model. Ideally, the time step should correspond to the scan cycle length used in the actual system. If a longer time step is used in the model, it is possible that certain behaviours of the system are missed. The issue is discussed further in Section 5.9.

Because of the discrete nature of the NuSMV tool, function blocks involving delays are modelled using counter variables of integer type. Analogue variables of the system need to be discretised as well. An often used modelling technique is a division into equivalence classes. When control logic is analysed that does not involve complex arithmetics and operates on a constant scan cycle length, the discretisation of counters and analogue variables can typically be applied in a way does not lose any control functionality of the original system, and can therefore be justified.

5.5 Justification of time discretisation

In the models used in this dissertation, the real time aspects of the system are handled by using discrete time models and counter variables. The control systems that are implemented based on the FBD programs use a discretised constant length scan cycle, and this justifies the use of time discretisation in this dissertation. On the more theoretical level, the problem of timer discretisation and its justification has been studied in several papers, see, e.g., [105] and [7]. Henzinger et al. [105] discuss veri-

fication methods that assume that time is observed at integer time points only, and the use of such methods to verify real time systems. First off, it is noted that restricting oneself to discrete trace models that formalise the behaviour of a system as an infinite sequence of snapshots of the global state at different times is adequate for modelling discrete processes, which change state only finitely often between two time points. The paper also shows that for real time systems that can be modelled as timed transition systems, and the class of properties including time-bounded invariance and time-bounded response, the verification problem can be reduced (digitised) to an integral-time problem. In addition, it is shown that even systems that cannot be digitised can benefit from integral-time verification, when conservative approximations on the time bounds are used. The implication with regard to this work is that since programmable logic controllers are discrete processes that operate finitely often between two time points, it suffices to use discrete trace modelling for them. Thus, the integer model of time used in the models of this dissertation can be justified.

Another more informal justification for time discretisation is that the analyses performed in this dissertation focus solely on extrinsic properties of the system. Namely, only properties that refer to the values of the inputs and outputs of the PLC are examined. For these extrinsic properties the intermediary values of variables calculated during a PLC cycle are not meaningful, and the only meaningful time point is at the end of the PLC cycle when the outputs are updated. It is also quite simple to discretise PLC behaviour as it by nature operates periodically. The models in this work assume that the intervals between consecutive PLC cycles are constant, which is the case in typical safety-critical systems.

Another issue related to time discretisation is the use of a coarser discretisation than the one used in the actual system (using a time step that is longer than the scan cycle length of the system). A coarser time discretisation was used for example in Publication III in the modelling of the emergency diesel generator control system. In Publication III the model was used in the evaluation of the developed iterative abstraction refinement technique, and the coarsening was necessary because a model with a coarser time discretisation was more serviceable for analysing a large set of benchmark properties in reasonable time. In Publication II, however, a discretisation of time corresponding to the actual system scan cycle length was used for verifying the same emergency diesel generator control system.

Utilising a coarser discretisation of time is not a sound method because it might ignore some important system behaviour (see Section 5.9). Instead, an approach based on using over-approximating delay function blocks with non-deterministic choice [64] could be a potential way to abstract the scan cycle length in a sound way. Such an abstraction approach could also be used together with the modelling approach presented in this dissertation. Similar over-approximations of delays have been used in research case studies closely related to this dissertation, see, e.g., [200] and [90]. The reason for not using an over-approximating approach in the first place is that a typical nuclear domain safety system can involve several time delays, and the lengths of the delays can vary from fractions of seconds to several minutes. Over-approximating the lengths of time delays can easily lead to a great number of spurious counterexamples, rendering the interpretation of verification results overly complicated. The development of a systematic over-approximation approach for FBD programs that handles this issue, however, is left for future work.

5.6 Modelling FBD programs

The employed modelling approach [165] exploits modularity as much as possible. Each function block type is modelled as a reusable module, and the correctness of the function blocks is verified separately. The function block modules constitute a function block library that can be imported into the models. When an FBD program is modelled, the elementary function blocks are instantiated using the library, and connected together. The model code corresponding to some often used function blocks can be found in Appendix A, along with short explanations on the functionalities of the blocks.

An example FBD shown in Figure 5.1 has three inputs: `START_ALLOWED`, `START_BUTTON_PUSHED`, and `RESET_START_SEQUENCE`. The single output `START` is calculated using three function blocks: an AND gate, a set dominant flip-flop memory, and a TON (Timer ON) timer. In the example system, whenever starting is allowed and the start button is pressed, the `START` output should be set and memorised. Also, a reset command should eventually reset the `START` output whenever the start button is not pushed or the start is not allowed. The function blocks types used in the example system are also listed in the Appendix.

A model is typically divided into several modules according to the dif-

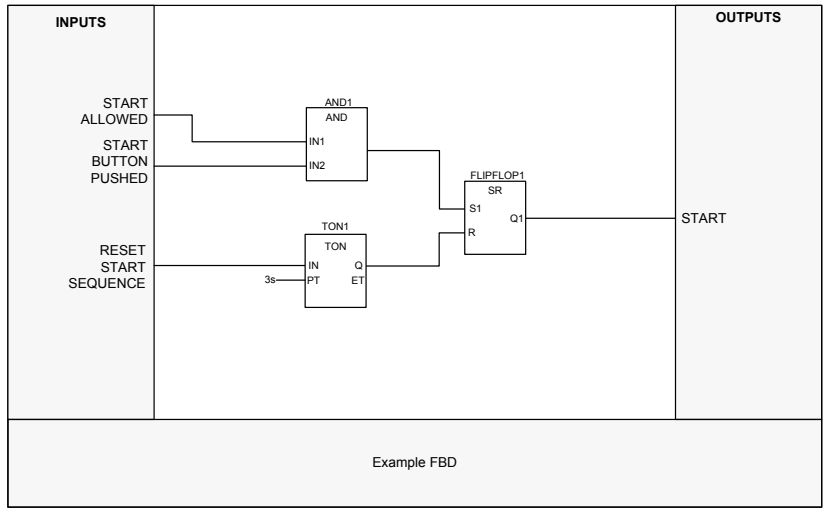


Figure 5.1. An example FBD

ferent functions on the top-level. The modularity on the top level allows redundant structures to be modelled more efficiently.

In the example case, the FBD may be encapsulated into a single reusable module. The idea is illustrated in Figure 5.2 where two redundant instances of the example FBD of Figure 5.1 are created. Both instances are treated as their own modules. The modules receive the same inputs, and their outputs are collected in an OR function block `VOTE_1oo2_1`. The voting function block sets the `START` output of the system whenever at least one of the signals received from the `EXAMPLE_FBD` modules is true. This encapsulation of certain parts of the model into reusable composite function blocks allows deep hierarchies to be implemented in the model. The techniques introduced in Chapter 6, however, operate on a non-nested hierarchy structure, in which the modules instantiated at the top level only create instances of the elementary function blocks and do not create instances of other modules.

Encapsulating the example FBD of Figure 5.1 as a composite module results in the model code shown in Listing 5.1. The inputs of the system are parameters of the module, and the output of the system `START` is in the `DEFINE` section of the module. The function block instances are created in the `VAR` section of the module. The inputs of the flip-flop function block instance `FLIPFLOP1` are connected to the outputs of the `AND1` instance and the `TON1` function block instance. In this example, it is assumed that the scan cycle length related to the design is 100 ms. Subsequently, the 3 s delay associated with the TON timer translates into 30 scan cycles,

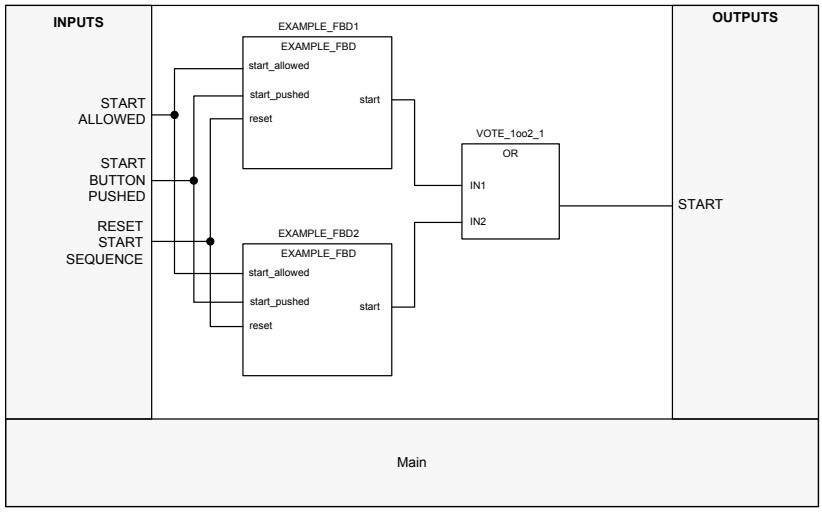


Figure 5.2. Two redundant instances of an encapsulated FBD are instantiated in the main module.

which is given as parameter to the TON block.

```
MODULE EXAMPLE_FBD(START_ALLOWED, START_BUTTON_PUSHED,
  RESET_START_SEQUENCE)
VAR
  AND1 : AND(START_ALLOWED, START_BUTTON_PUSHED);
  TON1 : TON(RESET_START_SEQUENCE, 30);
  FLIPFLOP1 : SR(AND1.OUT, TON1.Q);
DEFINE
  START := FLIPFLOP1.Q1;
ASSIGN
```

Listing 5.1. NuSMV code for the encapsulated example FBD

```
MODULE main
VAR
  START_ALLOWED : boolean;
  START_BUTTON_PUSHED : boolean;
  RESET_START_SEQUENCE : boolean;

  EXAMPLE_FBD1 : EXAMPLE_FBD (START_ALLOWED, START_BUTTON_PUSHED,
        RESET_START_SEQUENCE);
  EXAMPLE_FBD2 : EXAMPLE_FBD (START_ALLOWED, START_BUTTON_PUSHED,
        RESET_START_SEQUENCE);
  VOTE_1oo2_1 : OR (EXAMPLE_FBD1.START, EXAMPLE_FBD2.START);
DEFINE
  START := VOTE_1oo2_1.OUT;
```

Listing 5.2. NuSMV code for the main module

If the FBD program depicted in Figure 5.2 is modelled, two instances of this module need to be created in the main module. The corresponding model code is in Listing 5.2. In addition to the two composite modules, an instance of the OR function block is also created. The inputs of the environment of the system are defined as free variables in the VAR section of the main module, and the output of the model is defined as a DEFINE macro in the main module.

5.7 Requirement formalisation

Requirement formalisation goes hand in hand with modelling since the requirements are based on the selected level of abstraction, and the interface between the model and the environment. The main issues regarding requirement formalisation are related to minimising the effort needed for formalisation, verifying the correctness of formal specifications, and improving the understandability of formal notations. In this dissertation, requirement formalisation is not thoroughly addressed even though it is an important aspect of model checking. These problems have been studied widely by other researchers as discussed further below.

Several papers describe the use of a tabular notation in which the requirements are formally expressed as a set of relationships between monitored system variable values and required control actions. Lawford et al. [137] have utilised a tabular method to the specification and verification of systems used in the Darlington nuclear power plant.

Heitmeyer et al. [104] have used formal consistency checking to detect errors in requirement specifications expressed in the SCR (Software Cost Reduction) tabular notation. Furthermore, the application of the SCR tabular method to three NASA systems, and lessons learned are discussed in [103].

Cimatti et al. [58, 59, 54] have developed a methodology and a series of techniques for the formalisation and validation of requirements for safety-critical applications. The methodology consists of three phases: informal analysis, formalisation, and formal validation. They use techniques based on model checking for checking consistency, checking whether an expected property is implied by the formalised requirement fragments, and checking the compatibility of scenarios given the constraints imposed by formalised requirements. The techniques were applied within an industrial

project in the railway domain.

Several different approaches have also been developed for specification debugging. A model-checking tool only states that a specification is true on a model but it cannot determine whether the examined specification is meaningful for the model. Vacuity checking approaches (see, e.g., [17]) utilise witness traces of specifications to demonstrate that the specifications are not trivially valid. For instance, propositional logic formulas that contain implications are trivially valid when the pre-condition of the implication is not satisfiable. In [179], another sanity checking approach is introduced in which an LTL specification and its negation are both checked for satisfiability in order to detect specifications that are true in all models. In [180], a multi-encoding implementation of this approach is developed that can be used as a front-end to NuSMV.

Finally, restricted vocabularies and approaches based on the use of requirement templates have also been used to support property formalisation. For instance, see [194] for work closely related to this dissertation. In this paper, Tommila and Pakonen have utilised Controlled Natural Language (CNL) and identified temporal logic templates for requirement constructs commonly used in the context of safety critical I&C systems.

5.8 Typical errors found using model checking

Experience in utilising model checking for the verification of industrial safety systems has shown that the errors found using model checking are typically caused by not taking into account: (1) mistimed manual actions; (2) sensor, communication or hardware failures; (3) events occurring in an unexpected order; or (4) simultaneity of several signals. These kinds of issues are hard to take into account in test design. [166]

The use of certain function blocks also seems to correlate with problems in the design. Rising edge-triggered pulse blocks, set-reset flipflops and complex non-standard function blocks have been especially problematic. The pulse function block is triggered only by a rising edge and input changes during the pulse are ignored. If the input is set during a pulse, the output of the time pulse block can freeze to zero. With the flipflop function block the difficulty is in intuitively understanding how changing the prioritisation or the initial value of the function block affects the system behaviour.

An example of the problematic use of pulse blocks is described in Publi-

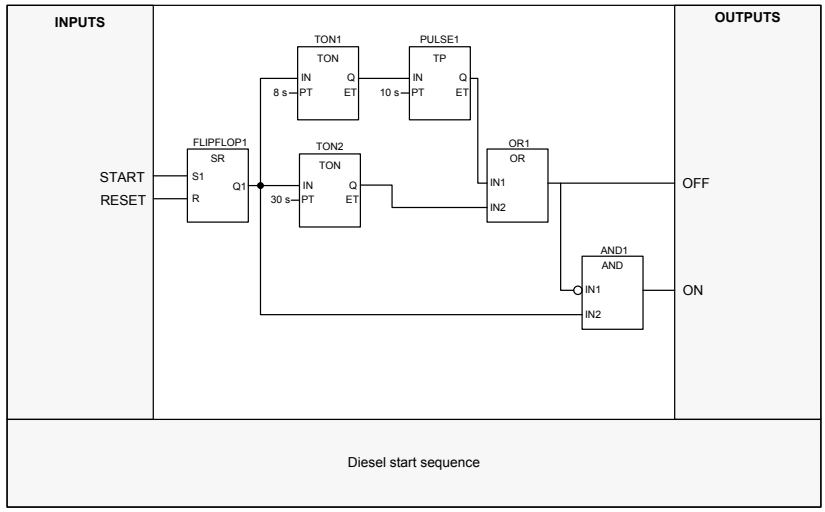


Figure 5.3. Part of the design for starting the diesel generator

cation II, where the control system of an emergency diesel generator was verified using model checking. One of the tasks of the diesel control system was to give start commands to the diesel generator according to a specific starting sequence. The desired starting sequence is such that the diesel generator is first given a command to start (ON) for 8 seconds. It is possible for the diesel generator to not start on the first attempt. After a 10 second rest period, the ON command is again given for 12 seconds. The part of the logic of the system implementing this sequence is illustrated in Figure 5.3. The logic consists of a set dominant flip-flop, two TON timer blocks (8 s, 30 s), a time pulse function block (10 s), AND-block and an OR-block.

The design is such that the sequence may be interrupted by a reset signal if the reason to start the diesel generator is no longer valid. An interruption to the starting sequence should be performed in a safe way. In particular, it is expected that the ON-signal is not given continuously for long periods of time, since this might be harmful to the device. However, when the system was analysed using model checking, a counterexample could be found in which the ON signal is continuously set for over 20 seconds. This long continuous ON signal is possible when two consecutive starting sequences interfere with each other.

Figure 5.4 illustrates the timing diagram related to the scenario. In the scenario, the start sequence is first initiated at the 0 s mark (Step 1 in Figure 5.4). After 8 seconds, the output of TON1 initiates PULSE1. Immediately after this a reset command is given followed by another start

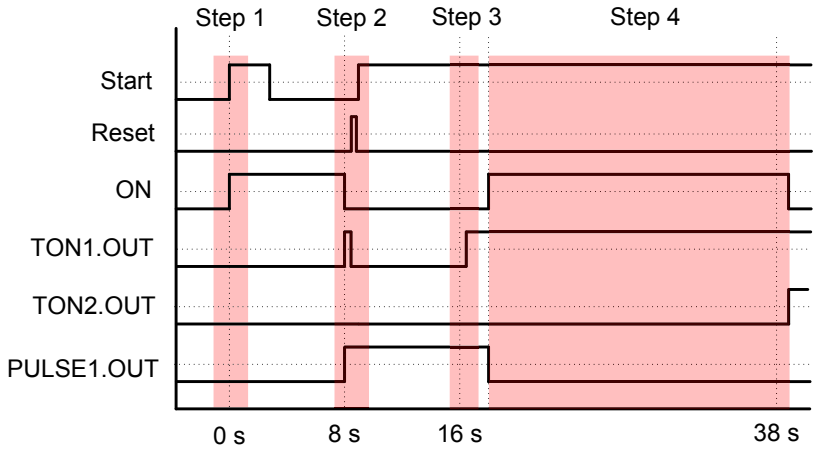


Figure 5.4. Timing diagram of the error scenario. Two consecutive starting commands are given (Steps 1 and 2). At the 16 s mark (Step 3) TON1.OUT is supposed to initiate the 10 s PULSE1. The rising edge of TON1, however, is not detected as the pulse is still running. As a result, the ON output is continuously set for over 20 s (Step 4).

command (Step 2 in Figure 5.4). At the 16 s mark, the output of TON1 should again initiate PULSE1. However, the previous starting sequence is still on-going (PULSE1 is running), and the rising edge from TON1 is not detected (Step 3 in Figure 5.4). As a result, as the time pulse ends, the ON output is set for over 20 seconds, and is finally reset when the output of TON2 is set (Step 4 in Figure 5.4).

Based on observations made in Publication I, it is recommended that more attention is put into: (1) testing of boundary values of systems; (2) testing of manual operations; (3) testing of system behaviour during hardware failures; (4) covering various timed sequences of inputs instead of mere combinatorial coverage; and (5) testing of parts of the design where time pulses, feedback or complex non-standard function blocks are used.

5.9 Threats to validity and limitations

Different function-block-based design paradigms, such as the distributed function block diagrams described in IEC 61499 [111], exist. The methodology described in this dissertation is intended only for modelling FBD programs that follow the semantics of the IEC 61131-3 [110] definitions, and vendor-specific variations thereof. The model-checking tool and modelling capabilities fit well with decision logic functions with many Boolean variables, memories, time delays, and some feedback. However, not all

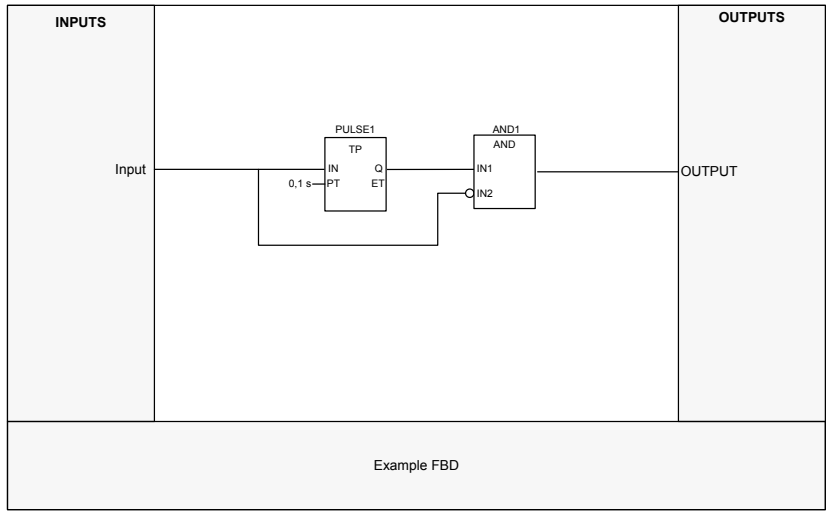


Figure 5.5. An example function block diagram illustrating an intricate timing scenario

functions that can be implemented using FBDs can be modelled and verified. Control logic implemented using proportional-integral-derivative (PID) controllers is one example of this. These control loop feedback mechanisms either cannot be modelled in the NuSMV modelling language, or lead to state explosion in the model-checking analysis. The NuSMV tool also has very limited support for complex arithmetic. Calculations involving real-valued variables and complex mathematical functions can not be performed. For example, the tool has a division operator that can be used but the results are rounded to the closest integer value.

Some of the assumptions made in the modelling are such that certain kind of errors are not found. Our modelling methodology does not take into account the asynchronicity issues of real systems. Thus, errors caused by, e.g., race conditions or errors caused by small differences in clock cycles of distributed computers are not found.

In some cases a coarsening of the scan cycle length of the PLC may be required to make system verification feasible in practice. The models used in this dissertation mainly employ a discretisation of time corresponding to a realistic system scan cycle length. An exception is the model of the emergency diesel generator control system. The version of the model used as a benchmark in Publication III utilised a coarser time discretisation so that analysing a large set of benchmark properties could be done in reasonable time. In Publication II, however, a discretisation of time corresponding to the actual system scan cycle length was used for verifying the same emergency diesel generator control system.

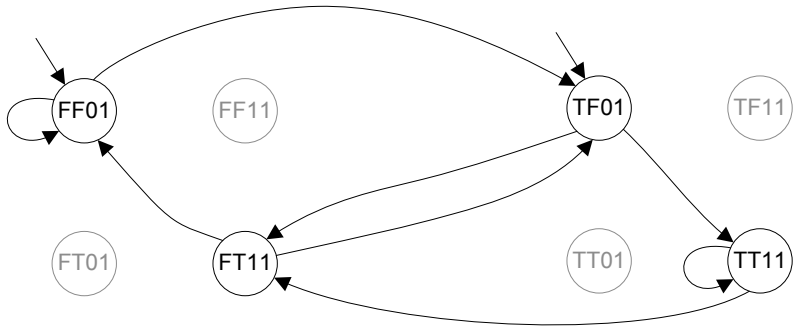


Figure 5.6. State space of the intricate timing example, with a scan cycle length of 0.1 s. The states are labelled with four values: the Boolean value of the input of the system (T/F); the Boolean value of the internal `prev_IN` variable of the pulse (T/F); the integer value of the `clock` variable of the pulse (0/1); and the `PT` parameter of the pulse. The output of the model is false in all reachable states.

A coarser time discretisation may be required in, e.g., systems where the application logic contains long time delays. Multiple second delays together with the short time step length cause state space explosion in the model. In addition the counterexamples tend to become overly long. A quick solution to this is to use a longer scan cycle length in the model, or modify the delays implemented in the application logic. If the model is modified in such a way, certain intricate timing related errors may not be found. The FBD in Figure 5.5 is such that certain behaviour is left out if the scan cycle length is extended in the model. The FBD includes a very short pulse and an AND block with one of its inputs negated. The pulse function block works so that whenever a rising edge is detected on its input, then the output is set for 0.1 seconds. If the input disappears during this pulse the output still remains set. The output of the FBD is true whenever the pulse is on and the input is not set. For this to happen, the input must first become true to set the pulse, and then quickly (within the 0.1 seconds) set to false so that both inputs of the AND block are true. The behaviour of the model depends on the scan cycle length. In a model where the scan cycle length is set to 0.1 seconds the length of the pulse corresponds to only a single time step, and the output of the function block diagram can never become true.

The reachable state spaces of two alternative implementations are depicted in Figures 5.6 and 5.7. If the intricate timing example is modelled with a scan cycle length of 0.1 s (Figure 5.6), the length of the pulse is a single time step (the `PT` parameter of the pulse is 1), and the output of

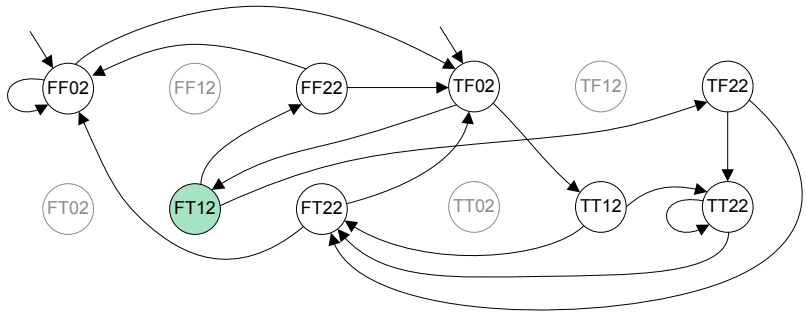


Figure 5.7. State space of the intricate timing example, with a scan cycle length of 0.05 s. The states are labelled with four values: the Boolean value of the input of the system (T/F); the Boolean value of the internal `prev_IN` variable of the pulse (T/F); the integer value of the `clock` variable of the pulse (0/1/2); and the `PT` parameter of the pulse. The reachable states in which the output becomes true are highlighted in green colour.

the model is false in all reachable states. If, however, a shorter scan cycle length of 0.05 s is used (Figure 5.7), the length of the pulse corresponds to two time steps (the `PT` parameter of the pulse is 2). Consequently, the state space of the model grows, and a state in which the output is true becomes reachable. The model code of the pulse function block is presented in Appendix A.

To avoid incorrect results on all models, the use of a time step length corresponding to the actual scan cycle length of the system is strongly encouraged. If for model checking scalability reasons a coarsening of the scan cycle length has to be made, very careful expert judgement should be used when analysing the results.

Also note that design choices such as the one in Figure 5.5 are potential sources for error, as the scan cycle length of the system may change during the design life-cycle. If a system is eventually operated on a different scan cycle length than what was originally intended, the system may become functionally different in a critical way.

Finally, since the environment of the system is completely free the modeller should be careful when verifying, e.g., existential properties of the system. The free environment of the model may allow certain behaviours that will not occur when the actual system is executing.

6. Iterative abstraction refinement on modular systems

This chapter describes the contents of Publication II and Publication III, and presents an iterative abstraction refinement technique for modular systems.

The methodology presented in Chapter 5 together with currently available classical model-checking methods suffices well for verifying individual safety functions. However, especially in safety-critical domains it is common to improve the system's tolerance to hardware failures by implementing several subsystems that execute the same protection function using design diversity in software and/or hardware. These diverse subsystems may need to be analysed simultaneously to check that no unintended interactions between the subsystems exist, and because system specifications may refer to their combined behaviour. The classical model-checking methods (such as BDD-based techniques, SAT-based bounded model checking, k-induction or the PDR algorithm) do not always scale sufficiently well to analysing these large and complex systems.

The scaling problem can often be avoided by creating abstractions of the model that are easier to verify. Depending on the examined specification, only a small portion of the model may be significant to the analysis. In Publication II it is described how insignificant parts of the model can be left out of the analysis using over-approximating abstractions.

Unfortunately, creating such an abstract model for each checked specification is non-trivial and requires quite a lot of manual work, becoming tedious and thus also error prone. For the best efficiency gains, the abstraction must be tailored for each specification separately.

In Publication III, it is described how these kinds of over-approximating abstractions can be created automatically using an iterative abstraction refinement technique. The described technique: (1) significantly reduces the amount of manual work needed to create the abstractions; (2) proves

system correctness based on verification runs automatically performed on the abstractions; and (3) can often significantly reduce the overall computational effort required for model checking, enabling the model checking of larger system models.

The technique is based on partitioning the system into modules and creating module-level over-approximations, as well as an iterative procedure for refining the abstraction level. In addition, the efficiency of the technique is improved by running several model-checking engines in parallel, focusing only on finding proofs for properties, instead of looking for counterexamples. In order to get faster verification results for both true and false specifications, it is useful to run the algorithm simultaneously with another approach that is good at finding counterexamples quickly, such as traditional bounded model checking. An implementation of the technique that is used in this dissertation is available online.[1]

Section 6.1 discusses the over-approximating abstractions. Section 6.2 goes through the iterative abstraction refinement technique. Section 6.3 presents results on applying the technique, and Section 6.4 discusses the validity of the technique.

6.1 Module level over-approximations

The developed technique works with systems that can be partitioned into modules. The modular partition must be such that the modules are on a single level of hierarchy, i.e., a module does not contain another module. The systems studied in this work consist of a set of FBDs. Typically a single FBD implements a single function of the system. In these systems it is simple to use a modular partition in which a single FBD corresponds to a single module. This partition is also quite natural as it is dictated by the design process. The resulting modules are compact, and the connections between the modules are limited. The restriction of the developed technique to models having a single level of hierarchy is not a major obstacle since all FBD programs can theoretically be organized syntactically on one level.

Figure 6.1 illustrates the modular partition of the fictional system used as a case study in Publication III. The system consists of 18 FBDs each separated as its own module. The big rectangles in Figure 6.1 represent modules and the blocks inside the rectangles represent individual func-

[1] https://github.com/JussiLahtinen/Dissertation

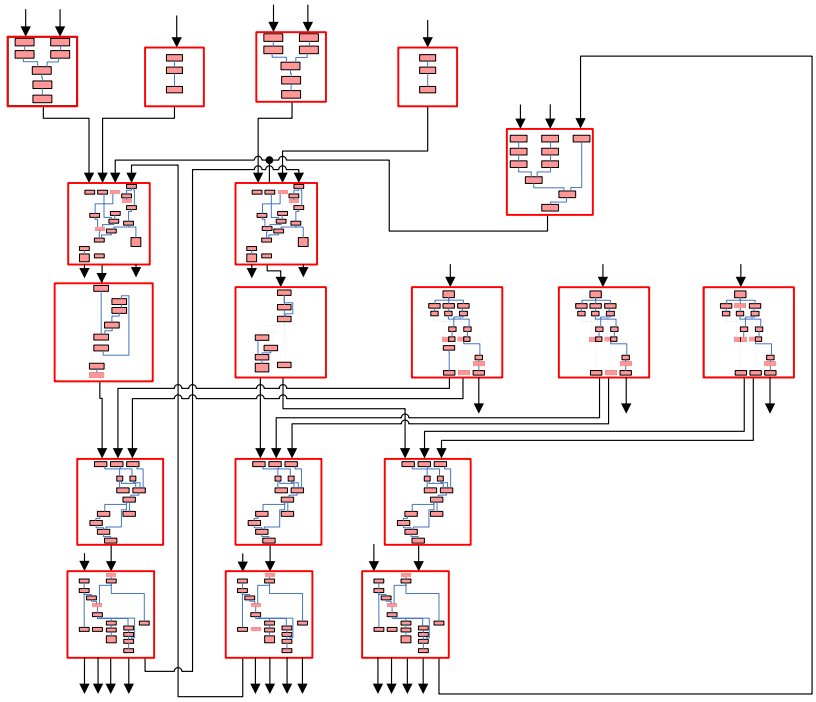


Figure 6.1. The modular partition of the fictional system used as a case study in Publication III, in which the system consists of 18 modules. Each module contains an FBD.

tion blocks. The modules have connections with each other as some of the outputs of the FBDs are used as input in other FBDs. Also, many of the inputs and outputs are not connected to other FBDs. Instead, they are connected to the environment of the model. In our models, however, the environment is free so the inputs/outputs are not connected anywhere and thus may obtain a different value at each time step.

Several other ways to split the system into modules exist. It would be interesting to see whether changing the level of coarseness of the modules can have influence on the performance of our technique. Such alternative system partitions are primarily left for future work.

6.1.1 Abstractions of the model

The abstractions in our technique are created on the module level so that a whole module is always replaced with an abstract version of it. The abstractions are based on the compositional minimisation technique, in which the system is abstracted using reduced versions of some of the system's modules. Such a reduced module version is called an *interface mod-*

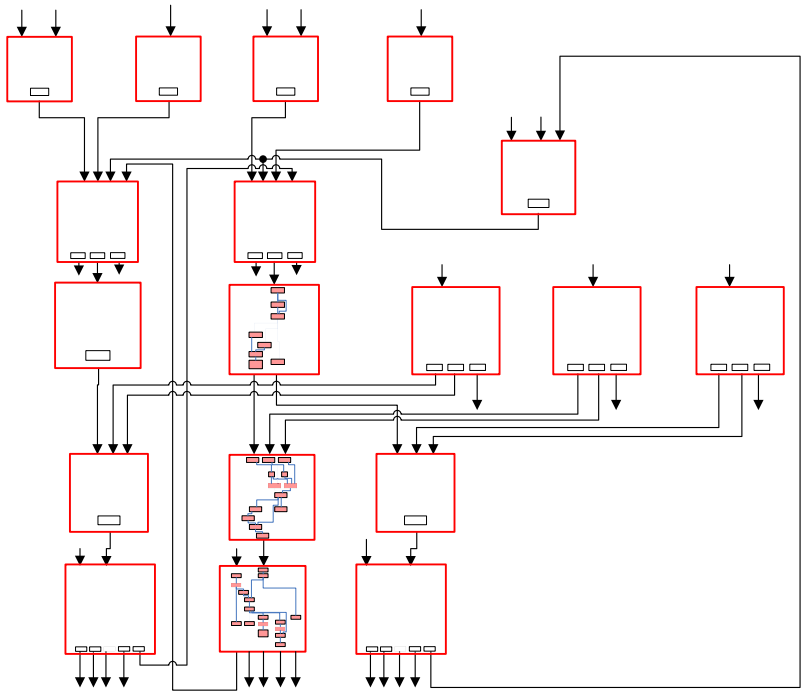


Figure 6.2. The fictional system where 15 out of the 18 modules have been replaced with interface modules

ule. It has the same input and output interfaces as the concrete module it substitutes, but no internal state. No restrictions are set to the outputs: they are completely non-deterministic. A model-checking model of an interface module does not contain any function blocks. Instead, only the outputs of the module are defined as free non-deterministic variables. For example, the interface module corresponding to the example FBD in Figure 5.1 and Listing 5.1 is illustrated in Listing 6.1.

An abstraction of the system is created by selecting either the concrete version or the interface version of each module. The idea is illustrated in Figure 6.2, in which 15 out of the 18 modules have been replaced with an interface module.

```
1 MODULE EXAMPLE_FBD(START_ALLOWED, START_BUTTON_PUSHED,
2 RESET_START_SEQUENCE, CYCLES_IN_SECOND)
3 VAR
4   START : boolean;
5 DEFINE
6 ASSIGN
```

Listing 6.1. NuSMV code for the interface module of an FBD

The abstraction discussed above is such that there is a simulation rela-

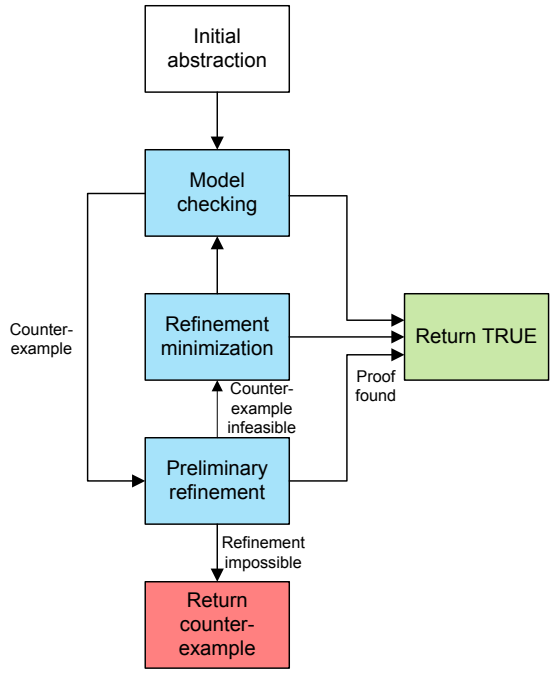


Figure 6.3. The iterative abstraction refinement loop for verifying large modular systems

tion between the abstract model and the concrete model, i.e. each transition of the concrete model can be matched by some transition in the abstract model. The interface modules of the abstract model are over-approximations of the modules' of the concrete model. Because of this universal properties (e.g., properties of LTL) that are true in the abstraction are also true in the concrete model.

6.2 Modular iterative abstraction refinement

A feasible abstraction level is found automatically using an iterative abstraction refinement algorithm. The main loop of the algorithm is illustrated in Figure 6.3. The algorithm follows the phases of the generic iterative abstraction refinement loop. The initial abstraction is first generated and model checked. If the examined property produces a counterexample, the algorithm refines the model and verifies the resulting new model again. The process is continued until the property is proved or no further model refinement is possible.

The general intention of the algorithm is to begin with as much abstraction as possible, and then iteratively add modules to the configuration

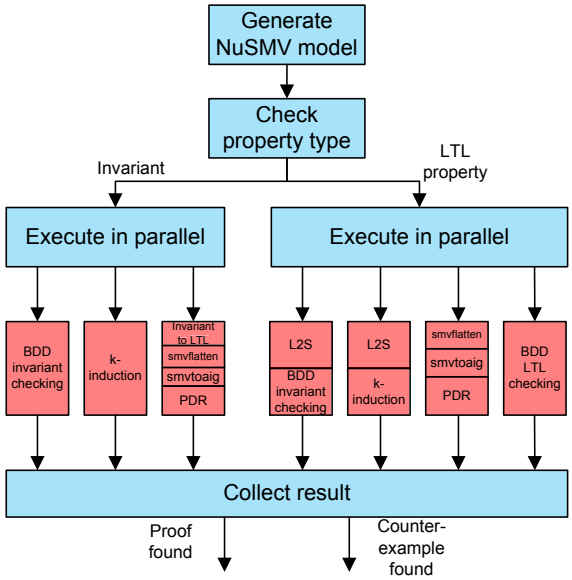


Figure 6.4. The model-checking phase of the technique

until the abstraction satisfies the property. For refinement, the algorithm follows a two-phase procedure. First, the algorithm creates a preliminary refinement by iteratively adding new modules to the abstraction, until the counterexample becomes infeasible. Then the algorithm minimises the refinement by sampling subsets of the preliminary refinement and checks the feasibility of the samples. In what follows, the individual steps of the algorithm are described in more detail.

6.2.1 Initial abstraction

The initial abstraction is created by extracting the variables used in the examined specification and determining the modules whose outputs these variables are. Concrete versions of these modules are used in the initial abstraction. All other modules of the model are replaced by their respective interface modules.

6.2.2 Model checking

In the model-checking phase several algorithms are utilised in parallel as illustrated in Figure 6.4. The implementation of the algorithm supports both state invariant properties and LTL properties.

In the case of state invariant properties the BDD-based algorithm, and the k-induction algorithm are run using NuSMV, while the property di-

rected reachability (PDR) algorithm is run using a model-checking tool called ABC/ZZ [80].

The ABC/ZZ tool requires models to be in the AIGER [28] format. In order to be able to use the PDR algorithm implemented in the ABC/ZZ tool, the NuSMV model has to be transformed into AIGER models. A suitable transformation tool called `smvtoaig` is available in the AIGER tool package. The tool requires that the model has been flattened beforehand. The flattening feature implemented in NuSMV is utilised by running the following commands in NuSMV: `read_model`, `flatten_hierarchy`, `encode_variables`, `build_boolean_model`, `write_boolean_model`. This command sequence creates a deterministic model with no modular hierarchy, in which integers are encoded using Boolean variables. The `smvtoaig` tool can translate this flattened model into the AIGER format but the translation does not support all modelling features of NuSMV. The most conventional modelling conventions used in Appendix A, for example, are supported.

Finally, the examined state invariant is transformed into an LTL property so that the model together with the LTL property translates correctly into an AIGER format model that can be model checked by the PDR model checker. The LTL formula is created simply by adding the LTL globally operator G before the state invariant.

When LTL properties are verified the algorithm first utilises the liveness-to-safety reduction described in Section 2.8.2 after which it runs BDD-based state invariant checking and the k-induction algorithm on the resulting model. For PDR model checking, the algorithm follows a similar procedure as in the case of state invariant properties, except that the property is already written in LTL and need not be changed. For LTL properties the BDD-based LTL checking algorithm is additionally run in parallel using NuSMV.

6.2.3 Preliminary refinement

In the refinement phase the objective is to find a new abstraction (i.e., a configuration of concrete modules and interface modules) that is more detailed than the current configuration of the model and makes the current counterexample infeasible. The original specification is then checked again on that refined abstraction.

The preliminary refinement phase itself is an iterative loop, in which each iteration adds new modules to the model configuration. The loop

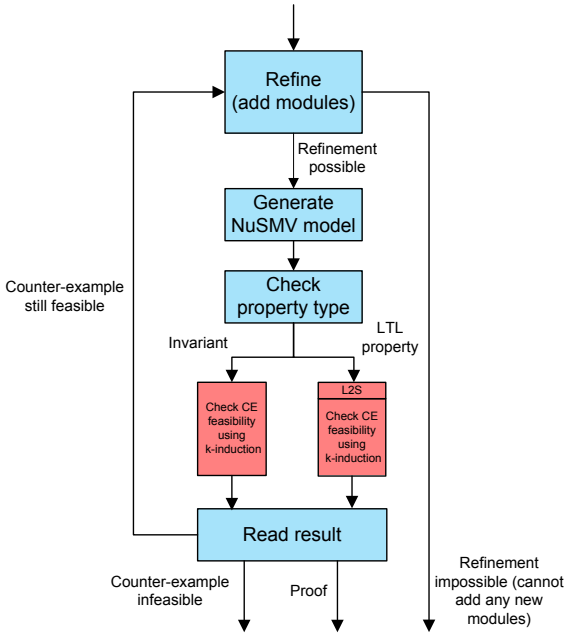


Figure 6.5. The preliminary refinement phase of the algorithm

is illustrated in Figure 6.5. Similarly, as in the model-checking phase of the algorithm, both state invariant properties and LTL properties are supported. In the case of LTL properties the liveness-to-safety reduction is again utilised to generate a model on which the k-induction algorithm can be used.

The algorithm uses the dependency graph of the model to identify new modules to be added to the model.

Definition 6.1. *A dependency graph of a system divided into modules* $v \in V$ *is a directed graph* $D = (V, E)$ *where the vertices* V *represent the modules and the edges represent the data dependencies of modules towards each other. A module* v *can be described with respect to its inputs and outputs as* $v = (I_v, O_v)$*, where* I_v *is the set of input signals of* v*, and* O_v *is the set of output signals of* v*. The transitions of the dependency graph are then defined so that:* $(t, v) \in E \Leftrightarrow \exists x$ *such that* $x \in O_t$ *and* $x \in I_v$ *where* $t = (I_t, O_t), v = (I_v, O_v) \in V$.

Note that in our definition of the dependency graph, the edges of the graph are in the direction of the data flow.

On each preliminary refinement iteration the algorithm the dependency graph of the model is traversed one step in the backwards direction starting from all the vertices representing the modules in the current abstrac-

tion, and all of the modules representing these neighbour vertices are added to the abstraction.

The preliminary refinement phase is presented as pseudo-code in Algorithm 6.2. The function *RefineConfiguration* has as input the set *Current* of modules that are concrete in the current abstraction, and the length of the recently received counterexample *CElength*. The algorithm returns the set of new modules *Refinement* that are added to the current model, and a string indicating whether no further refinement is possible or if a proof is found while checking the feasibility of the refinement.

Algorithm 6.2 Preliminary refinement

```
1:  procedure REFINECONFIGURATION(Current, CElength)
2:      Configuration ← Current
3:      Refinement ← ∅
4:      while True do
5:          newRefinement ← ∅
6:          for e ∈ getNeighbourModules(Configuration) do
7:              if e ∉ Configuration then
8:                  newRefinement ← newRefinement ∪ e
9:              end if
10:         end for
11:         if len(newRefinement) == 0 then
12:             return [Refinement, "no refinement"]
13:         end if
14:         Configuration ← Configuration ∪ newRefinement
15:         Refinement ← Refinement ∪ newRefinement
16:         CEinfeasible, proved ← checkFeasibility(Configuration, CElength)
17:         if CEinfeasible then
18:             if proved then return [Refinement ,"proof"]
19:             end if
20:             break          // R efinement found
21:         end if
22:     end while
23:     return [Refinement, ""]
24: end procedure
```

The function *getNeighbourModules* traverses the dependency graph one step in the backwards direction starting from all the vertices representing the modules in the current abstraction, and returns all of the modules

representing these neighbour vertices.

The algorithm generates a new model configuration *Configuration* in which the modules in the refinement are concrete, and all other modules are interface modules. If all of the relevant edges of the dependency graph have been traversed, and no new modules can be added, the property is false in the concrete model. The counterexample that was generated in a previous step is the final counterexample. Note that these counterexamples may have some abstract modules that are not capable of influencing the variables in the property.

When refinement candidates are being created, the algorithm checks the feasibility of the previous counterexample using the *checkFeasibility* function. The function checks the original specification using k-induction with the bound k set to the length of the counterexample, and thus the feasibility of all other counter-examples up to this length is also simultaneously checked. It is possible for longer counterexample traces to exist that are feasible at the same abstraction. These counterexamples will be eventually found in the model checking phase of subsequent iterations of the algorithm, as the abstraction-level has been refined so that the simpler counterexamples have become infeasible. Note that also classic BMC could be used but k-induction has the added benefit of occasionally proving the property during a feasibility check. If a counterexample can be found within the bound, the refinement has not been successful, and the algorithm continues with the preliminary refinement phase. If no counterexamples can be found within the bound, the refinement has been successful, and the algorithm moves on to refinement minimisation.

6.2.4 Refinement minimisation

After the first suitable refinement has been found, refinement minimisation is begun. The minimisation algorithm is represented as pseudo-code in Algorithm 6.3.

The function *Minimisation* has as input the set *Current* of modules that are concrete in the current abstraction, the length of the recently received counterexample *CElength*, and the set of modules *Refinement* in the preliminary refinement. The algorithm returns a new set of modules that includes all modules in *Current* and a locally minimal set of modules in *Refinement* that suffice to make the refinement feasible, and a string indicating if the property could be proved during the feasibility check.

The algorithm samples subsets of the modules in the preliminary refine-

Algorithm 6.3 Refinement minimisation

```
1:  procedure MINIMISATION(Current, CElength, Refinement)
2:      n ← 2
3:      NewRefinement ← Refinement
4:      while True do
5:          if len(NewRefinement) < 2 then
6:              return [Current ∪ NewRefinement, ""]
7:          end if
8:          subsets ← partitionSet(NewRefinement, n)
9:          complements ← getComplements(subsets, NewRefinement)
10:         CEinfeasible ← False
11:         for c ∈ subsets ∪ complements do
12:             Configuration ← Current ∪ c
13:             CEinfeasible, proved ← checkFeasibility(Configuration, CElength)
14:             if CEinfeasible then
15:                 if proved then return [Configuration, "proof"]
16:                 end if
17:                 if c ∈ subsets then
18:                     n ← 2          // c ∈ subsets
19:                 else
20:                     n ← max(n − 1, 2)          // c ∈ complements
21:                 end if
22:                 NewRefinement ← c
23:                 break
24:             end if
25:         end for
26:         if CEinfeasible then continue
27:         end if
28:         if n < len(NewRefinement) then
29:             n ← min(len(NewRefinement), 2n)
30:             continue
31:         else
32:             return [Current ∪ NewRefinement, ""]
33:         end if
34:     end while
35: end procedure
```

ment in an iterative manner, and checks the feasibility of the samples similarly as in preliminary refinement using the function $checkFeasibility$. If a suitable subset of modules is found, the algorithm restarts the minimisation procedure using the found subset as a starting point. The approach leads to a locally minimal subset of modules. Note that the refinement approach is cumulative, since the minimisation is applied only to the new modules in the preliminary refinement. The modules from previous iterations cannot be removed in the minimisation.

The subset sampling is based on the delta debugging technique for software code described in [216] and [217]. The purpose of the original technique is to generate a simple test case that captures the variable assignments that cause a particular failure. In delta debugging based refinement minimisation, the set of modules is first partitioned into two parts (the initial granularity is two), and refinements based on both sets are checked. If the minimisation is not successful, the granularity is increased, and the set of modules is divided into four parts. After this these four sets and their complement sets are checked. If none of these subsets is suitable, granularity is again increased. The process continues until the granularity reaches the size of the module set.

The function $partitionSet(set, n)$ partitions a set into n parts. For example, $partitionSet([1,2,3,4,5,6],3)$ returns $[[1,2],[3,4],[5,6]]$. The function $getComplements(partitions, set)$ is used to produce the complements of these sets. $getComplements([[1,2],[3,4],[5,6]],[1,2,3,4,5,6])$ returns the set of sets $[[3,4,5,6],[1,2,5,6],[1,2,3,4]]$.

The iterative structure of the refinement minimisation phase is illustrated in Figure 6.6. The liveness-to-safety reduction is again used to support the feasiblity checking of LTL properties.

6.2.5 Correctness of the algorithm

The soundness, completeness and termination of the developed technique are addressed in this section. The procedure related to the iterative abstraction refinement algorithm is depicted in Figure 6.7, where the transitions are numbered.

Proposition 6.1. *The iterative abstraction refinement algorithm always terminates.*

Proof sketch. As the models we consider are finite state, for the termination proof we will assume that the model checking phase of the algorithm

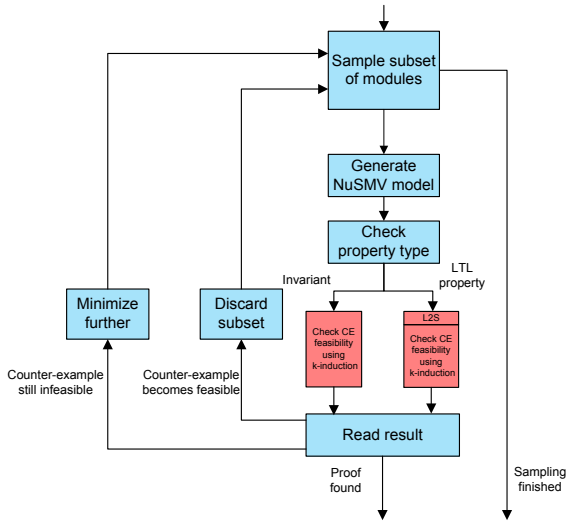


Figure 6.6. Refinement minimisation

always terminates. The preliminary refinement phase iteratively adds modules based on the modules' dependencies. It can not do this ad infinitum as the system model has a finite number of modules. Therefore, the preliminary refinement phase also always terminates. The refinement minimisation phase performs a finite number of iterations by sampling subsets of the modules in the preliminary refinement. The number of modules in the preliminary refinement is finite. Therefore, the number of subsets is also finite, and the refinement minimisation phase must eventually terminate.

Regarding delta debugging used in the refinement minimisation phase, we assume that it terminates and leaves at least one module provided by the preliminary refinement phase. The number of iterations of delta debugging in the worst case is addressed by Zeller and Hildebrandt (see Proposition 17 in [217]). Delta debugging leads to a locally minimal subset of modules (see Proposition 16 in [217]).

Now, assume that the algorithm does not terminate. Since individual phases of the algorithm terminate, this is possible only if the transitions numbered 3, 6 and 8 in Figure 6.7, are infinitely looped. Transition 3 is associated with a counterexample, which becomes infeasible in transition 6 as the model is refined by adding new concrete modules to the model. On the other hand, if no new modules are added in preliminary refinement, transition 4 should have been taken. Also, transition 6 can not be taken if no new modules have been added because the counterexample would then

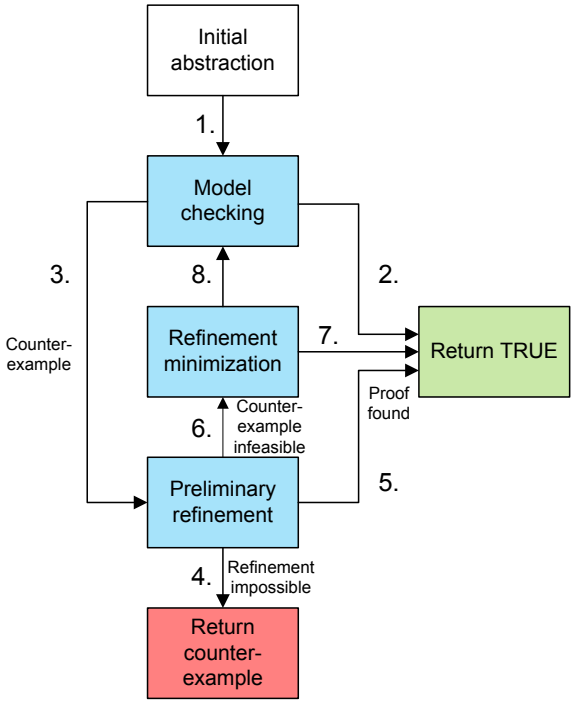


Figure 6.7. The iterative abstraction refinement loop with numbered transitions

be feasible as the model is equal to the model in transition 3. In refinement minimisation, modules are removed from the refinement. All modules can not be removed because the counterexample would then again become feasible. Therefore, a single iteration of the algorithm (transitions 3, 6, and 8) necessarily increases the number of concrete modules in the model. The model has a finite number of modules. Therefore, within a finite number of iterations, the algorithm must eventually be in the model checking phase with a model configuration in which no new modules can be added. On this model configuration the examined property is either true or false. Therefore, either transition 2 or transitions 3 and 4 are eventually taken terminating the algorithm.

□

Proposition 6.2. *If the iterative abstraction refinement algorithm returns an answer, the answer is correct.*

Proof sketch. First, assume that the algorithm returns "$True$". The answer is correct because all created abstractions are over-approximating abstractions, in which a set of modules of the model is replaced with interface modules. All abstractions have more behaviours than the concrete

model. If the examined property is true on any such abstraction, the property is also true on the concrete system model.

Now assume that the algorithm returns a counterexample. A counterexample is only given when no further refinement is possible by adding concrete modules to the model based on the modules' dependencies. The initial abstraction starts with modules immediately related to the examined property. From this starting point, if a model configuration is reached in which no new modules can be added, it necessarily corresponds to the result of applying the cone-of-inclusion (COI) reduction [65] on the module level. The counter-example is valid because the modules excluded in the model are not relevant to the property under verification.

□

Corollary 6.1. *By Propositions 6.1 and 6.2 the iterative abstraction refinement algorithm terminates and returns a correct answer for any input.*

6.3 Results of tests

In Publication III, the iterative abstraction refinement technique was used to verify properties on two different models. First, a fictive but realistic system[2] was used for demonstrating that the algorithm can be more efficient in proving meaningful safety properties of a system. Secondly, the technique was evaluated by using a real-world diesel control system as a benchmark, and comparing the technique with other model-checking algorithms. Since the technique focuses on finding proofs efficiently, only true state invariants were used in this comparison. A set of 100 randomly generated true state invariant specifications were verified on the diesel model using four approaches: our technique, k-induction, property directed reachability (PDR), and a BDD-based state invariant checking method. The BDD approach, the k-induction approach, and the PDR approach used the full concrete model. For simplicity, the comparison focused on state invariant specifications, and no LTL properties were used. In the comparison, the commands used for model checking were as follows. The BDD approach used the `check_invar` command of NuSMV. The NuSMV flags `-dynamic` and `-coi` were also used. The k-induction approach was run by executing the command `check_invar_bmc_inc`

[2]The model of the fictive system and its detailed description is available online: `https://github.com/JussiLahtinen/Dissertation`

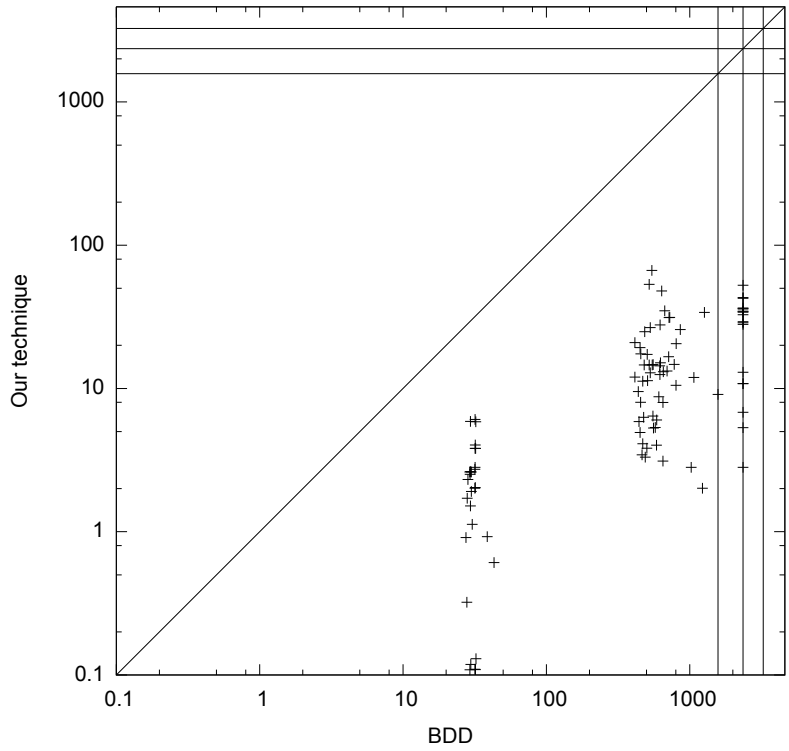


Figure 6.8. Run-times (in seconds) of BDD model checking and our technique

`-k 10000` in NuSMV. A large bound was used because running out of memory is preferred to not reporting a result. The NuSMV flags `-dynamic` and `-coi` were used for k-induction as well. The command used for running the PDR algorithm of ABC/ZZ was: `bip ,live -k=inc -eng=pdr2`.

The random state invariants were generated by randomly selecting two to three variables from the set of modules' outputs and non-deterministic variables of the main module. Negations were randomly placed in front of the chosen variables, and the operators joining the chosen variables were randomly selected (AND/OR). After this the random state invariant was model checked using the different model-checking methods until each state invariant was proved either true or false. Finally, only the true state invariants were picked out for the results. The invariants were also manually reviewed to ensure that all invariants were unique. The following is an example of a random state invariant property:

```
((not MOD14.output1 OR MOD2.output2) AND
not main_module_input1)
```

The results of the comparison are in Figures 6.8, 6.9 and 6.10.

The k-induction algorithm was able to verify 55% of the properties rather

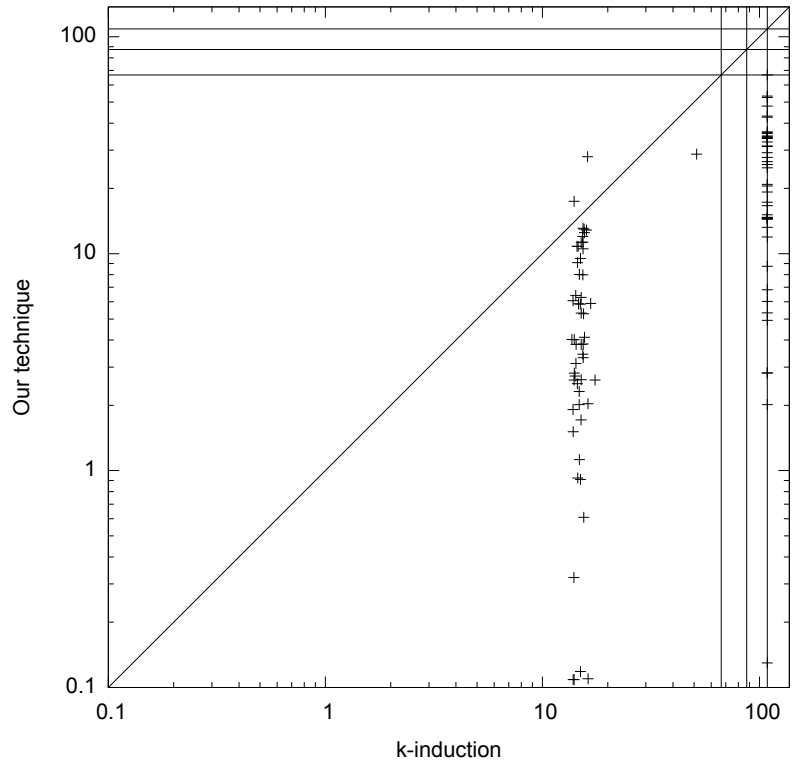


Figure 6.9. Run-times (in seconds) of k-induction and our technique

quickly, but for the rest of the properties (45%) the required bound for a proof was so large that the technique ran out of memory. The BDD-based method was able verify 81% of the properties within the 1800 second timeout. However, 13% of the generated random state invariants were such that neither the BDD-based technique nor the k-induction technique could verify them within the given resource limits. The PDR technique was able to solve all properties within the given resources. The Figure 6.10 does not show the time to generate the AIGER format model from the NuSMV model. On average this time was 17 s. The model transformation time is, however, included in the PDR verification runs performed as part of our technique since these model transformations depend on the model configuration and cannot be calculated beforehand.

Figure 6.9 contains a distinct vertical grouping of data points. The set of points indicates that many properties have been solved in approximately 15 seconds by the k-induction method. This is presumably because a certain amount of time is needed for building and initialising of the model, after which the properties may be quickly proved within a small bound. A similar phenomenon is apparent in Figures 6.8 and 6.10 as well. The

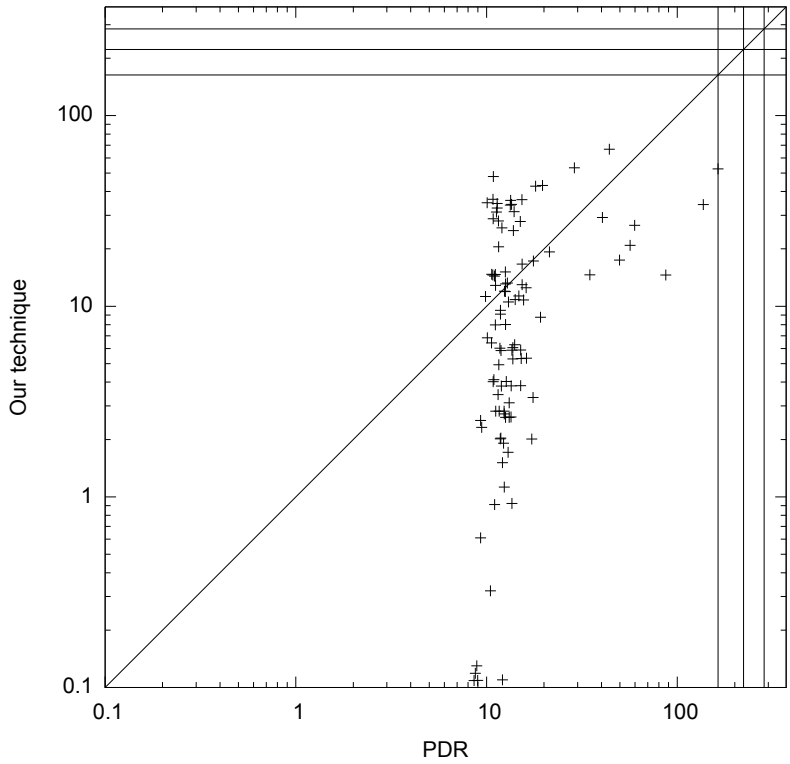


Figure 6.10. Run-times (in seconds) of PDR and our technique

preprocessing costs of the algorithms could be the reason why our technique outperforms them, as the preprocessing costs are low on smaller models used by our technique. The effect of the preprocessing costs to our technique could become more significant if even large models were considered.

Our technique could verify all of the properties within the given resources. In cases where the other techniques were able to provide a proof, our technique was typically faster. In 68% of the cases the properties were such that the proof could be found faster using our technique than by using the other three algorithms. In 1% of the cases, the k-induction technique was the fastest. The PDR-based technique was the fastest in 31% of the cases.

When our technique is analysed in detail, it is noted that 82 properties are proved in the model-checking phase of the algorithm, and 18 properties are proved in the preliminary refinement phase of the algorithm. The distribution of the model-checking phase proofs was such that 32 properties are proved by the k-induction subroutine, 10 properties are proved by the BDD-based subroutine, and 40 of the properties are proved by the

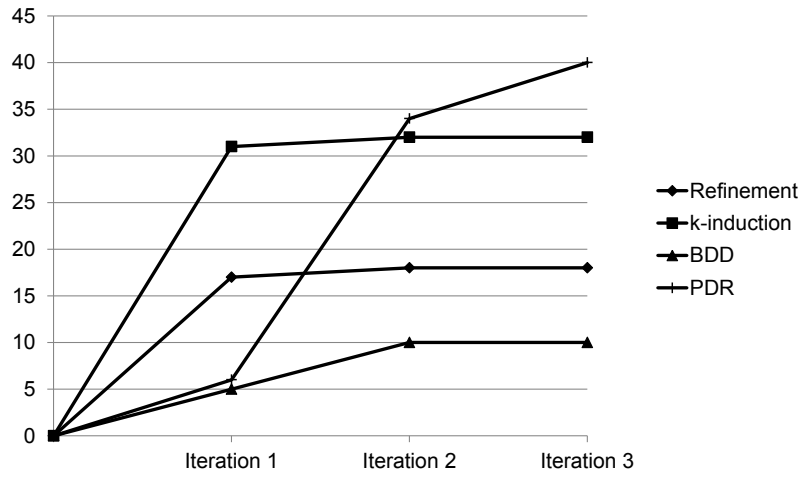


Figure 6.11. The number of proofs found by k-induction, BDD, PDR, and refinement phase k-induction, and the cumulative number of proofs found on different iterations of the algorithm

PDR-based subroutine. The proofs of the preliminary refinement phase are also generated by the k-induction subroutine as only that subroutine is used for checking the feasibility of the refinement. The cumulative frequency graph in Figure 6.11 shows the number of proofs obtained by the different subroutines (k-induction, BDD, PDR, and refinement phase k-induction) and the distribution of these proofs on the different iterations of the algorithm. Three iterations of the algorithm sufficed to prove all properties.

Based on Figure 6.11 it can be observed that the k-induction method handles the most trivial properties in the first iteration (31 properties). For a majority of the remaining properties, the refinement phase results in an abstraction that is detailed enough for a proof by the PDR-based method (34 properties) or the BDD-based method (10 properties) in the second iteration. The average for the number of modules needed in a proof was 3.4 in the system consisting of 14 modules.

In very large models, the initialisation of the model can already take quite long, and finding a proof with a very small bound can still take a lot of time. In the technique, only a small local subsystem is initially examined which is much faster. The results confirm the assumption that in large models, only a small subsystem is typically needed for proving a single property. The results also show that locating such a subsystem, and verifying the property on that subsystem can in most cases be faster than the analysis of the whole system model.

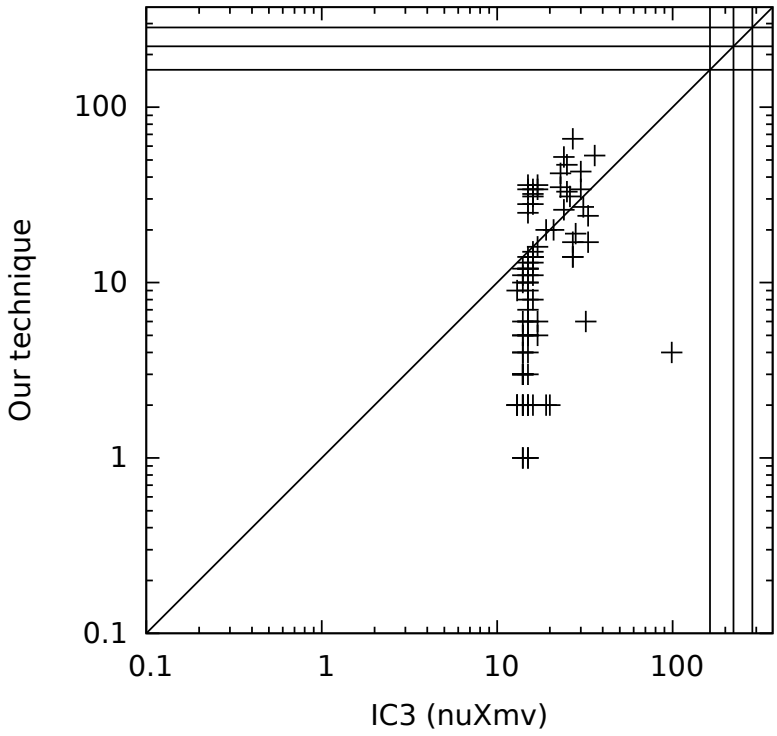


Figure 6.12. Run-times (in seconds) of the IC3 algorithm implemented in nuXmv and our technique

6.3.1 Comparison against the IC3 algorithm implemented in nuXmv

The nuXmv model checker [50] is an extension to NuSMV that has an implementation of the PDR (IC3) algorithm. In this work, however, nuXmv has not been used for verification. This is because one of the objectives of the research was to facilitate verification work in future customer projects. The license conditions of nuXmv prevent free commercial use rendering utilisation of that tool unattractive in this setting. On the other hand, the model-checking tools used in this work (NuSMV and ABC/ZZ) have less restricting open source license conditions.

To get a better picture of the performance of the developed technique, a comparison against the IC3 algorithm implemented in nuXmv was also conducted again using the real-world diesel control system benchmark. The nuXmv command `check_invar_ic3` was run on the full system model in which all of the modules are concrete. The previously generated set of 100 state invariants were utilised also in this comparison. The results of the comparison are in Figure 6.12.

The nuXmv tool could verify all of the properties within the given resources. In 78 % of the cases, our algorithm found a proof faster than the IC3 implementation of nuXmv. The average verification time of nuXmv was 30,2 s; and the median verification time was 15,6 s. The average verification time of our technique was 14,2 s; and the median verification time was 9,3 s. The performance of ABC/ZZ is slightly better when compared to nuXmv. The average verification time of ABC/ZZ was 18,5 s; and the median verification time was 12,6 s. These verification times, however, do not include the time needed for translating the model into the AIGER format.

6.4 Validity of the technique

The performance of the iterative abstraction refinement technique relies heavily on the structure of the model's dependency graph. Systems where there are a lot of dependencies between the modules, can quickly lead to a situation where all modules are included in the model. So far, the technique has been well-suited for safety-critical systems where unnecessary links between subsystems are avoided.

The technique is also more efficient in models that produce short counterexamples. Checking the feasibility of long counterexamples is much more time-consuming as the bound k used for k-induction increases. This is a potential problem in systems where there are, e.g., long time delays in the application logic.

There are also some limitations related to model transformations. In order to use the PDR algorithm implemented in the ABC/ZZ tool the NuSMV models need to be transformed into AIGER models. There can be problems in this transformation when unusual NuSMV modelling features are used in the model. Another problem is mapping the witness trace of ABC/ZZ back into the counterexample format used in NuSMV, which is fairly straight-forward in most cases, but again requires special attention when the more unusual NuSMV features are used.

For LTL properties the liveness-to-safety reduction is used to transform the model into another variant in which the liveness property can be stated as a state invariant. The result of the reduction leads to a model with a much bigger state space. When the reduction is used on a model that is already very large the result can become infeasible.

7. Analysing fault-tolerance of nuclear power plant safety systems

This chapter discusses methodology for modelling hardware failures developed in Publication IV, the motivation for this work, as well as potential integration approaches with probabilistic risk assessment (PRA).

7.1 Background and motivation

The main motivation for this work is the aim of analysing increasingly larger systems. Model checking has turned out to be a useful tool for verifying the correctness of individual systems. Thus, it makes sense to try to increase the scope of verification, and to see whether other problems of larger scale can also be solved. In nuclear power plants, the natural step from individual safety system is to move on to analysing the plant architecture level, i.e., how multiple safety systems work together, and how their functionality is organised. On this higher level, requirements for fault tolerance of the architecture become important, as the architecture is designed in such a way that design errors of individual safety systems do not lead to hazardous situations. A methodology for modelling hardware failures is needed in order to analyse fault-tolerance.

7.1.1 Traditional architecture-level analyses

Analysing the fault-tolerance of the plant can be quite a difficult task due to: (1) complex hardware architecture (a single computer unit may be used to execute the application logic of several different safety systems); (2) common cause failures (CCFs) (failure of multiple similar components due to the same cause); and (3) unpredictable behaviour of the system caused by software design errors.

Traditionally, architecture-level analysis is performed using methods such as failure mode and effects analysis (FMEA) [112], and probabilistic

risk assessment (PRA) [16].

In FMEA, the failure modes of subsystems and components are systematically defined, and the causes and effects of each individual failure mode are analysed. FMEA focuses on single failures, and its capability to analyse CCFs is quite limited. Postulating failures of software components is also possible in FMEA. Typically this is based on a list of presumed effects of these errors.

PRA is a method that is used for estimating different kinds of risks related to the operation of a nuclear power plant. One use of PRA is to estimate the frequency of accidents that cause damage to the reactor core (i.e., level 1 PRA). In this kind of analysis, a set of accidents (initiating events) such as the loss of coolant accident (LOCA) are first defined. Each initiating event is then analysed using event trees and fault trees.

Event trees (see, e.g., [206]) illustrate the consequence of different accident sequences. An example event tree is in Figure 7.1. An accident sequence is a series of successes and failures of safety systems represented as paths of the event tree. If one or more safety systems function properly the plant typically fully recovers from the accident without core damage. If enough safety systems fail, the accident can cause core damage or lead to some other less critical unwanted end state. In Figure 7.1, an accident leads to core damage only when both System 1 and System 2 fail.

In PRA the top events are further analysed using fault trees (see Figure 7.2). In fault tree analysis (FTA) [203] the top event is broken down into simple component failures (basic events) using Boolean logic gates. In FTA, a list of minimal cut sets (MCSs) (see, e.g. [203]) is typically calculated to create an understanding of the system's weak points. A cut set is a combination of failures that causes the top event. A minimal cut set is a set of component failures such that if any of the basic events in the set is removed, the remaining events no longer form a cut set. For example, the fault tree in Figure 7.2 has two minimal cut sets. The first minimal cut set consists of a power supply failure, and the second minimal cut set includes the failure of both actuator 1 and actuator 2. A cut set that consists of two failures: (1) failure of actuator 1; and (2) the failure of power supply is not minimal as the top event (failure of system 1) happens also when the first failure is removed from the set.

By estimating the failure rates of each component used in the plant, the core damage frequency can be estimated. It is also possible to analyse the criticality of the basic events using risk importance metrics such as the

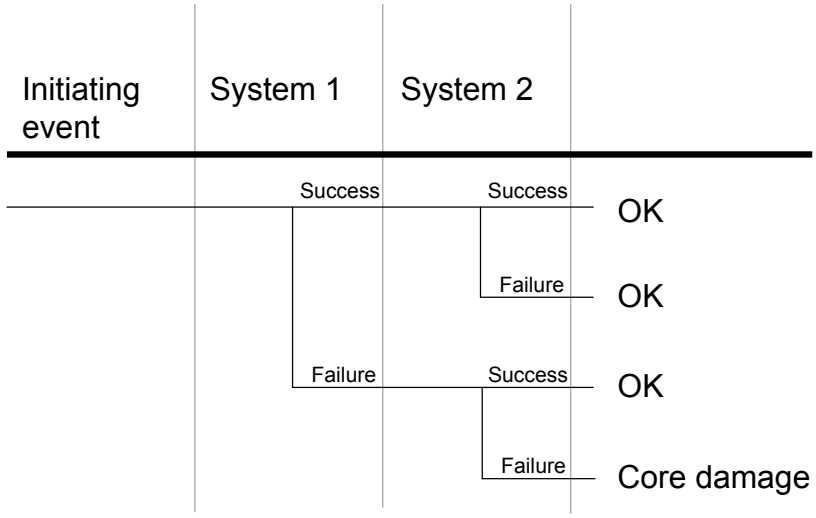


Figure 7.1. An example event tree

Fussell-Vesely measure of importance and the risk increase factor, see, e.g., [201]. In PRA, software errors can also be postulated, e.g., based on a software failure taxonomy (see [8]), but the methodology is not yet mature. Similarly as in FMEA, this approach is based on guessing the probable effects of a software failure.

7.1.2 Using model checking for architecture-level analysis

In FMEA and PRA software errors can be postulated but they are not methods that can find concrete scenarios in which the postulated errors occur. Model checking, on the other hand, can find these concrete scenarios if they exist. If model checking is applied on a system model that takes into consideration both the detailed behaviour of the application software and the hardware failures, it is possible to find new concrete scenarios in which the plant safety function is not fulfilled. These scenarios can be such that they involve both software design errors and hardware failures. Using model checking for analysing architecture-level properties enables a more exact and more extensive approach for analysing fault-tolerance.

7.2 Fault modelling methodology

The hardware failure methodology in Publication IV was developed based on using a PRA model (see, e.g., [9]) of a nuclear power plant as reference. The model-checking methodology purposefully followed the notations and

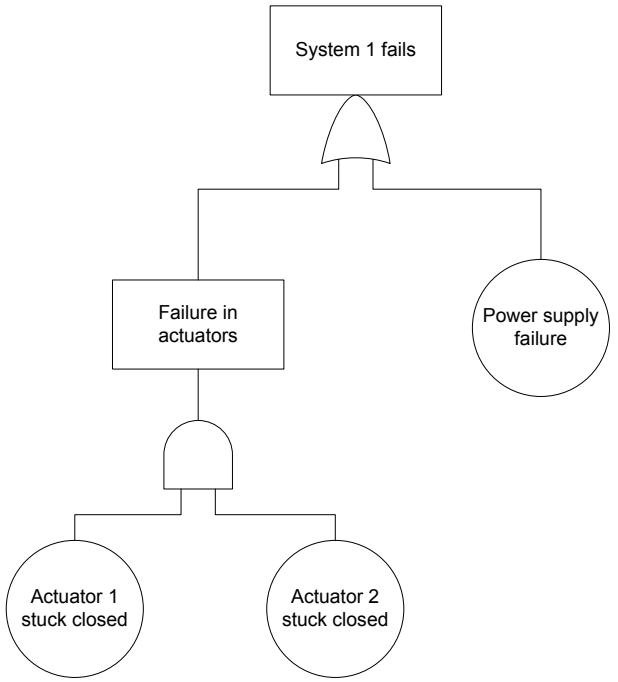


Figure 7.2. An example fault tree

conventions used in the PRA model whenever possible, and was designed to fit seamlessly with earlier methodology for modelling the application programs as described in Chapter 5. The new model components of the proposed methodology are link modules, a failure module and a process module.

The main idea of the methodology is to use link modules to encapsulate connections between measurements, logic modules and equipment. Link modules are parameterised with hardware components that implement that connection. Link modules may alter the signal value, as hardware failures may cause the value of the signal transmitted via that link to be interpreted in a different way. The failure module is used to keep track of all instances of hardware components, and to decide on the failure modes experienced by the components. The process module plays the role of an environment model, and decides on the values of the physical parameters of the plant based on predetermined scenarios.

The model used as a case study in this work included several four-redundant safety systems. A redundant subsystem is also called a division. A typical architecture of a single safety system from the point of view of division 1 is illustrated in Figure 7.3. The system consists of mea-

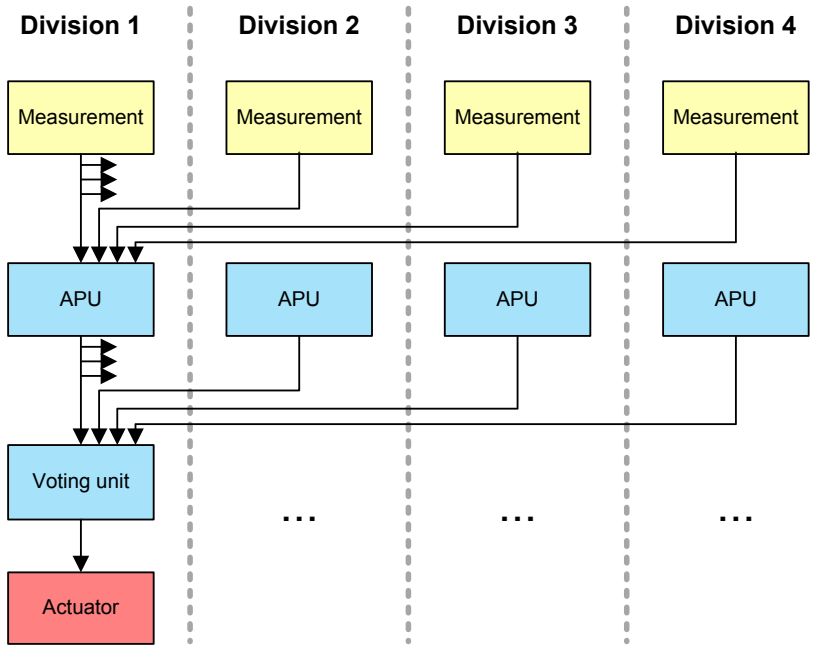
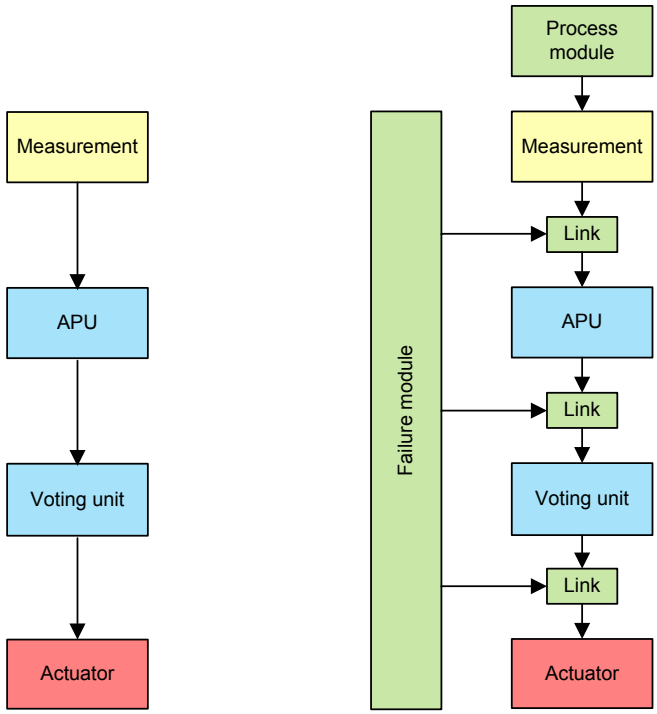


Figure 7.3. Typical I&C system architecture

surements, acquisition and processing units (APUs), voting units, and actuators. APUs are used for deciding on control actions sent to the actuator. Voting units are used for majority voting, so that diverging APU decisions can be ignored.

If the system is interpreted as a data flow graph, it can be seen that all data paths are of the form depicted in Figure 7.4(a). When a single data path is modelled, each of the three connections is encapsulated with a link module, see Figure 7.4(b). The link modules are parameterised with hardware component failure modes received from the failures module. For example, the link between the measurement and the APU is related to information of the failures of the measuring instrument, the connection medium (such as a cable) between the instrument and the APU, the input module of the APU, and failures of the power supply of the APU. If, e.g., the measuring device experiences an undetected failure in which the measured value freezes, the link module changes the value of its output accordingly. Figure 7.4(b) also illustrates how the measurements receive values from the process module. If a measuring device experiences no failures, its value will follow the physical parameters of the process module.

In Publication IV, the fulfilment of success criteria under various fault assumptions is analysed on the case study model. It should be noted that



(a) Data path of the system (b) Modelled data path

Figure 7.4. Modelling idea illustrated on a single data path

the analysis is possible mainly due to simplifications and abstractions used in the model. Most importantly, the exclusion of timing aspects in the case study significantly simplifies the state space of the model, making model checking of the case study model more feasible. The actuation logic used in realistic systems does use a lot of timing, and its analysis without these timing aspects is only of a very limited use. Even without these timing aspects, the case study model was very large. So large in fact, that the model checker failed to calculate the full state space, and the size of the state space could only be calculated for a simpler model including three of the safety systems used in the model. This simpler model consisted of 1.1×10^{260} different states out of which 1.9×10^{160} states were reachable.

The conclusion of this case study is that the methodology is not directly applicable to the verification of properties on the architecture-level simply because the model becomes too large. The systematic approach for hardware failure modelling provides a functioning framework for such

analyses, but other measures still need to be taken to simplify the size of the verification problem. One possibility is simply to use the methodology for analysing fault-tolerance properties of smaller system assemblies. It could also be possible to use an iterative abstraction refinement technique similar to the one described in Chapter 6 in order to obtain a smaller configuration of the model that is still within capacity limits of the verification tool. Another possibility is to limit the extent in which the hardware failures are examined by focusing only on the most important failure modes. This approach is discussed in Section 7.3.1.

Detailed descriptions of the model, the abstractions used, and the verification results can be found in a research report [129]. The models related to this work are available online.[1]

7.2.1 Limitations

A major limitation is that the verification of plant-level models is only possible when heavy simplifications and abstractions are made. Another limitation is that the methodology is based on the assumption that observable signal values can always be deduced from the experienced hardware failures. In some cases this can be very difficult. If several closely related hardware failures occur at the same time, the model should be able to decide which failure dominates over the other one. For example, if a failure occurs in both the input module of a computer and its power source, the failure of the power source is dominating because it masks the other failure. The possible propagation of failures related to, e.g., network failures was also not addressed in the developed methodology.

7.3 Integration of PRA and model checking

The developed modelling approach is compatible with PRA methodology, and may offer an opportunity to integrate the two approaches to produce a more extensive safety analysis method.

As an example, as the same data can be used as input for both approaches, it would be beneficial to have a common plant-model for both analysis methods, accompanied with tools capable of generating both the PRA model and the model used for model checking. The plant model could be expressed using a domain specific modelling language, and the model

[1] https://github.com/JussiLahtinen/Dissertation

could contain all information needed for building both models. The main advantage of using a common plant model is that the efforts needed for modelling and version control between the two models are reduced. A common plant model could also enable cross-verification of the models. For example, the correctness of calculating the minimal cut sets in the PRA tool could be independently verified using model checking.

7.3.1 A concept-level approach for coupled use of PRA and model checking

The two approaches could also be used together in order to make up for each other's weaknesses, and to provide new safety assessment capabilities. PRA is an efficient method for analysing hardware failures but it does not have mature methodology for handling software failures. In model checking, the situation is the opposite. Pure software properties can be analysed in realistic-sized models, but the method does not seem to scale well to large systems when hardware failures are assumed. Neither approach on its own is sufficient for analysing the middle ground: finding concrete scenarios involving both software design errors and hardware failures. The middle ground could, however, be tackled by an approach using both analysis methods in a coupled manner.

A concept-level approach for coupled use of PRA and model checking is illustrated in Figure 7.5. The approach is based on two ideas:

1. Using information received from PRA to restrict the model-checking analysis only to certain hardware failures, making the model-checking phase more scalable to large models
2. Verification of a single hardware failure combination at a time.

Some hardware failures are not as important to safety as others. PRA can provide information on the importance of the component failures. This can be done either by using risk importance measures such as the Fussell-Vesely measure of importance and the risk increase factor, or by exploiting the minimal cut sets calculated in PRA, as is shown in Figure 7.5. Basic events in short minimal cut sets are more critical than basic events in longer minimal cut sets. This is because short minimal cut sets describe scenarios in which only a small number of component failures is able to cause the top event (failure of the safety system). In the proposed approach, model checking focuses only on analysing failures that belong to a

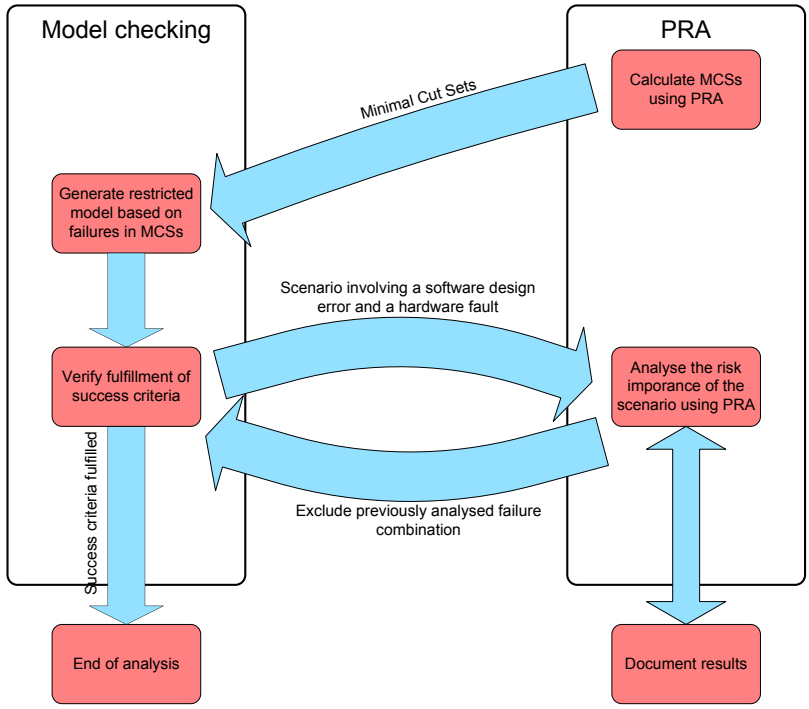


Figure 7.5. An integration approach for PRA and model checking

short (e.g., 1-3 basic events) minimal cut set. Subsets of these short minimal cut sets are interesting since a software design error during such a failure combination might cause the system to fail in a previously unknown manner.

Once the set of most important failures is selected, all relevant failure combinations (e.g., subsets of minimal cut sets) are gone through one by one.

Finally, if a new scenario is found using model checking, it should be analysed whether the scenario describes a previously unknown failure of the safety system. If this is the case, the risk importance of the new scenario can be analysed using PRA.

The proposed concept extends safety assessment to an area that the individual approaches themselves cannot reach. A more detailed description of the proposed technique and results of experimenting with it using a small case study model is available in a research report [132]. The developed technique is based on earlier work by Björkman et al., see [32]. The author of this dissertation has developed the proposed technique together with Björkman.

8. Using model checking for structure-based testing of FBD models

This chapter discusses the contents of Publication V, and the utilisation of model checking to support testing of systems designed using FBDs. In Publication V, a method was developed that utilises model checking and the structure of the FBDs, and automatically generates an efficient test suite with high structural coverage.

Publication V is the first paper written on the subject by the author of this dissertation. Consequently, the technique described in the paper still requires additional research into improving the proposed methodology. Main issues related to the technique have already been identified in the paper, and they are also discussed in Section 8.3. Further development related to the test generation technique can be found in a research report, see [130]. The report presents an alternative version of the test generation technique for FBDs, and evaluates the capability of generated test suites to detect errors using mutation analysis.

8.1 Motivation

Safety systems used in nuclear power plants are thoroughly verified using a vast spectrum of different tests. The ISO/IEC 29119-4 [114] standard defines three different categories of test techniques for software: specification-based, structure-based, and experience-based testing. In the verification of nuclear automation, specification-based testing is primarily used, and structure-based testing is used to increase the coverage of verification.

In specification-based testing the test cases are derived from the requirement specification of the system, while structure-based tests are derived from the structure of the system. The use of both testing techniques is required in nuclear regulatory guides, and recommended in commonly

used standards. For example, the USNRC Regulatory Guide 1.171 [197] requires that testing of safety system software is based on both the specifications of the system, and a structure-based coverage metric. In addition, the generic standard IEC-61508-3 [113] for programmable safety-related systems recommends that structure-based testing is applied to software systems in order to achieve 100% coverage according to several structure-based criteria.

Traditional model checking focuses on verifying the correctness of the system according to a set of formalised properties. The requirement specification of the system and consequently the properties used for model checking may be incomplete. When test cases are generated according to structure-based criteria, however, test sequences may emerge that describe behaviour that have not been fully addressed in the requirement specification. Structure-based testing is a complementary approach to traditional model checking that can be used to identify deficiencies in the requirement specification, and improve the coverage of the analysed properties.

As was explained in Chapter 3, application-level control software is typically designed using FBDs, and then automatically translated into software code. Structure-based testing in this framework could focus either on the structure of the FBDs or the structure of the generated code. In this work the FBDs, and not the software code are utilised. The reason for this is that test cases based on the structure of generated code can be non-intuitive and difficult to understand by humans. Also, as was noted in [183], structure-based testing of automatically generated code is not effective in detecting function block level defects in FBDs.

In order to apply structure-based testing, a metric is needed that can be used to calculate the coverage of the tests. Many structure-based test metrics have been defined for software code, see [114]. For example, statement coverage, branch coverage, and path coverage are widely used. Statement coverage in software is achieved when each line of the program is executed in one of the executed test cases. Branch coverage is achieved when each branch of the control flow (e.g., both the then-branch and the else-branch of an if-then-else statement) is covered in the tests. Path coverage is achieved when each execution path of the program is explored in one of the test cases.

There are also coverage metrics based on data flow (see, e.g., [173]) that focus on events related to the definition and use of variables and data ob-

jects. A definition of a variable is a statement that assigns a value to it, and a use of a variable is a statement that uses a value of it. As an example, the All-DU-paths (all define-use paths) coverage metric requires that all definition-free sub-paths of the control flow graph from each definition of a variable to every use of that definition are covered by a test.

Another quite rigorous test metric for software code is modified condition / decision coverage (MC/DC) [55]. MC/DC is commonly used in avionics for measuring test coverage of safety-critical software. The use of the metric is also recommended in the generic IEC 61508-3 standard [113]. MC/DC is based on the concepts of *condition* and *decision*. A condition is a Boolean valued expression that cannot be broken down into simpler Boolean expressions, e.g., $x > 1$. A decision is a Boolean expression that consists of conditions and Boolean operators, e.g., $(x > 1) \vee (y = 1) \wedge z$.

In order to achieve full MC/DC coverage, the test cases shall fulfil the following requirements:

1. Each entry and exit point in the program is covered.
2. Each condition takes every possible outcome.
3. Each decision in the program takes every possible outcome.
4. Each condition in a decision has been shown to independently affect the outcome of that decision.

The fourth item in the list is fulfilled when it is shown that changing the value of a selected condition also changes the value of the whole decision while all other condition values in that decision retain their value. Two test cases that show such an effect are called an independence pair.

```
1 if (A && B) || C) then
2 {
3 // omitted code
4 }
5 else
6 {
7 // omitted code
8 }
```

Listing 8.1. Example pseudocode extract

For an example of MC/DC coverage, consider the pseudocode extract in Listing 8.1 that consists of a single *if-then-else* structure. A set of test cases fulfilling the requirements for MC/DC coverage is shown in Table 8.1. The MC/DC requirement of covering each entry and exit point

Table 8.1. Test cases that achieve MC/DC coverage on the example pseudocode extract. Values of A, B, C and the result of the decision `((A && B) || C)` are shown.

test number	A	B	C	Decision
1	false	false	false	false
2	false	false	true	true
3	true	true	false	true
4	false	true	false	false
5	true	false	false	false

(MC/DC requirement 1) is trivially covered in this example. In the table, every condition (`A`, `B`, `C`) takes both possible outcomes (true and false) at least once fulfilling MC/DC requirement 2. MC/DC requirement 3 is also fulfilled because the decision `((A && B) || C)` evaluates to true in tests 2 and 3, and false in tests 1, 4 and 5. Finally, MC/DC requirement 4 is fulfilled by three independence pairs. Tests 1 and 2 form an independence pair for condition `C`; tests 3 and 4 form an independence pair for condition `A`; and tests 3 and 5 form an independence pair for condition `B`.

All of the above metrics are control-flow based meaning that they focus on the order of instructions that are executed during a test sequence. They are thus not directly applicable to FBDs. Even though the execution order of function blocks can be explicitly defined there is no control flow similar to software code in FBDs. Instead, all function blocks read inputs and produce outputs on every clock cycle.

Hardware testing (see, e.g., [196]) uses many of the same coverage metrics that are used in the context of software for designing tests based on the HDL (hardware description language) description of the system. Hardware testing also uses the concept of circuit coverage which includes both toggle coverage, and latch coverage. Toggle coverage means that each input and output of the circuit have both values in the tests. Latch coverage requires that the latches of the circuit are both on an off during the tests. While these coverage metrics are more applicable to FBDs, they are quite simple.

Fortunately, a few coverage metrics have been designed for the purpose of measuring test coverage in FBDs. The ones developed by Jee et al. [120, 118] are used in Publication V. Related similar test coverage metrics used by other authors are discussed in Section 4.6. The metrics by Jee et al. interpret an FBD as a data flow graph, and focus on different paths from the inputs to the outputs of that graph. Figure 8.1 highlights using

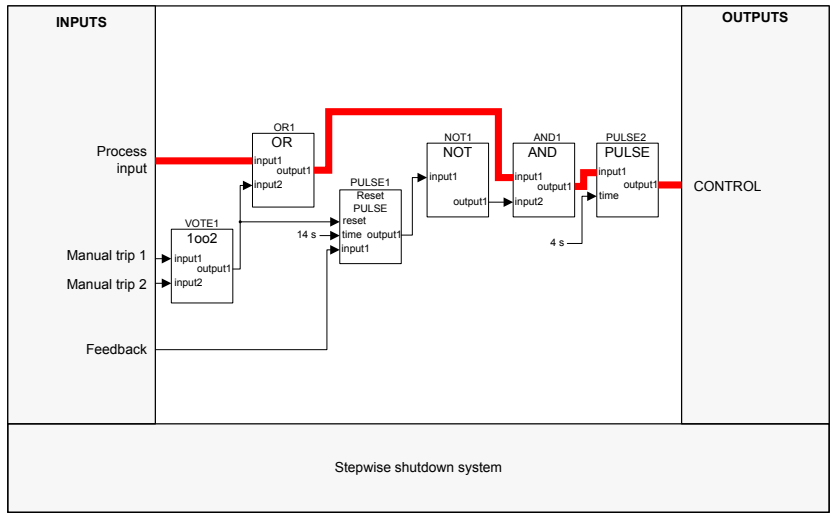


Figure 8.1. A single data path of a function block diagram

red colour one of the data paths of an example system used in Publication V. The data path starts from an input called *Process input*, goes through three function blocks (OR, AND, PULSE), and ends at an output called *CONTROL*.

Jee et al. noticed that it is possible to write a propositional formula that evaluates to true whenever a certain input of a function block has influence on the output. This formula is called a function block condition (FBC)[1]. For example, consider the AND function block in Figure 8.1. It has two inputs, called *input1* and *input2*, and a single output *output1*. According to Jee et al. the condition under which *input1* affects the value of *output1* is $FBC_{AND}(\langle input_1, output1\rangle) = \neg input1 \vee input2$. This is because *input1* can force the value of the output if it is false, and it can make the output value true only if *input2* is also true.

When these conditions have been written for all input-output pairs of every function block type, a similar condition can also be written for a data path by conjoining the function block conditions within that path. An input of a data path affects the output whenever all the function block conditions within that path evaluate to true simultaneously.

In Figure 8.1, the input *Process input* affects the value of *CONTROL* when the FBCs $FBC_{OR}(\langle input_1, output1\rangle)$, $FBC_{AND}(\langle input_1, output1\rangle)$, and

[1]Jee et al. make a distinction between a function condition (FC) and a function block condition (FBC). In their terminology, FBC is used in the context of complex function blocks with, e.g., internal memories. Otherwise, the term FC is used. In this chapter the term function block condition (FBC) is used to refer to all input-output condition formulas.

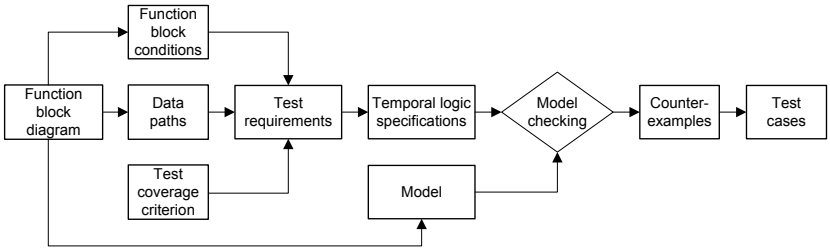


Figure 8.2. Test generation using model checking

$FBC_{PULSE}(\langle input_1, output1 \rangle)$ are all true at the same time. A conjunction of these formulas is called a data path condition (DPC).

Jee et al. have defined three different coverage metrics all based on examining the DPCs. The Basic Coverage criterion (BC) is met when each DPC is fulfilled by one of the test cases. The input condition coverage (ICC) criterion requires that for each Boolean input of a data path, there is a test case in which: (1) the DPC is fulfilled and that input is false; (2) the DPC is fulfilled and that input is true. The complex condition coverage (CCC) criterion requires that for each Boolean signal within a data path, there is a test case in which: (1) the DPC is fulfilled and that signal is false; (2) the DPC is fulfilled and that signal is true.

8.2 Test generation using model checking

The test generation technique developed in Publication V is based on the test coverage metrics of Jee et al. These metrics are such that it can be very difficult to manually come up with a test case in which the test requirement is true. First of all, the test requirements can be quite lengthy. Secondly, the FBD related to the test requirements can also be quite elaborate. It may be challenging to think of a way to drive the system into some desired state if the system has multiple delays or a feedback loop. In some cases it is in fact impossible for the FBD to fulfil a test requirement. Deducing this manually can be very hard.

The test generation technique relies on the classic idea of using model checking to produce test cases: negating the test requirement. If it is possible to fulfil a test requirement, model checking of the negated test requirement will produce a concrete sequence in which this happens.

The workflow related to the technique is illustrated in Figure 8.2. The examined FBD is first modelled using the methodology described in Chapter 5. The test requirements related to the system are deduced by first

determining the data paths of the FBD, writing the function block conditions, and then selecting the test coverage criterion applied on the system. The test requirements can then be negated and written in temporal logic. After model checking, the inputs and expected outputs of the test cases are derived from the counterexamples. FBD modelling and the creation of the function block conditions are manual work, and all the other work phases depicted in Figure 8.2 are automatic.

Following the procedure depicted in Figure 8.2 it is possible to generate a single test case for each test requirement. There may, however, be a lot of test requirements, especially when the FBD is large. In order to reduce the number of test cases, Publication V used a simple greedy algorithm to create efficient test cases in which as many test requirements as possible are fulfilled simultaneously. This algorithm makes repeated queries to the model checker, asking whether a previously generated test case can be modified so that an additional test requirement can also be fulfilled by it. The source code of the implementation of the algorithm used in Publication V is available online.[2]

Based on experience and the tests performed in Publication V the technique does scale to realistic-size nuclear domain safety systems. The test generation times, however, can become quite high for certain types of FBDs. An alternative more efficient test generation approach has later been developed and is described in a research report, see [130].

8.3 Technical issues

There are several technical challenges related to the use of the test generation technique. First of all, the definition Jee et al. use for defining function block conditions is not quite intuitive, and following the manner in which MC/DC defines this input-output relation can be unambiguous. In MC/DC an input of a function block is considered to affect an output only when flipping of the input value also flips the output value while all other inputs of the function block retain their value. One of the findings of Publication V was that function block conditions written according to the MC/DC ideology can be more intuitive and easier to verify automatically.

Another issue in need of further consideration is that in some cases an input of a function block may not have an instant effect on the output. Instead, the input may affect the output with a delay. As an example,

[2] https://github.com/JussiLahtinen/Dissertation

consider a DELAY function block that memorises its input and outputs the value of that input on the previous time point. The input of a DELAY function block at time point n influences the output at time point $n+1$. In order to take this kind of behaviour into account, the function block conditions of such input-output pairs would have to be written as a function of current as well as previous values of the inputs. Delayed dependencies also influence the definition of a data path. A delayed dependency divides the data path into two parts that occur at different time points.

Feedback loops pose a problem in the technique since they induce infinite paths from inputs to outputs. In Publication V this problem is solved by disconnecting the loop, introducing a new input variable that replaces the feedback signal in the loop, and calculating the data paths based on this alternative loop-free design. A design that has a feedback loop induces additional data path fragments to be considered that start from the beginning of the feedback loop and end at an output of the design. The feedback loops are disconnected at such a point that each intermediate signal of the design is present in one of the generated data paths. Disconnecting the feedback loops does not impact the correctness of the eventual test cases, since the tests are generated using the original model in which the feedback loops are still intact.

Analogue variables are also problematic because the coverage metrics assume Boolean valued signals. Function blocks that perform calculations based on analogue variables are problematic because it can difficult to decide when an analogue input affects the output value of that function block. In Publication V the function block conditions of such function blocks were simply defined to have the value $True$ because in some sense the input always affects the output in such analogue calculations.

8.4 Alternative implementation

As a continuation to Publication V, an alternative version of the test generation technique was later developed and documented in a research report [130]. This version of the technique incorporates all test requirements into the model as additional Boolean valued macro formulas. This makes it easy to determine whether other test requirements (in addition to the currently examined one) are also fulfilled by a counterexample. This kind of an approach is more efficient because it reduces the number of iterations needed to generate the whole test set. This alternative

version of the technique also defines the function block conditions in a more formal manner based on the ideology of MC/DC. The definition also supports delayed input-output dependencies enabling the methodology to be used together with a wider range of systems. Issues with analogue variables are dealt with by focusing only on the Boolean path fragments of the system.

The research report related to the alternative version analyses the fault detection capability of the generated test sets using mutation analysis. Two sets of function block diagrams are evaluated in the experiment: (1) a set of fictitious FBDs equivalent to the ones used in Publication V; and (2) a set of vendor-specific real-world FBDs. The results of the research paper suggest that the test generation approach is scalable to most nuclear domain safety systems. On the larger vendor-specific FBDs, however, test generation times can become excessively long. The average fault detection capability of the generated tests ranged from 90 % to 95 % in the mutation analysis experiment.

8.5 Limitations of the technique and threats to validity

One major threat to internal validity has to do with all coverage metrics in general. Namely, there exists no well-defined characterisation of software design errors in general making error models difficult to come up with. Consequently, there is only an intuitive connection between a given coverage metric and an error. In other words, the metrics have no formal meaning and there is no direct correlation between classes of bugs and coverage metrics.

Threats to internal validity might also come from errors in the implementation code, the model-checking model, and the function block conditions that are manually composed. To reduce possible errors in these parts, the implementation of the test generation algorithm was repeatedly tested on many FBDs, and intermediate products such as data paths and test requirements were manually reviewed.

Another fundamental limitation of the technique is related to the test oracle problem, i.e., the problem of distinguishing between desired and undesired behaviour given an input for that system. In the test generation technique a set of test cases is created based on a model of that system's design. The test sequences, however, may represent undesired functionality of the system even if the test sequence is according to sys-

tem specification. This undesired behaviour may arise from design errors, errors in modelling, or omissions in the requirement specification. Ideally, the test generation process would be coupled with a computer-based test oracle that would ensure that the tests always correspond to desired behaviour. In practice, creating such a test oracle for an arbitrary system can be very challenging and time-consuming. Using a human test oracle may be a more feasible approach.

The test generation algorithm of Publication V is intended for creating a small number of tests that fulfil multiple test requirements simultaneously. A threat to construct validity is that such small efficient test cases may be undesirable if there is need to, e.g., identify which test requirement is the one leading to the detection of an error in the system. It can also be quite difficult to determine the correctness of the outputs in elaborate test cases where multiple test requirements are fulfilled at once.

A threat to external validity is whether the methodology can be used for test design in the context of real-world systems. The limitations discussed in Section 5.9 apply here as well. In addition, based on the results of [130] it seems that test generation for larger FBDs can become quite infeasible in practice when the number of test requirements is very high. This problem, however, could be alleviated by dividing such systems into several smaller parts that are separately tested.

The technique as described in Publication V is applicable only to systems where inputs of function blocks always instantaneously affect outputs. The problems related to delayed input-output dependencies were handled in another paper [130] by an extension to the methodology. This extension relies on the fact that the system is cyclically run on constant length intervals so that the system behaviour corresponds to the discrete time model-checking model.

9. Conclusion

Model checking related research in the Finnish nuclear domain began in 2007 as a collaboration between Helsinki University of Technology (TKK) (currently known as Aalto university) and VTT. The author of this dissertation started to work on his Master's thesis at TKK on a related subject [128] in the same year, and was hired as a research scientist at VTT after that.

Since 2007, the topic has been considered important in the SAFIR research programmes (The Finnish Research Programme on Nuclear Power Plant Safety). During the years 2007-2010 the aim of the MODSAFE (Model-based safety evaluation of automation systems) project [198] was to demonstrate the usefulness of the model-checking technique and develop basic methodology for modelling systems. Later, in the SARANA (Safety evaluation and reliability analysis of nuclear automation) project (2011-2014) [31] the focus shifted to further extending the scope of the method, and increasing its scalability. One of the objectives of the currently on-going SAUNA (Integrated safety assessment and justification of nuclear power plant automation) project is the integration of model checking with other techniques. This dissertation is strongly linked to research done in the SAFIR programmes, and these research topics have developed into the research questions of this dissertation.

9.1 Answers to the research questions

This dissertation tackles the question of how to utilise model checking in the verification of large nuclear domain safety systems. This general research topic has been divided into four more specific research questions. Answers to these questions can be found in preceding chapters, and they are summarised in what follows.

RQ1: How can modelling and abstraction techniques be used to enable the model checking of larger nuclear domain automation systems?

Publication I developed generic methodology for modelling safety systems, and Chapter 5 described a more specific technique for modelling system designs described as FBDs. This modelling technique is based on abstracting away from the PLC scan cycle, and utilising a free environment model. Secondly, the system is partitioned into interconnected modules. Each module represents a part of the system design, and consists of a collection of function block instances that are connected together. This modular hierarchy enables simple interface modules developed in Publication II to be used for abstraction. The interface modules replace a single module with an over-approximating abstraction of it. The interface module abstraction simplifies the state graph of the system by releasing model constraints, and thus allows the state space to be more compactly represented, for example, as a BDD. The benefit of this approach is that this kind of coarse high-level abstraction is simple and easily understandable. From the verification point of view the main benefit is that such abstractions retain the truth value of universal properties, making system verification more straight-forward.

RQ2: How can a suitable abstraction level of the system model be found automatically?

Chapter 6 represents a description of an iterative abstraction refinement algorithm developed in Publication III for the purpose of finding an abstraction level suitable for verification. The technique is based on using interface modules for abstraction, and on the analysis of the module-level dependency graph of the system. The fundamental idea of the technique is that typically only a small part of the system is needed for the verification of some specific property. The algorithm starts by trying to find a proof locally using only modules associated with variables used in the property. If this fails the scope of the verification is extended to new modules based on the dependencies between the modules.

RQ3: How can plant-level models be created that cover both the

detailed operational logic of multiple automation systems and the hardware failures related to these systems?

Chapter 7 describes an extension to the FBD modelling methodology that addresses hardware failure modelling. The extension was developed in Publication IV. The principal idea of this work is to use link modules to encapsulate connections between measurements, computers and actuators. The methodology allows large plant-level models to be created, in which fault-tolerance properties can be analysed. However, verification of such large plant models is very challenging in practice due to the sheer size of the model. The modelling technique adopted hardware failure related concepts and terminology from PRA methodology, which makes integration with the two approaches easier. Chapter 7 proposes a concept-level idea of coupling model checking with PRA. This approach uses PRA results to focus the model-checking analysis to only the most critical failures, in an attempt to reduce the complexity of verification.

RQ4: Can model checking be used to support structure-based test design of function block diagrams?

Publication V presented the development of a technique for automatically generating a set of test cases that have high coverage according to structure-based criteria. Chapter 8 describes the technique on a general level. The technique first calculates a set of test requirements the test cases should fulfil, and then uses model checking to create concrete counterexamples representing test cases in which these test requirements are efficiently fulfilled. The developed technique can be of practical value in test design when the examined system is complicated enough to make manual test design infeasible.

9.2 Theoretical implications

One theoretical implication of the work is that the scalability of model checking has been increased for analysing large modular systems. The tests performed on the iterative abstraction refinement algorithm demonstrate that it is often possible to find a proof of a system property locally, and that finding such a local abstraction and extracting a proof from it can be faster than analysing the system as a whole. The systematic

methodology for modelling hardware failures is another important result of this work that extends the scope of model checking to fault-tolerance analyses. The sharing of concepts and terminology with PRA for modelling hardware failures also allows a multi-faceted modelling approach in which models used for either model checking or PRA analysis are generated from a single common plant model. Both, the improved scalability and the extended applicability of model checking can be seen as part of a continuum towards larger plant-level models, and towards new all-encompassing safety analysis approaches. Further integration of model checking and reliability methods may be a useful step in this direction.

9.3 Practical implications

Model checking has been demonstrated to be a valuable tool for detecting design errors. Due to this, model checking has already become a part of the software verification processes used in the Finnish nuclear industry.

The Finnish power company Fortum used model checking in the Loviisa NPP automation renewal project (LARA) to verify the correct functionality of application I&C software [167]. In the Finnish Olkiluoto 3 project, VTT Technical Research Centre of Finland Ltd used model checking in the analysis of two safety-critical systems: the Protection System (PS) and the Priority Actuation and Control System (PACS). Due to non-disclosure agreements, further information on this work is not available. Some of the developed techniques, i.e., the PLC scan cycle abstraction technique, a free environment model, and the utilisation of interface modules for abstraction were utilised in these projects, and have thus also been validated in practice.

9.4 Reliability and validity

The threats to validity and the limitations of the developed methods and algorithms have been extensively covered in the preceding chapters. On a general level, one of the threats to validity is whether systems are being modelled using a suitable abstraction level. This dissertation focuses on the verification of FBDs that are used for designing control application software in nuclear power plants. Any other software such as platform software has not been modelled and is out of scope of this dissertation.

Also, verification of the equivalence between the design and the final system implementation is not included in this work. Furthermore, any asynchronous behaviour due to, e.g., clock drift is out of the scope of this work. Throughout the work it is assumed that distributed systems operate synchronously, and that the timing of the systems has been separately verified to be correct. The limitations of the modelling methodology for FBDs and threats to its external validity was discussed in more detail in Section 5.9. Threats to validity related to the iterative abstraction refinement algorithm was discussed in Section 6.4. Limitations of the hardware modelling methodology was addressed in Section 7.2.1, and the issues related to structure-based testing were covered in Section 8.5.

9.5 Recommendations for future research

Verification of system designs could benefit significantly from machine readable models and automatic translations. Currently the modelling phase consists of a lot of manual work even though tools that facilitate modelling based on a graphical user interface are already being used, see, e.g., [165]. Compiling the model-checking model from the design of the system automatically could further diminish this workload, while simultaneously reducing the effects of human errors.

Another work phase that currently still requires extensive human attention is property formalisation. Tools based on, e.g., the use of temporal logic templates could speed up this work phase, and help the modeller and the engineers developing the requirement specification of the system better understand the meaning of the formalised properties.

A potential useful direction for future research is the development of over-approximating abstractions for function blocks that involve time delays. System designs where long time delays occur may often be modelled by coarsening the time discretisation of the model by lengthening the scan cycle of the model or by shortening the length of the delays. These kind of changes to the model are potentially dangerous, as they disregards certain behaviour existing in the real system. An alternative approach to this could be to develop over-approximations of the function blocks that involve time delays, and develop a verification strategy for using these kinds of abstractions, leading to a more extensive verification result.

The coupled use of PRA and model checking is also left for future work. It is recognised that the methods are somewhat related, and an integrated

approach using both methods could offer a more extensive safety analysis of a plant-level model with results that cannot be obtained in practice by using the methods individually. Applying an iterative approach similar to the one developed in this dissertation to the analysis of these plant-level models is also a potential research direction. Additionally, as the same data can be used as input for both model checking and PRA, it could be beneficial to utilise a common plant-model expressed using a domain specific modelling language for both analysis methods, accompanied with tools capable of automatically generating both the PRA model and the model used for model checking.

Some important aspects of plant operation such as asynchronicity, human operator actions and software code are also left for future research. In this dissertation it is assumed that the analysed system operates as a single synchronous entity. In reality, nuclear safety systems are typically distributed among several computers that use different clocks. Asynchronicity rising from, e.g., data transmission delays or processor clock drift in a large distributed control system also require systematic analysis as they may cause severe problems in these systems. Human-machine interfaces have previously been studied using model checking, see, e.g., [182, 36]. Similar analysis could be used for finding scenarios that lead to poor situational awareness of operators, and for analysing the correctness of the procedural guides that the operators use. This dissertation has focused primarily on the correctness of the system design expressed as FBDs. The final implementation of the system as software code should also be formally analysed. Finally, only application software has been addressed in this work. The correctness of platform level software employed in the I&C systems should also be formally analysed.

Appendix A: Model code for a set of commonly used IEC 61131-3 function blocks

In this appendix, the model code for a small set of often used function blocks is given in the modelling language of NuSMV. The modelled function blocks are: AND, OR, NOT, SR, RS, TP, TON, and TOF. If possible, the conventions of the IEC 61131-3 standard are followed. If a function block has a single input it is named _IN in the models since IN is a reserved word in the NuSMV modelling language. Similarly, the input variable name S is a reserved word in the NuSMV modelling language, and the variable name _S is used instead. If a function block has several inputs they are either named IN1, IN2, etc., unless differing input names are specified in IEC 61131-3. By default the output of a function block is named OUT, but the standard also uses other output variable names for some of the function blocks.

A.1 AND

The AND function block calculates the Boolean AND based on its two inputs: IN1 and IN2. The result is set as the value of OUT. The corresponding NuSMV module is in Listing A.1.

```
1 MODULE AND(IN1, IN2)
2 VAR
3 DEFINE
4    OUT := IN1 & IN2;
5 ASSIGN
```

Listing A.1. NuSMV model code for the AND function block

A.2 OR

The OR function block calculates the Boolean OR based on its two inputs: `IN1` and `IN2`. The result is set as the value of `OUT`. The corresponding NuSMV module is in Listing A.2.

```
MODULE OR(IN1, IN2)
VAR
DEFINE
   OUT := IN1 | IN2;
ASSIGN
```

Listing A.2. NuSMV model code for the OR function block

A.3 NOT

The NOT function block calculates the Boolean NOT based on its input `_IN`. The result is set as the value of `OUT`. The corresponding NuSMV module is in Listing A.3.

```
MODULE NOT(_IN)
VAR
DEFINE
   OUT := ! _IN;
ASSIGN
```

Listing A.3. NuSMV model code for the NOT function block

A.4 SR

```
MODULE SR(S1, R)
VAR
   prev_Q1 : boolean;
DEFINE
   Q1 := (prev_Q1 & ! R) | S1;
ASSIGN
   init(prev_Q1) := FALSE;
   next(prev_Q1) := Q1;
```

Listing A.4. NuSMV model code for the set dominant bistable function block SR

The set dominant bistable function block (also known as a flip-flop memory) has two Boolean inputs: `S1`, and `R`, and a single Boolean internal variable `prev_Q1`. The `S1` input sets `prev_Q1` to true and the `R` input

resets it. S1 is prioritised over R. If both inputs are false prev_Q1 retains its value. The output of the flip-flop (Q1) is true whenever S1 is true, or when prev_Q1 is true and R is false. The initial value of prev_Q1 is set as false. The NuSMV code is in Listing A.4.

A.5 RS

```
1 MODULE RS(_S, R1)
2 VAR
3    prev_Q1 : boolean;
4 DEFINE
5    Q1 := (prev_Q1 | _S) & ! R1;
6 ASSIGN
7    init(prev_Q1) := FALSE;
8    next(prev_Q1) := Q1;
```

Listing A.5. NuSMV model code for the reset dominant bistable function block RS

The reset dominant bistable function block is similar to the set dominant bistable function block except that the resetting input is prioritised over the set input. The function block has two Boolean inputs: _S, and R1, and a single Boolean internal variable prev_Q1. The _S input sets prev_Q1 to true and the R1 input resets it. If both inputs are false prev_Q1 retains its value. The output of the flip-flop (Q1) is true whenever R1 is false and either _S or prev_Q1 are true. The initial value of prev_Q1 is set as false. The NuSMV code is in Listing A.5.

A.6 TP

The time pulse (or TP) function block has two inputs _IN and PT. The PT input is the parameter for the time delay (as a number of scan cycles). TP has two outputs Q and ET. The Q output is set the _IN output turns from false to true, and the output is set for the time period indicated by PT. After the pulse a new rising edge of _IN is needed in order to start a new pulse. Rising edges of _IN are ignored if a pulse is already being output. The ET output indicates the time that has elapsed from the beginning of a pulse. Example timing diagram illustrating the behaviour of TP is presented in Figure A.1.

The NuSMV code for the TP function block is in Listing A.6. The implementation uses a variable prev_IN to track the previous value of _IN,

and a counter variable `clock`. The upper value of `clock` has to be manually adjusted according to the modelled implementation. In this example the value has been set to 100. Initially `clock` is set to 0. The `clock` counter is started when a rising edge is detected in the `_IN` input, and the pulse is currently off (`clock` is 0). The counter is increased on every time step until the value `PT` is reached. After this `clock` can be reset if the `_IN` input is false. The `ET` output follows the value of `clock` except in one case: when `_IN` and the output `Q` are both false `ET` is set to 0. This exception is because it takes one time step to update the value of `clock` when it is reset. The exception is needed in order to show the effect of the reset immediately. The primary output `Q` is set when either a new rising edge is detected in the input, or the clock is running. Otherwise `Q` is false.

```
1  MODULE TP(_IN, PT)
2  VAR
3     prev_IN : boolean;
4     clock : 0..100;
5  DEFINE
6     Q := case
7        clock = 0 & ! prev_IN & _IN : TRUE;
8        clock > 0 & clock < PT : TRUE;
9        TRUE : FALSE;
10    esac;
11    ET := case
12       ! Q & ! _IN : 0;
13       TRUE : clock;
14    esac;
15 ASSIGN
16    init(clock) := 0;
17    next(clock) := case
18       clock = 0 & ! prev_IN & _IN :  1;
19       clock > 0 & clock < PT : clock +1;
20       clock = PT & _IN : PT;
21       clock = PT & ! _IN : 0;
22       TRUE : 0;
23    esac;
24
25    init(prev_IN) := FALSE;
26    next(prev_IN) := _IN;
27
```

Listing A.6. NuSMV model code for the TP function block (time pulse)

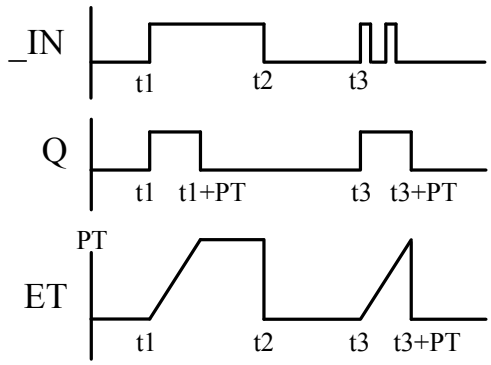


Figure A.1. Example timing behaviour of the TP function block

A.7 TON

The timer ON (or TON) function block has the same inputs and outputs as TP. The output `Q` of the TON timer is set whenever `_IN` has been continuously true for the time period indicated by `PT`. Whenever the `_IN` is false the output `Q` is also false, and the elapsed time counter `ET` is reset. Example timing diagram illustrating the behaviour of TON is presented in Figure A.2.

```
1  MODULE TON(_IN, PT)
2  VAR
3     clock : 0..100;
4  DEFINE
5     ET := case
6        ! _IN : 0;
7        TRUE : clock;
8     esac;
9
10    Q := case
11       clock = PT & _IN : TRUE;
12       TRUE : FALSE;
13    esac;
14 ASSIGN
15    init(clock) := 0;
16    next(clock) := case
17       _IN & clock < PT : clock + 1;
18       _IN & clock = PT : PT;
19       TRUE : 0;
20    esac;
```

Listing A.7. NuSMV model code for the TON function block (timer ON)

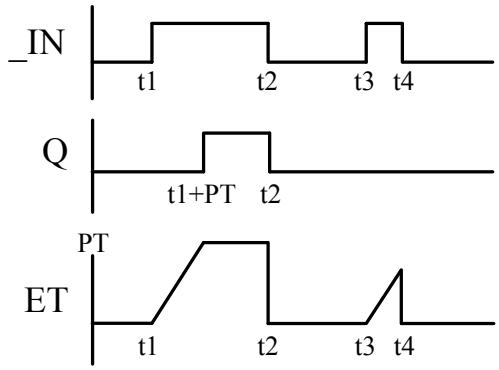


Figure A.2. Example timing behaviour of the TON function block

The NuSMV code for the TON function block is in Listing A.7. The implementation uses a counter variable `clock`. The upper value of `clock` has to be manually adjusted according to the modelled implementation. In this example the value has been set to 100. Initially `clock` is set to 0. The counter is increased by 1 whenever `_IN` is true. If the limit `PT` is reached, the counter will remain at that value. The counter is reset whenever `_IN` is false. The output `Q` is set iff `clock` is at value `PT` and `_IN` is true. The `ET` output follows the value of `clock`. However, if `_IN` is false `ET` is immediately set to 0 so that the reset delay that occurs in the counter is ignored.

A.8 TOF

The timer OFF (or TOF) function block has the same inputs and outputs as TP and TON. The output `Q` of the TOF timer is set whenever `_IN` is set. When `_IN` experiences a falling edge, `Q` remains set for the time period indicated by `PT`. If `_IN` is again set during that time period, the count to `PT` is restarted. Example timing diagram illustrating the behaviour of TOF is presented in Figure A.3.

The NuSMV code for the TOF function block is in Listing A.8. The implementation uses a variable `prev_IN` to track the previous value of `_IN`, and a counter variable `clock`. The upper value of `clock` has to be manually adjusted according to the modelled implementation. In this example the value has been set to 100. Initially `clock` is set to 0. The counter is reset whenever `_IN` is true. The counter is started if the `_IN` has been true at the previous time point, and is now false. If the limit `PT` is reached, the counter will remain at that value. The output `Q` is set whenever `_IN` is

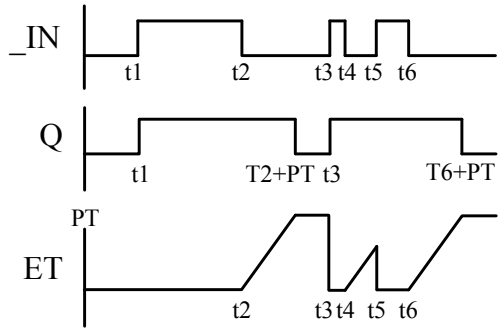


Figure A.3. Example timing behaviour of the TOF function block

true, or the counter is running. The `ET` output follows the value of `clock`. However, if `_IN` is false `ET` is immediately set to 0 so that the reset delay that occurs in the counter is ignored.

```
1  MODULE TOF(_IN, PT)
2  VAR
3     clock : 0..100;
4     prev_IN : boolean;
5  DEFINE
6     ET := case
7        _IN : 0;
8        TRUE : clock;
9     esac;
10
11    Q := case
12       _IN : TRUE;
13       ! _IN & prev_IN  & PT > 0 : TRUE;
14       clock > 0 & clock < PT : TRUE;
15       TRUE : FALSE;
16    esac;
17
18 ASSIGN
19    init(clock) := 0;
20    next(clock) := case
21       _IN : 0;
22       ! _IN & prev_IN : 1;
23       clock > 0 & clock < PT : clock + 1;
24       TRUE : clock;
25    esac;
26    init(prev_IN) := FALSE;
27    next(prev_IN) := _IN;
```

Listing A.8. NuSMV model code for the TOF function block (timer OFF)

Bibliography

[1] Parosh Aziz Abdulla, Johann Deneux, Gunnar Stålmarck, Herman Ågren, and Ove Åkerlund. Designing safe, reliable systems using Scade. In *Leveraging Applications of Formal Methods*, pages 115–129. Springer, 2006.

[2] Nina Amla and Kenneth L. McMillan. Combining abstraction refinement and SAT-based model checking. In Orna Grumberg and Michael Huth, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4424 of *Lecture Notes in Computer Science*, pages 405–419. Springer Berlin / Heidelberg, 2007.

[3] Paul E. Ammann, Paul E. Black, and William Majurski. Using model checking to generate tests from specifications. In *Formal Engineering Methods, 1998. Proceedings. Second International Conference on*, pages 46–54, Dec 1998.

[4] AREVA. *Software Program Manual for TELEPERM XS™ Safety Systems*. Topical report. ANP-10272. Available at `http://pbadupws.nrc.gov/docs/ML0636/ML063610100.pdf`, 2006. [Last accessed: Jan/13/2016].

[5] AREVA. *Instrumentation and Control – TELEPERM XS System Overview*. Available at `http://www.areva.com/mediatheque/liblocal/docs/activites/reacteurs-services/reacteurs/pdf-teleperm-xs-feat.pdf`, 2008. [Last accessed: Jan/13/2016].

[6] André Arnold, Gérald Point, Alain Griffault, and Antoine Rauzy. The AltaRica formalism for describing concurrent systems. *Fundam. Inf.*, 40(2,3):109–124, August 1999.

[7] Eugene Asarin, Oded Maler, and Amir Pnueli. On discretization of delays in timed automata and digital circuits. In *CONCUR'98 Concurrency Theory*, pages 470–484. Springer, 1998.

[8] Stefan Authén and Jan-Erik Holmberg. Reliability analysis of digital systems in a probabilistic risk analysis for nuclear power plants. *Nuclear Engineering and Technology*, 44(5):471–482, 2012.

[9] Stefan Authén, Jan-Erik Holmberg, Tero Tyrväinen, and Lisa Zamani. Guidelines for reliability analysis of digital systems in PSA context - Final report. NKS Report NKS-330, Nordic Nuclear Safety Research (NKS), 2015.

[10] Felice Balarin and Alberto L. Sangiovanni-Vincentelli. An iterative approach to language containment. In *Proceedings of the 5th International*

Conference on Computer Aided Verification, CAV '93, pages 29–40, London, UK, UK, 1993. Springer-Verlag.

[11] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. SLAM and static driver verifier: Technology transfer of formal methods inside microsoft. In Eerke A. Boiten, John Derrick, and Graeme Smith, editors, *IFM*, volume 2999 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2004.

[12] Thomas Ball, Vladimir Levin, and Sriram K. Rajamani. A decade of software model checking with SLAM. *Communications of the ACM*, 54(7):68–76, 2011.

[13] Jiří Barnat. *Distributed memory LTL model checking*. PhD thesis, PhD thesis, Faculty of Informatics, Masaryk University Brno, 2004.

[14] Jiří Barnat, Luboš Brim, Vojtěch Havel, Jan Havlíček, Jan Kriho, Milan Lenčo, Petr Ročkai, Vladimír Štill, and Jiří Weiser. DiVinE 3.0 – An Explicit-State Model Checker for Multithreaded C & C++ Programs. In *Computer Aided Verification (CAV 2013)*, volume 8044 of *LNCS*, pages 863–868. Springer, 2013.

[15] Henning Basold, Henning Günther, Michaela Huhn, and Stefan Milius. An open alternative for SMT-based verification of Scade models. In *Formal Methods for Industrial Critical Systems*, pages 124–139. Springer, 2014.

[16] Tim Bedford and Roger Cooke. Probabilistic risk analysis: foundations and methods, 2001.

[17] Ilan Beer, Shoham Ben-David, Cindy Eisner, and Yoav Rodeh. Efficient detection of vacuity in temporal model checking. *Formal Methods in System Design*, 18(2):141–163, 2001.

[18] Gerd Behrmann, Kim Guldstrand Larsen, Henrik Reif Andersen, Henrik Hulgaard, and Jørn Lind-Nielsen. Verification of hierarchical state/event systems using reusability and compositionality. In *Fifth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '99)*, pages 163–177. Springer, 1999.

[19] Shoham Ben-David, Cindy Eisner, Daniel Geist, and Yaron Wolfsthal. Model checking at IBM. *Formal Methods in System Design*, 22(2):101–108, 2003.

[20] Sergey Berezin, Sérgio Campos, and Edmund M. Clarke. Compositional reasoning in model checking. In Willem-Paul de Roever, Hans Langmaack, and Amir Pnueli, editors, *Compositionality: The Significant Difference*, volume 1536 of *Lecture Notes in Computer Science*, pages 81–102. Springer Berlin Heidelberg, 1998.

[21] Cinzia Bernardeschi, Alessandro Fantechi, and Stefania Gnesi. Model checking fault tolerant systems. *Softw. Test., Verif. Reliab.*, 12(4):251–275, 2002.

[22] Gérard Berry. SCADE: Synchronous design and validation of embedded control software. In *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*, pages 19–33. Springer, 2007.

[23] Pierre Bieber, Charles Castel, and Christel Seguin. Combination of fault tree analysis and model checking for safety assessment of complex system. In *In Proc. 4th European Dependable Computing Conference, volume 2485 of LNCS*, page page. Springer-Verlag, 2002.

[24] Armin Biere, Cyrille Artho, and Viktor Schuppan. Liveness checking as safety checking. *Electronic Notes in Theoretical Computer Science*, 66(2):160–177, 2002.

[25] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In Rance Cleaveland, editor, *TACAS*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1999.

[26] Armin Biere and Koen Claessen. Hardware model checking competition. In *Hardware Verification Workshop*, 2010.

[27] Armin Biere, Keijo Heljanko, Tommi A. Junttila, Timo Latvala, and Viktor Schuppan. Linear encodings of bounded LTL model checking. *Logical Methods in Computer Science*, 2(5:5):1–64, 2006.

[28] Armin Biere, Keijo Heljanko, and Siert Wieringa. AIGER 1.9 and beyond. *Available at* `http://fmv.jku.at/hwmcc11/beyond1.pdf`, 2011. [Last accessed: Jan/13/2016].

[29] Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of satisfiability*, volume 185. IOS press, 2009.

[30] Kim Björkman, Juho Frits, Janne Valkonen, Jussi Lahtinen, Keijo Heljanko, Ilkka Niemelä, and Jari J Hämäläinen. Verification of safety logic designs by model checking. In *Sixth American Nuclear Society International Topical Meeting on Nuclear Plant Instrumentation, Control, and Human-Machine Interface Technologies, (NPIC & HMIT 2009)*, pages 5–9. American Nuclear Society (ANS), 2009.

[31] Kim Björkman, Keijo Heljanko, Kari Kähkönen, Jussi Lahtinen, Antti Pakonen, Markus Porthin, Tero Tyrväinen, and Janne Valkonen. Safety evaluation and reliability analysis of nuclear automation (SARANA). In Jari Hämäläinen and Vesa Suolanen, editors, *SAFIR2014 – The Finnish Research Programme on Nuclear Power Plant Safety 2011–2014 Final Report, (VTT Technology 213)*, chapter 7, pages 103–112. VTT Technical Research Centre of Finland, Espoo, Finland, 2015.

[32] Kim Björkman, Jussi Lahtinen, Tero Tyrväinen, and Jan-Erik Holmberg. Coupling model checking and PRA for safety analysis of digital I&C systems. In *The International Topical Meeting on Probabilistic Safety Assessment and Analysis (PSA 2015)*, pages 384–392. American Nuclear Society (ANS), 2015.

[33] Kim Björkman, Janne Valkonen, and Jukka Ranta. Verification of automated changeover switching unit by model checking. In *Proceedings of the seventh international topical meeting on nuclear plant instrumentation, control and human–machine interface technologies (NPIC & HMIT 2010). Las Vegas (NV)*, pages 1719–28, 2010.

[34] Nikolaj Bjørner, Anca Browne, Eddie Chang, Michael Colón, Arjun Kapur, Zohar Manna, Henny B. Sipma, and Tomás E. Uribe. STeP: Deductive-algorithmic verification of reactive and real-time systems. In *Computer Aided Verification*, pages 415–418. Springer, 1996.

[35] Roderick Bloem, Harold N Gabow, and Fabio Somenzi. An algorithm for strongly connected component analysis in $n \log n$ symbolic steps. In *Formal Methods in Computer-Aided Design*, pages 56–73. Springer, 2000.

[36] Matthew L. Bolton, Radu Siminiceanu, Ellen J. Bass, et al. A systematic approach to model checking human–automation interaction using task analytic models. *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, 41(5):961–976, 2011.

[37] Marco Bozzano, Alessandro Cimatti, Anthony Fernandes Pires, David H. Jones, Greg Kimberly, Tyler Petri, Richard Robinson, and Stefano Tonetta. Formal design and safety analysis of air6110 wheel brake system. In *Computer Aided Verification*, pages 518–535. Springer, 2015.

[38] Marco Bozzano and Adolfo Villafiorita. The FSAP/NuSMV-SA safety analysis platform. *International Journal on Software Tools for Technology Transfer*, 9(1):5–24, 2007.

[39] Aaron R. Bradley. SAT-based model checking without unrolling. In Ranjit Jhala and David A. Schmidt, editors, *VMCAI*, volume 6538 of *Lecture Notes in Computer Science*, pages 70–87. Springer, 2011.

[40] Aaron R. Bradley, Fabio Somenzi, Zyad Hassan, and Yan Zhang. An incremental approach to model checking progress properties. In *Formal Methods in Computer-Aided Design (FMCAD), 2011*, pages 144–153. IEEE, 2011.

[41] Ed Brinksma and Angelika Mader. Verification and optimization of a PLC control schedule. In *SPIN Model checking and software verification*, pages 73–92. Springer, 2000.

[42] Glenn Bruns and Ian Sutherland. Model checking and fault tolerance. In Michael Johnson, editor, *Algebraic Methodology and Software Technology*, volume 1349 of *Lecture Notes in Computer Science*, pages 45–59. Springer Berlin Heidelberg, 1997.

[43] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[44] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and Lain-Jinn Hwang. Symbolic model checking: 10 20 states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[45] Jerry R. Burch, D.E. Long, and Edmund M. Clarke. Symbolic model checking with partitioned transition relations. Technical Report CMU-CS-91-195, Carnegie-Mellon University. Pittsburgh (PA US), 1991. Winner of the Sidney Michaelson best paper award at VLSI 91, Edinburgh, Scotland.

[46] John Callahan, Francis Schneider, Steve Easterbrook, et al. Automated software testing using model-checking. In *Proceedings 1996 SPIN workshop*, volume 353. Citeseer, 1996.

[47] Juan Campos, Rui Abreu, Gordon Fraser, and Marcelo d'Amorim. Entropy-based test generation for improved fault localization. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 257–267, Nov 2013.

[48] Géraud Canet, Sandrine Couffin, Jean-Jacques Lesage, Antoine Petit, and Philippe Schnoebelen. Towards the automatic verification of PLC programs written in instruction list. In *Systems, Man, and Cybernetics, 2000 IEEE International Conference on*, volume 4, pages 2449–2454. IEEE, 2000.

[49] Yan Cao, Qiuzi Lu, Tianhua Xu, Tao Tang, Haifeng Wang, and Yongcheng Xu. Integrating dsl-cbi and nusmv for modeling and verifiying interlocking systems. In *Proceedings of the 5th International Conference on Secure Software Integration & Reliability Improvement Companion (SSIRI-C)*, pages 136–143. IEEE, 2011.

[50] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuxmv symbolic model checker. In *Computer Aided Verification*, pages 334–342. Springer, 2014.

[51] Roberto Cavada, Alessandro Cimatti, Charles Arthur Jochim, Gavin Keighren, Emanuele Olivetti, Marco Pistore, Marco Roveri, and Andrei Tchaltsev. *NuSMV 2.5 User Manual*. FBK-irst, 2010.

[52] Sungdeok Cha, Hanseong Son, Junbeom Yoo, Eunkyung Jee, and Poong Hyun Seong. Systematic evaluation of fault trees using real-time model checker UPPAAL. *Reliability engineering & system safety*, 82(1):11–20, 2003.

[53] Edward Chang, Zohar Manna, and Amir Pnueli. *The safety-progress classification*. Springer, 1993.

[54] Angelo Chiappini, Alessandro Cimatti, Luca Macchi, Oscar Rebollo, Marco Roveri, Angelo Susi, Stefano Tonetta, and Berardino Vittorini. Formalization and validation of a subset of the european train control system. In *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, volume 2, pages 109–118. IEEE, 2010.

[55] John Joseph Chilenski and Steven P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9(5):193–200, 1994.

[56] Yunja Choi and Mats Heimdahl. Model checking software requirement specifications using domain reduction abstraction. In *Proceedings of 18th IEEE International Conference on Automated Software Engineering*, pages 314–317. IEEE, 2003.

[57] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV 2: An opensource tool for symbolic model checking. In Ed Brinksma and Kim Guldstrand Larsen, editors, *CAV*, volume 2404 of *Lecture Notes in Computer Science*, pages 359–364. Springer, 2002.

[58] Alessandro Cimatti, Marco Roveri, Angelo Susi, and Stefano Tonetta. From informal requirements to property-driven formal validation. In *FMICS*, pages 166–181. Springer, 2008.

[59] Alessandro Cimatti, Marco Roveri, Angelo Susi, and Stefano Tonetta. Formalization and validation of safety-critical requirements. *arXiv preprint arXiv:1003.1741*, 2010.

[60] Koen Claessen and Niklas Sorensson. A liveness checking algorithm that counts. In *Formal Methods in Computer-Aided Design (FMCAD), 2012*, pages 52–59. IEEE, 2012.

[61] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, September 2003.

[62] Edmund Clarke and Daniel Kroening. Hardware verification using ANSI-C programs as a reference. In *Proceedings of the 2003 Asia and South Pacific Design Automation Conference*, pages 308–311. ACM, 2003.

[63] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In Dexter Kozen, editor, *Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1981.

[64] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1512–1542, 1994.

[65] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT Press, 2001.

[66] Edmund M. Clarke, Anubhav Gupta, James H. Kukula, and Ofer Shrichman. SAT based abstraction-refinement using ilp and machine learning techniques. In *Proceedings of the 14th International Conference on Computer Aided Verification*, CAV '02, pages 265–279, London, UK, UK, 2002. Springer-Verlag.

[67] Edmund M. Clarke, Anubhav Gupta, and Ofer Strichman. SAT-based counterexample-guided abstraction refinement. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 23(7):1113–1123, 2004.

[68] Edmund M. Clarke, David E. Long, and Kenneth L. McMillan. Compositional model checking. In *LICS*, pages 353–362. IEEE Computer Society, 1989.

[69] Jamieson M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Păsăreanu. Learning assumptions for compositional verification. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 331–346. Springer, 2003.

[70] Patrick Cousot, Radhia Cousot, Jerôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTRÉE analyzer. In *Programming Languages and Systems*, pages 21–30. Springer, 2005.

[71] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C. In *Software Engineering and Formal Methods*, pages 233–247. Springer, 2012.

[72] Leandro Dias Da Silva, Luiz Paulo de Assis Barbosa, Kyller Gorgônio, Angelo Perkusich, and Antonio Marcus Nogueira Lima. On the automatic generation of timed automata models from function block diagrams for safety instrumented systems. In *Industrial Electronics, 2008. IECON 2008. 34th Annual Conference of IEEE*, pages 291–296, Nov 2008.

[73] Satyaki Das and David L. Dill. Successive approximation of abstract transition relations. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*, LICS '01, pages 51–, Washington, DC, USA, 2001. IEEE Computer Society.

[74] David Déharbe and Anamaria Martins Moreira. Using induction and BDDs to model check invariants. In *CHARME*, volume 97, 1997.

[75] Alain Deutsch. Static verification of dynamic properties. *PolySpace White Paper*, 2004.

[76] Guy Durrieu, Odile Laurent, Christel Seguin, and Virginie Wiels. Automatic test case generation for critical embedded systems. *Proceedings of Data Systems In Aerospace (DASIA 2004), Nice, France*, 2004.

[77] Niklas Een, Alan Mishchenko, and Robert Brayton. Efficient implementation of property directed reachability. In *Formal Methods in Computer-Aided Design (FMCAD), 2011*, pages 125–134. IEEE, 2011.

[78] Niklas Eén and Niklas Sörensson. Temporal induction by incremental SAT solving. *Electr. Notes Theor. Comput. Sci.*, 89(4):543–560, 2003.

[79] Ernest Allen Emerson and Chin-Laung Lei. Efficient model checking in fragments of the propositional mu-calculus. In *IEEE Symposium on Logic in Computer Science*, pages 267–278. IEEE Computer Society Press, 1986.

[80] Niklas Eén. The ABC/ZZ verification and synthesis framework, 2014.

[81] André Engels, Loe Feijs, and Sjouke Mauw. Test generation for intelligent networks using model checking. In Ed Brinksma, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1217 of *Lecture Notes in Computer Science*, pages 384–398. Springer Berlin Heidelberg, 1997.

[82] Eduard P. Enoiu, Adnan Čaušević, Thomas J. Ostrand, Elaine J. Weyuker, Daniel Sundmark, and Paul Pettersson. Automated test generation using model checking: an industrial evaluation. *International Journal on Software Tools for Technology Transfer*, pages 1–19, 2014.

[83] Eduard Paul Enoiu, Daniel Sundmark, and Paul Pettersson. Model-based test suite generation for function block diagrams using the UPPAAL model checker. In *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on*, pages 158–167. IEEE, 2013.

[84] FBK-IRST, Carnegie Mellon University, University of Genova and University of Trento. NuSMV model checker v.2.5.4, 2012.

[85] Limor Fix. Fifteen years of formal property verification in intel. In Orna Grumberg and Helmut Veith, editors, *25 Years of Model Checking*, volume 5000 of *Lecture Notes in Computer Science*, pages 139–144. Springer, 2008.

[86] Institut für Sicherheitstechnologie (Istec GmbH). RETRANS. Available at `http://www.istec-gmbh.de/leistungen/qualifizierung/produkte`, [Last accessed Oct/30 2015].

[87] Gordon Fraser and Andrea Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276–291, 2013.

[88] Gordon Fraser, Franz Wotawa, and Paul E. Ammann. Testing with model checkers: a survey. *Software Testing, Verification and Reliability*, 19(3):215–261, 2009.

[89] Juho Frits. Model checking embedded control software. Master's thesis, 2010.

[90] Xiang Gan, Jori Dubrovin, and Keijo Heljanko. A symbolic model checking approach to verifying satellite onboard software. *Science of Computer Programming*, 82:44–55, 2014.

[91] Angelo Gargantini and Gordon Fraser. Generating minimal fault detecting test suites for general boolean specifications. *Information and Software Technology*, 53(11):1263 – 1273, 2011.

[92] Angelo Gargantini and Constance Heitmeyer. Using model checking to generate tests from requirements specifications. In Oscar Nierstrasz and Michel Lemoine, editors, *Software Engineering — ESEC/FSE '99*, volume 1687 of *Lecture Notes in Computer Science*, pages 146–162. Springer Berlin Heidelberg, 1999.

[93] Dimitra Giannakopoulou, Falk Howar, Malte Isberner, Todd Lauderdale, Zvonimir Rakamarić, and Vishwanath Raman. Taming test inputs for separation assurance. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 373–384. ACM, 2014.

[94] Marcelo Glusman, Gila Kamhi, Sela Mador-Haim, Ranan Fraer, and Moshe Y. Vardi. Multiple-counterexample guided iterative abstraction refinement: an industrial evaluation. In *Proceedings of the 9th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'03, pages 176–191, Berlin, Heidelberg, 2003. Springer-Verlag.

[95] Vincent Gourcuff, Olivier De Smet, and Jean-Marc Faure. Efficient representation for formal verification of PLC programs. In *Discrete Event Systems, 2006 8th International Workshop on*, pages 182–187. IEEE, 2006.

[96] Marco Gribaudo, András Horváth, Andrea Bobbio, Enrico Tronci, Ester Ciancamerla, and Michele Minichino. Model-checking based on fluid petri nets for the temperature control system of the icaro co-generative plant. *Lecture notes in computer science*, 2434:273–283, 2002.

[97] Matthias Güdemann, Frank Ortmeier, and Wolfgang Reif. Using deductive cause-consequence analysis (DCCA) with SCADE. In *Computer Safety, Reliability, and Security*, pages 465–478. Springer, 2007.

[98] Grégoire Hamon, Leonardo De Moura, and John Rushby. Generating efficient test sets with a model checker. In *Software Engineering and Formal Methods, 2004. SEFM 2004. Proceedings of the Second International Conference on*, pages 261–270, Sept 2004.

[99] Hannu Harju, Jussi Lahtinen, Jukka Ranta, Risto Nevalainen, and Mika Johansson. Software safety standards for the basis of certification in the nuclear domain. In *Quality of Information and Communications Technology (QUATIC), 2010 Seventh International Conference on the*, pages 54–62, Sept 2010.

[100] Klaus Havelund, Mike Lowry, and John Penix. Formal analysis of a spacecraft controller using SPIN. *Software Engineering, IEEE Transactions on*, 27(8):749–765, 2001.

[101] Klaus Havelund and Thomas Pressburger. Model checking JAVA programs using JAVA PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[102] Klaus Havelund and Grigore Roşu. Efficient monitoring of safety properties. *International Journal on Software Tools for Technology Transfer*, 6(2):158–173, 2004.

[103] Constance L. Heitmeyer and Ralph D. Jeffords. Applying a formal requirements method to three NASA systems: Lessons learned. In *Aerospace Conference, 2007 IEEE*, pages 1–10. IEEE, 2007.

[104] Constance L. Heitmeyer, Ralph D. Jeffords, and Bruce G. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(3):231–261, 1996.

[105] Thomas A Henzinger, Zohar Manna, and Amir Pnueli. What good are digital clocks? In *Automata, Languages and Programming*, pages 545–558. Springer, 1992.

[106] Gerard J. Holzmann. *The SPIN model checker: Primer and reference manual*, volume 1003. Addison-Wesley Reading, 2004.

[107] Michaela Huhn and Stefan Milius. Observations on formal safety analysis in practice. *Science of Computer Programming*, 80:150–168, 2014.

[108] Ralf Huuck. *Software verification for programmable logic controllers*. PhD thesis, University of Kiel, 2003.

[109] International Atomic Energy Agency (IAEA). Defence in depth in nuclear safety, INSAG-10 – A report by the International Nuclear Safety Advisory Group, 1996.

[110] IEC. *IEC 61131-3 (2013): International Standard for Programmable Controllers — Part 3: Programming Languages*. 1993.

[111] IEC. *IEC 61499 (2005): International Standard IEC 61499, Function Blocks, Part 1 — Part 4*. 2005.

[112] IEC. IEC 60812: Analysis techniques for system reliability – Procedure for failure mode and effects analysis (FMEA), 2006.

[113] IEC. *ISO/IEC 61508-3 ed2.0 (2010): Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 3: Software requirements*. 2010.

[114] IEC. *ISO/IEC 29119-4 (2013): Software and systems engineering — Software testing — Part 4: Test techniques*. 2013.

[115] IEEE. IEEE standard for property specification language (PSL). *IEEE Std 1850-2005*, pages 1–143, 2005.

[116] International Atomic Energy Agency (IAEA). Safety of Nuclear Power Plants: Design, IAEA Safety Standards series No. SSR-2/1, 2012.

[117] Eunkyoung Jee, Seungjae Jeon, Sungdeok Cha, Kwangyong Koh, Junbeom Yoo, Geeyong Park, Poonghyun Seong, et al. FBDVerifier: interactive and visual analysis of counter-example in formal verification of function block diagram. *Journal of Research and Practice in Information Technology*, 42(3):171, 2010.

[118] Eunkyoung Jee, Suin Kim, Sungdeok Cha, and Insup Lee. Automated test coverage measurement for reactor protection system software implemented in function block diagram. In *Proceedings of the 29th International Conference on Computer Safety, Reliability, and Security*, SAFECOMP'10, pages 223–236, Berlin, Heidelberg, 2010. Springer-Verlag.

[119] Eunkyoung Jee, Donghwan Shin, Sung Deok Cha, Jang-Soo Lee, and Doo-Hwan Bae. Automated test case generation for FBD programs implementing reactor protection system software. *Softw. Test., Verif. Reliab.*, 24(8):608–628, 2014.

[120] Eunkyoung Jee, Junbeom Yoo, Sung Deok Cha, and Doohwan Bae. A data flow-based structural testing technique for FBD programs. *Information & Software Technology*, 51(7):1131–1139, 2009.

[121] Anjali Joshi and Mats P. E. Heimdahl. Model-based safety analysis of Simulink models using SCADE design verifier. In *Proceedings of the 24th International Conference on Computer Safety, Reliability, and Security*, SAFECOMP'05, pages 122–135, Berlin, Heidelberg, 2005. Springer-Verlag.

[122] Kwang Yong Koh and Poong Hyun Seong. SMV model-based safety analysis of software requirements. *Reliability Engineering & System Safety*, 94(2):320 – 331, 2009.

[123] Seo-Ryong Koo, Poong Hyun Seong, JunBeom Yoo, Sung Deok Cha, Cheong Youn, and Hyun-Chul Han. NuSEE: an integrated environment of software specification and V&V for PLC based safety-critical systems. *Nuclear Engineering and Technology*, 38(3):259–276, 2006.

[124] Matti Koskimies. Applying model checking to analysing safety instrumented systems. Master's thesis, 2008.

[125] Lars M. Kristensen and Kent Inge Fagerland Simonsen. Applications of Coloured Petri Nets for functional validation of protocol designs. In *Transactions on Petri Nets and Other Models of Concurrency VII*, pages 56–115. Springer, 2013.

[126] Orna Kupferman and Moshe Y. Vardi. Model checking of safety properties. *Formal Methods in System Design*, 19(3):291–314, 2001.

[127] Robert P. Kurshan. *Computer-aided verification of coordinating processes: the automata-theoretic approach*. Princeton University Press, Princeton, NJ, USA, 1994.

[128] Jussi Lahtinen. Model checking timed safety instrumented systems. Master's thesis, 2008.

[129] Jussi Lahtinen. Hardware failure modelling methodology for model checking. Research report VTT-R-00213-14, VTT Technical Research Centre of Finland, Espoo, Finland, 2014.

[130] Jussi Lahtinen. Supporting structure-based test design using model checking. Research report VTT-R-04004-15, VTT Technical Research Centre of Finland Ltd., Espoo, Finland, 2016.

[131] Jussi Lahtinen, Kim Björkman, Janne Valkonen, Juho Frits, and Ilkka Niemelä. Analysis of an emergency diesel generator control system by compositional model checking. VTT Working Papers 156, VTT Technical Research Centre of Finland, 2010.

[132] Jussi Lahtinen and Kim Björkman. Feasibility study on the integration of PRA methods and model checking. Research report VTT-R-04924-15, VTT Technical Research Centre of Finland Ltd., Espoo, Finland, 2016.

[133] Jussi Lahtinen, Mika Johansson, Jukka Ranta, Hannu Harju, and Risto Nevalainen. Comparison between IEC 60880 and IEC 61508 for certification purposes in the nuclear domain. In Erwin Schoitsch, editor, *Computer Safety, Reliability, and Security*, volume 6351 of *Lecture Notes in Computer Science*, pages 55–67. Springer Berlin Heidelberg, 2010.

[134] Jussi Lahtinen, Janne Valkonen, Kim Björkman, Juho Frits, and Ilkka Niemelä. Model checking methodology for supporting safety critical software development and verification. In *European Safety and Reliability Conference, ESREL2010*, pages 2056–2063, September 2010.

[135] Leslie Lamport. Proving the correctness of multiprocess programs. *Software Engineering, IEEE Transactions on*, (2):125–143, 1977.

[136] Timo Latvala. Efficient model checking of safety properties. In *Model Checking Software*, pages 74–88. Springer, 2003.

[137] Mark Lawford, Peter Froebel, and Greg Moum. Application of tabular methods to the specification and verification of a nuclear reactor shutdown system. *Formal Methods in System Design*, 2004.

[138] Dong-Ah Lee, Junbeom Yoo, and Jang-Soo Lee. A systematic verification of behavioral consistency between FBD design and ANSI-C implementation using HW-CBMC. *Reliability Engineering & System Safety*, 120:139 – 149, 2013.

[139] Dominique L'Her, Philippe Le Parc, and Lionel Marcé. Proving sequential function chart programs using automata. In *Automata Implementation*, pages 149–163. Springer, 1999.

[140] Orna Lichtenstein and Amir Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 97–107. ACM, 1985.

[141] Jørn Lind-Nielsen, Henrik Reif Andersen, Henrik Hulgaard, Gerd Behrmann, Kåre Kristoffersen, and Kim Guldstrand Larsen. Verification of large state/event systems using compositionality and dependency analysis. *Formal Methods in System Design*, 18(1):5–23, 2001.

[142] Alessio Lomuscio, Franco Raimondi, and Marek J. Sergot. Towards model checking interpreted systems. In *Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, pages 1054–1055. ACM, 2003.

[143] Angelika Mader. A classification of PLC models and applications. In R. Boel and G. Stremersch, editors, *Discrete event systems: analysis and control*, pages 239–247. Kluwer Academic, 2000.

[144] Angelika Mader and Hanno Wupper. Timed automaton models for simple programmable logic controllers. In *Real-Time Systems, 1999. Proceedings of the 11th Euromicro Conference on*, pages 106–113. IEEE, 1999.

[145] Zohar Manna and Amir Pnueli. A hierarchy of temporal properties (invited paper, 1989). In *Proceedings of the ninth annual ACM symposium on Principles of distributed computing*, pages 377–410. ACM, 1990.

[146] Cristian Mattarei, Alessandro Cimatti, Marco Gario, Stefano Tonetta, and Kristin Y. Rozier. Comparing different functional allocations in automated air traffic control design. In *Formal Methods in Computer-Aided Design (FMCAD 2015), Austin, Texas, USA*. IEEE/ACM, 2015.

[147] Kenneth L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. PhD thesis, Pittsburgh, PA, USA, 1992. UMI Order No. GAX92-24209.

[148] Kenneth L. McMillan. *Symbolic model checking*. Kluwer, 1993.

[149] Kenneth L. McMillan. Circular compositional reasoning about liveness. In Laurence Pierre and Thomas Kropf, editors, *Correct Hardware Design and Verification Methods*, volume 1703 of *Lecture Notes in Computer Science*, pages 342–346. Springer Berlin Heidelberg, 1999.

[150] Kenneth L. McMillan. Getting started with SMV. *Cadence Berkeley Laboratories*, 1999.

[151] Kenneth L. McMillan and Nina Amla. Automatic abstraction without counterexamples. In Hubert Garavel and John Hatcliff, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2619 of *Lecture Notes in Computer Science*, pages 2–17. Springer Berlin / Heidelberg, 2003.

[152] Horst Miedl. RETRANS – a tool to verify the functional equivalence of automatically generated source code with its specification. 1998.

[153] Steven Miller, Elise Anderson, Lucas Wagner, Michael Whalen, and Matts Heimdahl. Formal verification of flight critical software. In *Proceedings of the AIAA Guidance, Navigation and Control Conference and Exhibit*, pages 15–18, 2005.

[154] Steven P. Miller. Will this be formal? In *Theorem Proving in Higher Order Logics*, pages 6–11. Springer, 2008.

[155] Steven P. Miller, Alan C. Tribble, and Mats P. E. Heimdahl. Proving the shalls. In *FME 2003: Formal Methods*, pages 75–93. Springer, 2003.

[156] Jan Mrázek, Petr Bauch, Henrich Lauko, and Jiří Barnat. SymDIVINE: Tool for control-explicit data-symbolic state space exploration. In Dragan Bošnački and Anton Wijs, editors, *Model Checkings Software*, volume 9641 of *Lecture Notes in Computer Science*, pages 208–213. Springer, 2016.

[157] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[158] Erzsébet Németh and Tamás Bartha. Formal verification of safety functions by reinterpretation of functional block based specifications. In Darren Cofer and Alessandro Fantechi, editors, *Formal Methods for Industrial Critical Systems*, volume 5596 of *Lecture Notes in Computer Science*, pages 199–214. Springer Berlin Heidelberg, 2009.

[159] Erzsébet Németh, Tamás Bartha, Cs Fazekas, and Katalin M. Hangos. Verification of a primary-to-secondary leaking safety procedure in a nuclear power plant using coloured Petri nets. *Reliability Engineering & System Safety*, 94(5):942 – 953, 2009.

[160] Thomas M. Niermann, Rabindra K. Roy, Janak H. Patel, and Jacob Abraham. Test compaction for sequential circuits. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 11(2):260–267, Feb 1992.

[161] Frank Ortmeier, Gerhard Schellhorn, Andreas Thums, Wolfgang Reif, Bernhard Hering, and Helmut Trappschuh. Safety analysis of the height control system for the Elbtunnel. *Reliability Engineering & System Safety*, 81(3):259–268, 2003.

[162] Alain Ourghanlian. Evaluation of static analysis tools used to assess software important to nuclear power plant safety. *Nuclear Engineering and Technology*, 47(2):212 – 218, 2015. Special Issue on ISOFIC/ISSNP2014.

[163] Tolga Ovatman, Atakan Aral, Davut Polat, and Ali Osman Ünver. An overview of model checking practices on verification of PLC software. *Software & Systems Modeling*, pages 1–24, 2014.

[164] Antti Pakonen, Jussi Lahtinen, Veli-Pekka Kuutti, and Tommi Karhela. Integrating model checking with safety-critical I&C software design. In *7th International International Topical Meeting on Nuclear Plant Instrumentation, Control, and Human-Machine Interface Technologies, (NPIC & HMIT 2010)*, pages 1729–1740. American Nuclear Society (ANS), 2010.

[165] Antti Pakonen, Teemu Mätäsniemi, Jussi Lahtinen, and Tommi Karhela. A toolset for model checking of PLC software. In *IEEE 18th Conference on*

Emerging Technologies & Factory Automation (ETFA), pages 1–6, September 2013.

[166] Antti Pakonen, Teemu Mätäsniemi, and Janne Valkonen. Model checking reveals hidden errors in safety-critical I&C software. In *8th International Topical Meeting on Nuclear Plant Instrumentation, Control and Human-Machine Interface Technologies, (NPIC & HMIT 2012)*, pages 1823–1834. American Nuclear Society (ANS), 2012.

[167] Antti Pakonen, Janne Valkonen, Sami Matinaho, and Markus Hartikainen. Model checking for licensing support in the Finnish nuclear industry. In *International Symposium on Future I&C for Nuclear Power Plants (ISOFIC 2014), Jeju Island, Repulic of Korea*, 2014.

[168] Olivera Pavlovic and Hans-Dieter Ehrich. Model checking PLC software written in function block diagram. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pages 439–448, April 2010.

[169] Daniel Pilaud, Nicolas Halbwachs, and John A. Plaice. LUSTRE: A declarative language for programming synchronous systems. In *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages (14th POPL 1987). ACM, New York, NY*, volume 178, page 188, 1987.

[170] Amir Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57. IEEE, 1977.

[171] Corina S. Păsăreanu, Matthew B. Dwyer, and Michael Huth. Assume-guarantee model checking of software: A comparative case study. In Dennis Dams, Rob Gerth, Stefan Leue, and Mieke Massink, editors, *Theoretical and Practical Aspects of SPIN Model Checking*, volume 1680 of *Lecture Notes in Computer Science*, pages 168–183. Springer Berlin Heidelberg, 1999.

[172] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In Mariangiola Dezani-Ciancaglini and Ugo Montanari, editors, *Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer, 1982.

[173] Sandra Rapps and Elaine J. Weyuker. Selecting software test data using data flow information. *Software Engineering, IEEE Transactions on*, (4):367–375, 1985.

[174] Kavita Ravi, Roderick Bloem, and Fabio Somenzi. A comparative study of symbolic algorithms for the computation of fair cycles. In *Formal Methods in Computer-Aided Design*, pages 162–179. Springer, 2000.

[175] Steffen Richter and Jens-Uwe Wittig. Verification and validation process for safety I&C systems. *Nuclear Plant Journal*, 21(3):36–40, 2003.

[176] Rolls-Royce. *Spinline^{TM} – A Rolls-Royce modular I&C digital platform dedicated to nuclear safety*. Technical sheet. Available at `http://www.rolls-royce.com/~/media/Files/R/Rolls-Royce/documents/customers/nuclear/spinlinetm-tcm92-50342.pdf`, 2012. [Last accessed: Jan/13/2016].

[177] Olivier Rossi and Philippe Schnoebelen. Formal modeling of timed function blocks for the automatic verification of Ladder Diagram programs. In *Proc. 4th Int. Conf. Automation of Mixed Processes: Hybrid Dynamic Systems (ADPM'2000), Dortmund, Germany*, pages 177–182. Citeseer, 2000.

[178] Kristin Y. Rozier. Linear temporal logic symbolic model checking. *Computer Science Review*, 5(2):163 – 203, 2011.

[179] Kristin Y. Rozier and Moshe Y. Vardi. Ltl satisfiability checking. In *14th Workshop on Model Checking Software (SPIN '07), volume 4595 ofLecture Notes in Computer Science (LNCS)*, pages 149–167. Springer-Verlag, 2007.

[180] Kristin Y. Rozier and Moshe Y. Vardi. A multi-encoding approach for ltl symbolic satisfiability checking. In *17th International Symposium on Formal Methods (FM2011), volume 6664 of Lecture Notes in Computer Science (LNCS)*, pages 417–431. Springer-Verlag, 2011.

[181] John Rushby. Formal verification of McMillan's compositional assume-guarantee rule. In *University of Minnesota, Minneapolis. His*. Citeseer, 2001.

[182] John Rushby. Using model checking to help discover mode confusions and other automation surprises. *Reliability Engineering & System Safety*, 75(2):167–177, 2002.

[183] Mahdi Sarabi. Evaluation of structural testing effectiveness in industrial model-driven software development. Master's thesis, Mälardalen University, June 2012.

[184] Bastian Schlich, Jörg Brauer, Jörg Wernerus, and Stefan Kowalewski. Direct model checking of PLC programs in IL. *Proceedings of DCDS*, pages 28–33, 2009.

[185] Francis Schneider, Steve M. Easterbrook, John R. Callahan, and Gerard J. Holzmann. Validating requirements for fault tolerant systems using model checking. In *ICRE*, pages 4–13. IEEE Computer Society, 1998.

[186] Viktor Schuppan and Armin Biere. Efficient reduction of finite state model checking to reachability analysis. *International Journal on Software Tools for Technology Transfer*, 5(2-3):185–204, 2004.

[187] Septavera Sharvia and Yiannis Papadopoulos. IACoB-SA: An approach towards integrated safety assessment. In *CASE*, pages 220–225. IEEE, 2011.

[188] Septavera Sharvia and Yiannis Papadopoulos. Integrating model checking with HiP-HOPS in model-based safety analysis. *Reliability Engineering & System Safety*, 135:64 – 80, 2015.

[189] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a SAT-solver. In Warren A. Hunt Jr. and Steven D. Johnson, editors, *FMCAD*, volume 1954 of *Lecture Notes in Computer Science*, pages 108–125. Springer, 2000.

[190] Doaa Soliman and Georg Frey. Verification and validation of safety applications based on PLCopen safety function blocks. *Control Engineering Practice*, 19(9):929 – 946, 2011. Special Section: DCDS'09 – The 2nd {IFAC} Workshop on Dependable Control of Discrete Systems.

[191] Doaa Soliman, Kleanthis Thramboulidis, and Georg Frey. Function block diagram to UPPAAL timed automata transformation based on formal models. *Information Control Problems in Manufacturing*, 14(1):1653–1659, 2012.

[192] Doaa Soliman, Kleanthis Thramboulidis, and Georg Frey. Transformation of function block diagrams to UPPAAL timed automata for the verification of safety applications. *Annual Reviews in Control*, 36(2):338 – 345, 2012.

[193] Baruch Sterin, Niklas Een, Alan Mishchenko, and Robert Brayton. The benefit of concurrency in model checking. In *IWLS'11*, pages 176–182, 2011.

[194] Teemu Tommila and Antti Pakonen. Controlled natural language requirements in the design and analysis of safety critical I&C systems. Research report VTT-R-01067-14, VTT Technical Research Centre of Finland, 2014.

[195] Alan C. Tribble and Stephan P. Miller. Software safety analysis of a flight management system vertical navigation function – a status report. In *In Proceedings of the 22ndDigital Avionics Systems Conference (DASC'03)*, volume 1, pages 1–B. IEEE, 2003.

[196] Iñigo Ugarte and Pablo Sanchez. Formal meaning of coverage metrics in simulation-based hardware design verification. In *High-Level Design Validation and Test Workshop, 2005. Tenth IEEE International*, pages 221–228. IEEE, 2005.

[197] USNRC. Software Unit Testing for Digital Computer Software Used in Safety Systems of Nuclear Power Plants, Regulatory Guide 1.171, 1997.

[198] Janne Valkonen, Kim Björkman, Jussi Lahtinen, Jukka Ranta, Juho Frits, Keijo Heljanko, and Ilkka Niemelä. Model-based safety evaluation of automation systems (MODSAFE). In Eija Karita Puska and Vesa Suolanen, editors, *The Finnish Research Programme on Nuclear Power Plant Safety 2007–2010 Final Report, (VTT Research notes 2571)*, chapter 4, pages 55–65. VTT Technical Research Centre of Finland, Espoo, Finland, 2011.

[199] Janne Valkonen, Matti Koskimies, Ville Pettersson, Keijo Heljanko, Jan-Erik Holmberg, Ilkka Niemelä, and Jari J. Hämäläinen. Formal verification of safety I&C system designs: Two nuclear power plant related applications. In *Enlarged Halden Programme Group Meeting-Proceedings of the Man-Technology-Organisation Sessions C*, volume 4, 2008.

[200] Janne Valkonen, Ville Pettersson, Kim Björkman, Jan-Erik Holmberg, Matti Koskimies, Keijo Heljanko, and Ilkka Niemelä. Model-based analysis of an arc protection and an emergency cooling system. In *VTT Working Papers 93, VTT Technical Research Centre of*, 2008.

[201] Mario Van der Borst and Herman Schoonakker. An overview of PSA importance measures. *Reliability Engineering & System Safety*, 72(3):241–245, 2001.

[202] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. In *1st Symposium in Logic in Computer Science (LICS)*. IEEE Computer Society, 1986.

[203] William E. Vesely, Francine F. Goldberg, Norman H. Roberts, and David F. Haasl. Fault tree handbook. Technical report, DTIC Document, 1981.

[204] Amol Wakankar, Raka Mitra, Anup K. Bhattacharjee, Shraddha V. Shrikhande, Sham D. Dhodapkar, and Rajendra K. Patil. Formal model based methodology for developing controllers for nuclear applications. In *Proceedings of 20th IEEE International Symposium on Software Reliability Engineering (ISSRE-2009), Mysore, India*, 2009.

[205] Dong Wang, Pei-Hsin Jiang, James Kukula, Yunshan Zhu, Tony Ma, and Robert Damiano. Formal property verification by abstraction refinement with formal, simulation and hybrid engines. In *Proceedings of the 38th annual Design Automation Conference*, DAC '01, pages 35–40, New York, NY, USA, 2001. ACM.

[206] John X. Wang and Marvin L. Roush. *What every engineer should know about risk engineering and management*. CRC Press, 2000.

[207] Nick J. Ward. The rigorous retrospective static analysis of the Sizewell 'B' primary protection system software. In *SAFECOMP'93*, pages 171–181. Springer, 1993.

[208] Alan Wassyng and Mark Lawford. Lessons learned from a successful implementation of formal methods in an industrial project. In *FME 2003: Formal Methods*, pages 133–153. Springer, 2003.

[209] John T. Webb. MALPAS—an automatic static analysis tool for software validation and verification. In *Edited papers presented at the 1st International Conference on Reliability and robustness of engineering software*, pages 67–75. Elsevier Science Publishers BV, 1987.

[210] Rik Willems. Compact timed automata for PLC programs. 1999.

[211] Junbeom Yoo, Sungdeok Cha, and Eunkyoung Jee. A verification framework for FBD based software in nuclear power plants. In *Software Engineering Conference, 2008. APSEC '08. 15th Asia-Pacific*, pages 385–392, Dec 2008.

[212] Junbeom Yoo, Sungdeok Cha, and Eunkyoung Jee. Verification of PLC programs written in FBD with VIS. *Nuclear Engineering and Technology*, (1):79–90, 2009.

[213] Junbeom Yoo, Sungdeok Cha, Chang Hwoi Kim, and Duck Yong Song. Synthesis of fbd-based PLC design from NuSCR formal specification. *Reliability Engineering & System Safety*, 87(2):287–294, 2005.

[214] Junbeom Yoo, Eunkyoung Jee, and Sung Deok Cha. Formal modeling and verification of safety-critical software. *IEEE Software*, 26(3):42–49, 2009.

[215] Petr Závodskỳ. Independent assessment of the Temelín safety system software. *NEA/CSNI/R (2002) 1/VOL1 Un classified*, page 63, 2002.

[216] Andreas Zeller. Isolating cause-effect chains from computer programs. In *SIGSOFT FSE*, pages 1–10, 2002.

[217] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Software Eng.*, 28(2):183–200, 2002.

[218] Yang Zhao and Kristin Yvonne Rozier. Formal specification and verification of a coordination protocol for an automated air traffic control system. *Science of Computer Programming*, 96:337–353, 2014.

[219] Hao Zheng, Haiqiong Yao, and Tomohiro Yoneda. Modular model checking of large asynchronous designs with efficient abstraction refinement. *IEEE Trans. Comput.*, 59(4):561–573, April 2010.

Errata

Publication I

The actual size of the state space mentioned in Section 4.2 should be 10^{18} instead of 1018.

Publication II

Reference 7 of the paper has been incorrectly written. The correct reference is:
Corina S. Păsăreanu, Matthew B. Dwyer, and Michael Huth. Assume-guarantee model checking of software: A comparative case study. In Dennis Dams, Rob Gerth, Stefan Leue, and Mieke Massink, editors, *Theoretical and Practical Aspects of SPIN Model Checking*, volume 1680 of *Lecture Notes in Computer Science*, pages 168–183. Springer Berlin Heidelberg, 1999.

Publication I

Jussi Lahtinen, Janne Valkonen, Kim Björkman, Juho Frits, Ilkka Niemelä and Keijo Heljanko. Model checking of safety critical software in the nuclear engineering domain. *Reliability Engineering & System Safety*, Vol. 105, p. 104 – 113, Elsevier 2012.

Reliability Engineering and System Safety 105 (2012) 104–113

Contents lists available at SciVerse ScienceDirect

Reliability Engineering and System Safety

journal homepage: www.elsevier.com/locate/ress

Model checking of safety-critical software in the nuclear engineering domain

J. Lahtinen [a,*], J. Valkonen [a], K. Björkman [a], J. Frits [b], I. Niemelä [b], K. Heljanko [b]

[a] VTT Technical Research Centre of Finland, Systems Research, P.O. Box 1000, FI-02044 Espoo, Finland
[b] Department of Information and Computer Science, School of Science, Aalto University, PO Box 15400, FI-00076 Aalto, Finland

ARTICLE INFO

Article history:
Received 31 March 2011
Received in revised form
6 March 2012
Accepted 25 March 2012
Available online 2 April 2012

Keywords:
Model checking
Verification
Safety
I&C
Automation
Nuclear

ABSTRACT

Instrumentation and control (I&C) systems play a vital role in the operation of safety-critical processes. Digital programmable logic controllers (PLC) enable sophisticated control tasks which sets high requirements for system validation and verification methods. Testing and simulation have an important role in the overall verification of a system but are not suitable for comprehensive evaluation because only a limited number of system behaviors can be analyzed due to time limitations. Testing is also performed too late in the development lifecycle and thus the correction of design errors is expensive. This paper discusses the role of formal methods in software development in the area of nuclear engineering. It puts forward model checking, a computer-aided formal method for verifying the correctness of a system design model, as a promising approach to system verification. The main contribution of the paper is the development of systematic methodology for modeling safety critical systems in the nuclear domain. Two case studies are reviewed, in which we have found errors that were previously not detected. We also discuss the actions that should be taken in order to increase confidence in the model checking process.

1. Introduction

The traditional method of assessing the system correctness relies on testing and simulation techniques. In testing, the basic idea is to exercise the implemented system itself and assess its correctness using a collection of test cases. In simulation, the aim is to capture the system behavior in a system model and verify the correctness of the system by simulating different scenarios one by one using this model. Another more formal method is deductive verification, which uses computer-aided theorem provers to prove system correctness. It is time-consuming and can only be performed by experts with considerable experience, but can scale to very large systems [1]. All these methods play their part in the design process, but for exhaustive verification with reasonable effort and time, none of them alone is suitable.

When systems become increasingly complicated, both testing and simulation are faced with serious scaling problems. On one hand, testing or simulating different scenarios is time-consuming and, hence, only a small fraction of all possible behaviors can be covered in practice. On the other hand, it becomes increasingly difficult to develop a collection of test cases or scenarios that exercises all relevant behavior of the system model in order to provide sufficient evidence of its correctness.

The poor coverage of the traditional verification methods sets high requirements for the verification of safety-critical industrial systems that are increasingly dependent on software components and the use of digital technology. The use of digital programmable logic controllers often increases system complexity through a wide range of different function blocks the designer can choose from. Suboptimal design decisions frequently lead to unnecessarily complicated software and system designs. This complexity often leads to problems in system verification. For this reason, in safety-critical applications the use of formal methods based on mathematics and logic is becoming a more fundamental part of development and verification processes.

Model checking is a computer-aided formal method for verifying the correct functioning of a system design model [2,3]. Unlike traditional verification methods, model checking examines all possible behaviors of the model.

In this paper, we describe how formal model checking methodology can complement the traditional safety methods and help reveal unlikely but fatal system behaviors that on occasion go undiscovered with traditional methods. The proposed methodology can be used efficiently for exhaustive verification of safety critical I&C systems with reasonable effort, especially in the early phases of the design process. The employed model checking approach can also be used to generate interesting scenarios for more detailed inspection with traditional testing and simulation.

The main contribution of the paper is the development of systematic methodology for modeling function block based

* Corresponding author.
E-mail address: jussi.lahtinen@vtt.fi (J. Lahtinen).

0951-8320/$ - see front matter © 2012 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.ress.2012.03.021

system designs using two model checking tools: NuSMV and UPPAAL. The efficiency of the methodology is based on a reusable function block library, a modular model structure, and a non-restricted environment model.

We also demonstrate how system-level analysis of a device can be accomplished by model checking. Building a system-level model of a device can be done even when the detailed documentation of the system is not available. This requires reverse-engineering and interviewing the system developers.

The modeling methodology was successfully employed in several case studies. This paper reviews two of these case studies, in which a real-world industrial system was successfully analyzed using model checking. In both cases, an error was found through model checking that had previously gone undetected. Based on our model checking experience, we also present errors typically found by model checking, and analyze the root causes of these errors.

Finally, we identify the reasons why model checking is not yet used in larger scale. We give recommendations to improve the model checking process in order to make it more efficient, and increase confidence in the correctness of its results.

The remainder of this paper is structured as follows: Section 2 introduces model checking. Section 3 reviews the related work. Section 4 reviews our model checking methodology and two case studies. Discussion regarding the limitations of model checking is presented in Section 5. Section 6 concludes the paper.

2. Model checking

Model checking [2,3] is a computer-aided verification method developed to formally verify the correctness of a system design model by examining all of its possible behaviors. The models used in model checking are quite similar to those used in simulation, as the model must essentially describe the behavior of the system design for all sequences of inputs. Typically, some variant of state machines or digital circuits is used to model the system. However, unlike simulation, model checkers examine the behavior of the system design with all input sequences and compare it to a formal specification of the system. The specification is expressed in a suitable language, temporal logics being a prime example, describing the permitted system behaviors. In model checking, at least in principle, the analysis can be carried out fully automatically using computer aided tools. Given a model and a specification as input, a model checking algorithm determines whether the system violates the specification. If there are system behaviors that violate a given specification, the model checker will automatically give a counter-example execution of the model demonstrating how the property can be violated. In this work two model checking tools are examined: NuSMV [4,5] and UPPAAL [6].

2.1. Model checking process for critical I&C system designs

The process of using model checking for verifying system designs is illustrated in Fig. 1. The first step is to analyze the type of properties to be verified using the model by examining the overall design of the system. This facilitates the definition of model boundaries and the selection of system parts that will be abstracted away as part of the environment model. This step helps to avoid unnecessarily complicated models that can pose challenges to the efficiency and performance of the model checking tools.

The model is built based on design documentation such as the system's functional descriptions and logic diagrams. The model is usually a state machine model but the behavior of the model is written in the modeling language of the model checking tool used. For systematic modeling it is important to identify the system boundaries and the interface between the system and its environment. Another key issue is to choose an appropriate level of abstraction for the model so that irrelevant details are abstracted away and, thus, the computational cost of performing the model checking task remains reasonable. The objective is to include only the most important system behaviors, known as the smallest sufficient model [22]. The rest is abstracted on a suitable level depending on the property to be verified.

Another important and challenging task is the definition of the properties to be verified in a detailed level. Typically, the system requirements that are the basis of designing the system are not precise and detailed enough to be directly used for model checking. They must first be dismantled into more detailed requirements that define the desired behavior of the system at the input–output level of the model. They can then be formalized as temporal logic formulas that are given as input to the model checker. Temporal logic is an extension of propositional logic with temporal modalities used to describe the behavior of reactive systems [2]. The formalization of these properties interacts with modeling where the level of abstraction and the system/environment interface as well as component interfaces are designed to support requirement specification so that requirements can be formalized as temporal logic properties on these interfaces.

When the system has been modeled and the properties to be verified have been formalized using temporal logic properties, the actual model checking is performed. Running the model checking tool is often the most straightforward part. If the model does not satisfy the verified properties, the model checker gives a counter-example execution of the model showing the sequence of state transitions that leads to the violation of the property. Interpreting the results produced by the tool is in principle quite simple if the created model supports traceability, i.e. supports interpretation of the behaviors of the model as behaviors of the original design. However, counter-example executions demonstrating that a specification is violated by the system design can be quite complicated and further tool support may be needed to illustrate the underlying erroneous system behavior detected. By analyzing the counter-example the user can decide whether the violation was caused by an actual error in the system, an error made while modeling the system, inaccurately written natural language requirement, inaccurately written temporal logic formula or

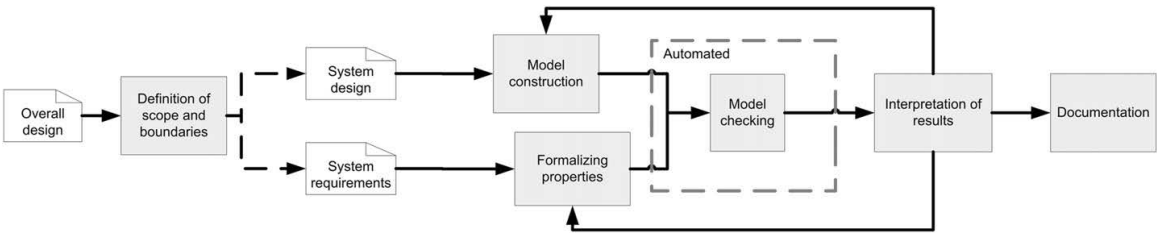


Fig. 1. Model checking process.

whether it was due to over-simplification of the model. If the counter-example is not caused by an error originating from the system design, the model or the properties to be checked must be refined and model checking is performed again. Thus, model checking is an iterative process that educates the modeler as the model checking process goes on.

Currently, while using model checking for verifying safety-critical I&C system designs, only the model checking phase is automated because it is done with a model checking tool. In our current workflow all other phases of the process are performed manually and, thus, may be prone to human error. However, the iterative nature of the model checking process remarkably improves the quality of models because the errors made in the modeling phase are usually revealed when the counter-examples are interpreted. See [23] for an approach to microprocessor verification that fully automates the model building process for a hardware description language. The first step in this direction for I&C systems would require a machine readable system description language with formally defined semantics.

3. Related work

The model checking concept was proposed in the early 1980s but it was in the 1990s when new symbolic model checking techniques led to novel tools such as Symbolic Model Verifier (SMV) and a breakthrough in scalability [7]. Model checking first proved effective in hardware verification [7–9]. Currently, for example most major microprocessor manufacturers use model checking techniques to verify their processor designs [8,9]. More recently, the focus has shifted to software model checking [10]. One of the main success stories has been the Windows C code device driver verification tool SLAM [11] which has been at production use at Microsoft for a number of years. All Microsoft-developed device drivers for Windows 7 have undergone model checking for proper Windows device driver API usage before their release [43]. Model checking has been used for verifying data communications protocols [12,13], for understanding human–computer interaction problems in avionics [14] and for analyzing real-time controllers [15], to name a few example application domains.

Model checking has also been utilized in the analysis of safety-critical systems. In aviation, model checking has been applied in the formal verification of flight critical software for identification of design errors early in the lifecycle [16]. In the safety analysis of an embedded control system, model checking was used to prove the functional correctness of the system, and the reliability of the system was assessed using fault tree analysis [17].

In the nuclear context, [18] presents an approach where model checking is combined with fault tree analysis to provide formal, automated and qualitative assistance to informal and quantitative risk analysis. However, the closest work to ours is the work on model checking Korean nuclear power plant automation systems [19–21]. In their work, they used model checking to analyze Programmable Logic Controllers that had been programmed as function block diagrams (FBD). They also checked the equivalence of FBD designs using model checking. They found several critical logic errors in the FBDs of reactor shutdown systems.

4. Model checking critical I&C designs

We have studied the applicability of model checking for safety analysis of various kinds of critical I&C designs by analyzing several industrial cases. The analyzed cases include an emergency cooling system of a nuclear reactor [39], an industrial arc protection system [39], a changeover switching unit for a busbar [38], an emergency diesel generator control system [33], a stepwise shutdown system [32], and embedded control software of an uninterruptible power supply (UPS) [34]. In what follows, we introduce methodology for modeling these systems, and shortly present two of the case studies we have analyzed. The cases show how time-related erroneous system behavior easily escapes the analyses done with traditional methods but can be discovered with model checking.

4.1. Model checking methodology

Several of the case designs were given as function block diagrams. We have created systematic methodology to model function block diagrams for two model checking tools: NuSMV and UPPAAL.

NuSMV models are basically collections of variable declarations and assignments that define the valid initial states and transition relations for these variables. In addition, the modeling language uses, e.g. simple data structures, macro definitions, and module hierarchies.

In NuSMV, our modeling approach is based on a collection of function block modules. These reusable modules are used to build up the functionalities of the system. Since a module can contain instances of other modules, our models employ a structural hierarchy. The model is typically divided into several modules according to different top-level functionalities. The reusable function block modules reduce the modeling effort, and their function can easily be separately verified. The modularity on the top level allows redundant structures to be modeled more efficiently. Because of the modular structure of our models it is also easier to create abstract models in which only a part of the system functionality is examined. These abstractions are sometimes necessary as the size of the model increases.

Because NuSMV does not support continuous time, the time-dependent components are modeled to operate in discrete time steps of fixed length. During each time step, first the inputs of the functional blocks are sampled and then the outputs are updated.

When using model checking to verify whether a system design satisfies a specification, this is done against an environment model describing how the environment and the system interact. Both in the UPPAAL and NuSMV modeling, no assumptions about the environment of the system are made. In our approach simple environment models that allow the environment to behave quite freely and independently of the system under verification are used wherever possible. This leads to safe model checking results: if the model checking tool determines that the system model satisfies the specification in this liberal environment model, then the system model will satisfy the specification in all more restricted environments and thus the correctness of system behavior is not based on strict assumptions on the environment behavior.

In UPPAAL a model is built up from several timed automata that synchronize with each other through channels. A timed automaton is a finite state automaton extended with real-valued clock variables. The edges of a timed automaton can have guard constraints which have to be true in order for the transition to be enabled. Processes send synchronization events to a channel to inform other automata about important events, e.g. the change of an input signal state. For further information on timed automata, see for instance [37].

When modeling function block diagrams in UPPAAL, the same principles mentioned above are followed. Each function block type is translated into a reusable timed automaton. In addition we use one separate automaton for input sampling in the model. We have also developed methods that allow the discovery of erroneous asynchronous behavior in the UPPAAL, see [33].

In NuSMV modeling we make the assumption that the modules of the system function synchronously (the whole system is an entity that samples the inputs and produces the outputs within one clock cycle). In UPPAAL, it is possible to examine other asynchronous subsystem behavior as well.

The level of abstraction that we use in our methodology is highly dependent on the available documentation. Typically, we intend to focus on the logical function of a system in detail, and leave out the hardware and system level aspects. When making abstractions of the system, we opt for over-approximations so that the model has more behaviors than the actual system.

4.2. Stepwise shutdown system

The stepwise shutdown system is a safety-related system used for stepwise control of the process towards the normal operating state in case of disturbances. The purpose of the system is to reduce the possibility of the process entering an undesired state where the more complicated actual shutdown function is required.

The stepwise shutdown system was modeled based on the system design documentation. The input documentation used for modeling was from the early phases of the development lifecycle. The design was based on function block diagrams, and the methodology as described before was applied. The system was modeled using both NuSMV and UPPAAL.

A part of the stepwise shutdown system is illustrated in Fig. 2. In case of e.g. high temperature or over-pressure, the alarm signal is activated and the stepwise action is initiated. The process is first driven towards a safer state for a certain period (3 s) after which the shutdown action is inactive for another period (12 s). A check is then done to determine whether the conditions that triggered the shutdown are still valid. This cycle continues until a normal or safe state is reached, in which the alarm signal disappears. The period between the control commands can be shortened by activating a manual bypass command in the control room.

We compared the performance and applicability of NuSMV and UPPAAL model checking tools and utilized them in the model checking of the system design. The UPPAAL model for the abstracted design is presented in Fig. 3. A more detailed description of the model and the checked properties can be found in [32].

Both tools were successfully employed to verify several basic safety properties of the system, and were able to reveal the same hidden design error in the design. The found error violated the following natural language property: if alarm is set, then eventually the output of the system gets value 1. To get reasonable results, an additional condition was included in the checked property: the output of the 15 s time pulse block must be zero when the alarm is set. This additional condition ensures that only an alarm signal set after the inactive period is considered. This eliminates the spurious counter-examples where the alarm signal is first reset, then set and finally reset again during the 15 s period. In UPPAAL the property can be formalized using the temporal logic TCTL [36]. The property can be captured by the following TCTL formula:

$A\Box$ (((In.Alarm and Pulse15_out = = 0) imply $A\Diamond$ Pulse3.Out1).

This formula was expressed in the UPPAAL specification language with the special leads-to operator "→" as follows:

(In.Alarm and Pulse15_out = = 0)→Pulse3.Out1.

In NuSMV, the specifications were formalized with LTL [2] and the property was expressed in LTL as follows:

LTLSPEC G((ALARM & !pulse15S.BO)→F(Output)).

The discovered error results in that the output of the system freezes at a value of zero. The error is caused by a mistimed operator action combined with the misuse of a time pulse block. The used time pulse blocks are triggered by a rising edge and all input changes during the pulse are ignored. The error occurs when the manual bypass is activated during the 3 s control. Subsequently, the 15 s block is reset, which creates a new rising edge for the 3 s pulse block when the alarm is set, but because the pulse is already set the rising edge is ignored. Since the 15 s pulse block is reset a new rising edge to trigger the 3 s pulse block cannot be created as long as the alarm is set.

Model checking times of the NuSMV model ranged between 0.3 s and 30 s depending on the used time step (10–1000 ms) and the verified property. The size of the state space of the most complicated scenario modeled with NuSMV was 1018. The computation times of the UPPAAL model were between 9 s and 20 s, being a bit longer than those of NuSMV on average. Even though the functionality of the system is quite simple and the number of inputs is low, the timing functions remarkably increase the size of the state space. In practice, it is not feasible to provide an exhaustive analysis of these kinds of systems with traditional methods.

In this case study the temporal logic specifications were typically derived from a collection of natural language requirements that could be based on regulatory requirements, laws or standards. However, these requirements seldom include all the necessary detailed requirements of a system. Also, the task of interpreting the high level regulatory requirements as precise temporal logic specifications is quite challenging. Requirements expressed in natural language are frequently ambiguous, or inaccurate. It is often necessary to split the high level requirements into several precise formal statements that together cover the high level requirement.

For instance, regarding the stepwise shutdown system, the Finnish Regulatory Guide on nuclear safety YVL 5.5 [40] specifies the following requirement "…that errors in data communication do not cause faulty functions or prevent the functioning of safety functions…" Similarly, the requirements specification of the system defines that the system shall fulfill the single failure criterion. In the stepwise shutdown system, these criteria were interpreted as follows: (i) a single failure in input signals shall not prevent the functioning of the system, and (ii) a single failure in input signals shall not spuriously trigger the system. These two interpretations were then translated into temporal logic specifications.

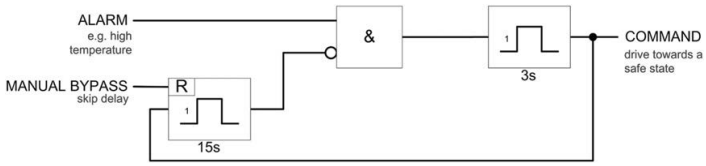


Fig. 2. Stepwise shutdown system.

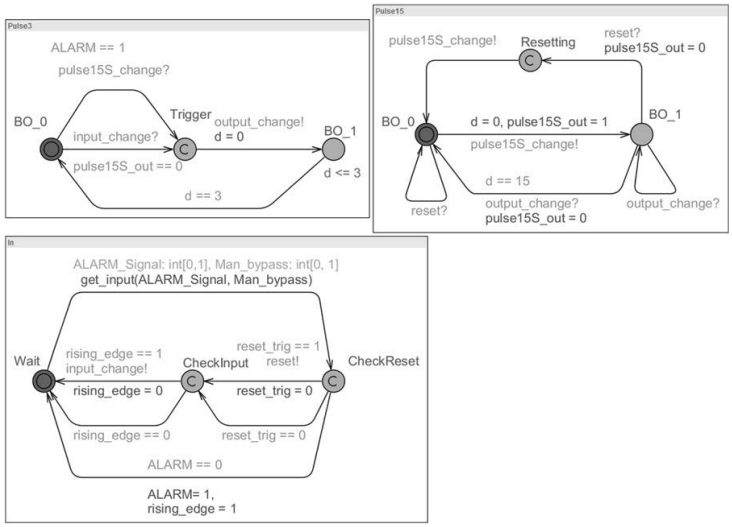


Fig. 3. UPPAAL model of the stepwise shutdown system.

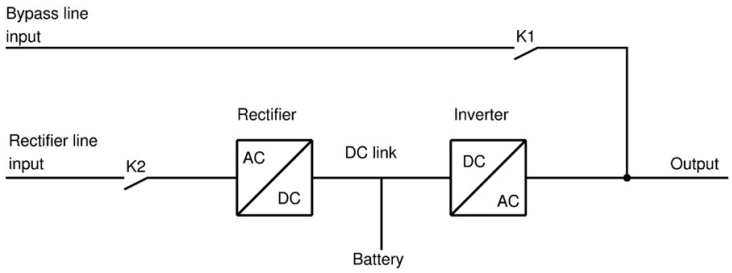


Fig. 4. Simplified design diagram of the UPS.

4.3. UPS control software

An uninterruptible power supply is a device used to provide back-up power in case of power failures and to protect connected equipment from various power disturbances. UPS devices are used in safety-critical systems to guarantee an uninterrupted power supply to the devices that must be continuously available to ensure the safety of the system.

We applied model checking to verify the control software of an industrial UPS [34]. The UPS was not specifically designed for nuclear applications but a similar device could also be used in a nuclear plant. The case study differs from the stepwise shutdown system in many aspects. The software of the system was not based on a function block design. The model was created by reverse-engineering the control software implementation that was already employed in the devices. The modeling task required interviewing system developers as well. Detailed requirement specification of the system was not available, and the formalization of the temporal logic specifications was based on the statements of the developers. The analysis of the system could not be done using a totally non-restricted environment model as is usually preferred. The speed at which the input line voltage can possible rise had to be taken into account. In addition, the behavior of the model is highly dependent on the various operational delays in the electrical components. These parameters had to be defined and verified with the developers of the system as well. A simplified design diagram of the UPS is presented in Fig. 4. The UPS feeds power to output either from a DC link through an inverter or from a bypass input. The DC link is powered through a rectifier, which filters sudden voltage changes of the input power. It is possible to connect the inputs to separate supply sources or to a common supply source. Two switches are used to change between the two inputs: K1 in the bypass line and K2 before the rectifier to cut power from the DC link.

The UPS should provide uniform quality power to the output. For instance, it is not acceptable to feed overvoltage to the output. Therefore, the voltage of both input lines is constantly measured. If the rectifier line has overvoltage, the K2 switch is opened and the bypass line is connected by closing the K1 switch. On the other hand, if the bypass line has overvoltage, the K1 switch must be opened and the power is provided only by a battery connected to the DC link. If the inputs are connected to a common supply source, the UPS should not switch to the bypass line if overvoltage is detected on both inputs (i.e. symmetric input overvoltage).

An UPPAAL model of the system was built to investigate the operation of the UPS in different overvoltage situations. A simplified model of the system, presented in Fig. 5, consisted of four automata. One automaton was used to model the operation of the K1 switch. The DCLink and bypass automata modeled the relevant part of the control software that controls the K1 switch in overvoltage situations. The environment automaton captured the system environment, i.e. when the overvoltage threshold is exceeded in the UPS

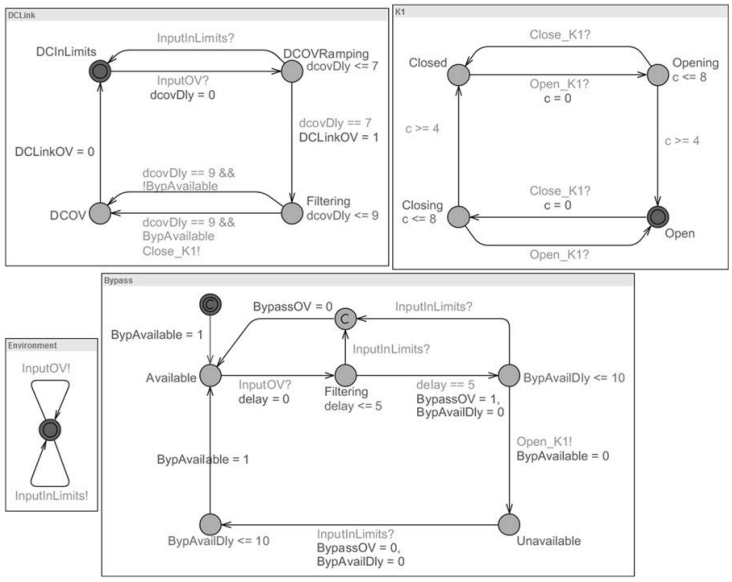


Fig. 5. UPPAAL model of the UPS system.

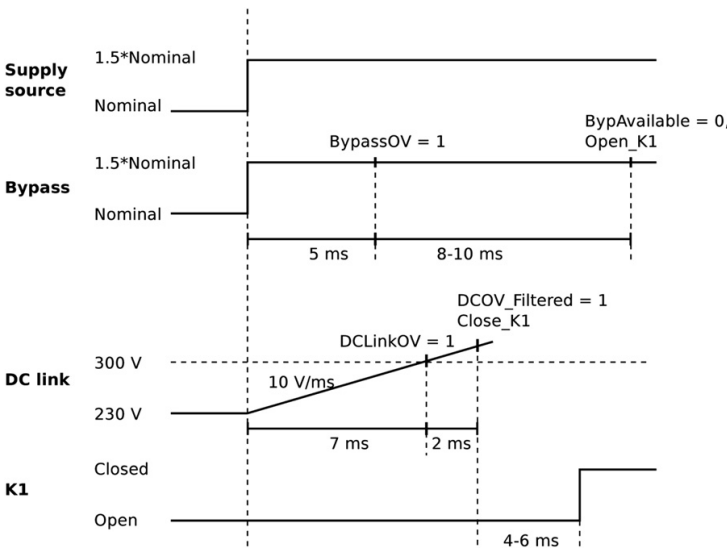


Fig. 6. Unwanted behavior revealed by model checking: overvoltage in the Bypass and K1 is closed.

inputs. For the purposes of simplicity, K2 was abstracted away from the model and it was assumed that K2 could open anytime. A more detailed description of the employed modeling approach and the checked properties can be found in [34].

The design criterion of the UPS is to feed power to the output as long as possible but it is not acceptable to feed overvoltage to the output. This requirement is clear but it is very difficult to test all possible event sequences. This difficulty stems from a range of timing issues that have to be taken into account:

1. The switches open and close with a delay of 4–8 ms.
2. Filtering the bypass voltage measurement takes 5 ms.
3. Filtering the DC link voltage measurement takes 2 ms.
4. The bypass voltage measurement is read by an interrupt handling routing that is called every 10 ms.
5. The rectifier delays the increase of voltage in the DC link measurement. The magnitude of the overvoltage in the input network affects the rate at which the voltage in the DC link increases.

An error was discovered in the system that causes overvoltage to be fed to the output. The error occurs in the symmetric overvoltage situation. In the UPPAAL model, this was modeled

with a common InputOV synchronization that was sent to both the DCLink and bypass automata by the environment automaton. The timings related to the error are presented in Fig. 6. If the voltage in the input network rises to a level 1.5 times the nominal voltage, it causes the voltage in the DC link to increase at a rate of 10 V/ms. It takes seven milliseconds for the DC link voltage to reach the overvoltage limit (DCLinkOV is set). The alarm bit (DCOV_Filtered) is set after measurements filtering. At this point, the K1 switch is commanded to close because the system is trying to switch to the bypass line. In the bypass line the overvoltage is recorded as soon as the measurement is filtered (BypassOV is set). However, the system only reacts (BypAvailable is reset) to the value when the interrupt handling routine is invoked. The routine gives K1 switch the command to open. The error occurs because the system may switch to bypass before the bypass line reacts to the overvoltage. The demonstrated erroneous behavior can occur only when the periodic interrupt handling routing is called between 13 and 15 ms after the beginning of the overvoltage situation.

The error was found by checking the property: "The UPS should not transfer to bypass while there is overvoltage in the bypass line". In UPPAAL specification language this behavior can be written as:

`A[ ] !(BypassOV && K1.Closed)`.

The specification states that there should never be a situation, in which there is overvoltage in the bypass and K1 is closed.

4.4. Typical errors discovered by model checking

Model checking has proven to be a useful tool for design verification in several application areas. The design errors that have remained unobserved by traditional testing and verification methods, but have been discovered by model checking, are in our experience typically infrequent scenarios where a few independent events occur during a short time frame.

Experience in utilizing model checking in several industrial cases has shown that hidden design errors are typically caused by the designer of the system not taking into account (1) mistimed manual actions, (2) events outside the actual logic, such as sensor, communication or hardware failures, (3) physical events occurring in an unexpected order or (4) simultaneity of several signals. In function block based designs it has been noted that the use of certain function blocks often facilitates the occurrence of design faults. It might be that the functionality of these function blocks is difficult for the designer to comprehend intuitively. Thus the designer may incorrectly determine that the abovementioned issues are not possible in the design.

The most problematic function blocks based on our experience are rising edge triggered time pulse blocks, set–reset flip-flops and modified function blocks that implement non-standard functionality. The time pulse block is problematic since it is only triggered by a rising edge and all input changes during the pulse are ignored. Thus, if the input remains set or is set during the pulse, the output of the time pulse block can freeze to zero. The problem concerning the set–reset flip-flop is that it is difficult to intuitively understand how changing the prioritization or the initial value of the inner memory affects system behavior. Changing the type of a problematic flip-flop may cause a different error that may be even more severe than the original during some other rare event. The problem with non-standard function blocks is that a function block might not act as expected in all situations, e.g. the non-standard functionality may mask other actions of the function block.

Errors like the ones described are hard to detect by simulation or testing because the test designer must think of all the possible system behaviors when the test plan is made. On the other hand, the advantage of model checking is that it suffices to describe the most important properties formally, i.e. the test designer does not have to think of all possible behaviors himself, as the model checking tool does this for him. The model checking tool investigates all possible behaviors and if errors are found, automatically returns a counter-example showing the state transitions leading to the erroneous behavior. Another advantage of model checking is that it is possible to prove negative properties, i.e. what the system should not do, which is hard to show by testing.

Based on our case studies thus far, we recommend that more attention is put into testing boundary values of systems, manual actions, and hardware failures. The test cases should focus more on covering various timed sequences of inputs rather than only increasing combinatorial coverage. In function block based designs, parts using non-standard function blocks should be analyzed more rigorously.

5. Discussion

As we have been able to reveal hidden design errors in our case studies, model checking has been increasingly used as part of the verification and inspection process in the Finnish nuclear sector. The power companies and the nuclear safety authority have been supporting the development and application of model checking methods right from the beginning of our work. Integrating model checking more closely to the development, licensing and qualification processes of systems could be very beneficial. It would lead to verifying nuclear safety systems in a more formal fashion, and the potential faults in the systems could be found earlier in the system lifecycle.

Despite the clearly demonstrated benefits of using model checking, it is still not widely in use for verification of safety-critical systems. The reasons for this are: (1) amount of resources needed to apply model checking, (2) lack of confidence in the model checking results, and (3) lack of support in relevant standards and regulations. In what follows, we discuss these issues and give recommendations to improve the situation.

5.1. Model checking effort

In many cases model checking is not used simply because of the assumed high volume of human resources needed for the adoption and use of the method. Compared to effort needed for the industrial deployment of some other formal methods, however, model checking requires relatively little training and human resources. In model checking, the modeler does not have to understand how the model checking tool itself works. It suffices to be able to generate a model of the system. If methodology exists for modeling a certain type of system, a new model can be created quite quickly. Based on our experience, a moderate sized function block based design can be modeled and analyzed within five working days. The required effort naturally depends on the complexity of the design. Also, if a model already exists, it is easy to modify it according to changes made in the system, or analyze new properties of the system. However, in cases where conventional modeling methodology is not readily available, such as in the UPS case study, the required modeling effort may be substantial.

To improve the cost-effectiveness of the method, we recommend improving the model checking process by automated model creation. This reduces the manual effort required for modeling and cuts the costs of correcting errors that would otherwise be found later in the product lifecycle. The approach would also significantly reduce the number of human errors in modeling. Integrating a

model checking tool into the semantic modeling and simulation platform Simantics is one step in this direction [35]. The semantic approach of Simantics enables automatic conversion of function block based design models to the format required by a model checking tool.

5.2. Increasing confidence in the model checking results

The model checking method is not as fully developed as traditional V&V methods, and the correctness of the model, the temporal specifications checked and the tool used can be questioned if these issues are not addressed. This lack of confidence in the model checking results limits the use of the results in the licensing and qualification processes of systems.

More confidence in the results of model checking can be achieved with a well-defined and documented process of using the method, as well as ensuring that no mistakes are made in the model checking process. This includes ensuring the correctness of the tool, the model, and the specifications.

When a property of a system does not hold, model checking tool produces a counter-example. It is then rather straightforward to validate the counter-example, by testing for example. Thus, the correctness of the model checking result is easy to ensure when a fault is discovered. However, model checking tools do not provide counter-examples when a property is satisfied and the model is considered to function according to the specification. This is problematic, because it is difficult to be assured of the correctness of the claim. An error in the model, model checking tool or specification could cause an erroneous response from the tool. Absence of errors is difficult to show, but all issues concerning the correctness of the model checking result should be addressed in some way.

Model checking is a method that is capable of exhaustively analyzing whether a system is according to its specification. As it is with all verification methods, absolute certainty of the correctness of the system can never be achieved with model checking. However, the reliability of the result can be increased through various means up to a certain point.

It is obvious that reviewing (inspecting) the model and the specification is beneficial, but reviews cannot prove the correctness of systems. In order to increase confidence in the model, we recommend model checking additional properties of the model, and verifying that the outcomes match the expectations of the modeler.

The temporal logic formulas checked must represent the original specifications of the system. Not only is it difficult to formalize specifications, but often the natural language specifications themselves are incomplete or ambiguous. Correct formalization of properties can be facilitated by training, and review work, as well as using temporal logic requirements debugging tools [26]. Other possible means of improvement are the use of specification patterns, the use of demonstrative tools and syntactic/semantic analysis of the formula.

Confidence in the tool itself can be supported by certification, reverse-engineering and cross-checking with another model checker [27]. The common position of nuclear regulators [24] also states that previous operational usage can be used to validate a tool. These approaches, however, are usually not feasible in many practical cases. In our experience, cross-checking the results with another model checker is the most feasible approach. There are attempts [28,29] to combine model checking with theorem proving. This approach can be used to automatically generate a deductive proof when a model checking tool reports that the specification holds. This proof could then be mechanically checked by either humans or simple proof checking algorithms. Even though theorem proving is time consuming and labor intensive and these methods are not yet in routine production use, the approach could be significant in the future due to the strengths discussed above.

Elaborate documentation increases confidence in correct application of model checking. We recommend that in order to make the model checking results more credible, more effort should be put in the justification of the selected abstraction level, and the consequences of the selected assumptions in the model. Discussion of the errors that are left outside the analysis should also be included. Furthermore, the measures taken to ensure correctness of the model checking process should be addressed in the documentation.

The common position of nuclear regulators [24] has similar guidelines for formal methods. It requires that the system boundaries are defined and selection of methods is justified. The common position also requires the formalisms and methods to have been previously used in a related application, and requires a syntactical verification of the formal descriptions. These matters should be part of the documentation of model checking results to make them more credible for the safety demonstration.

Another recommendable addition to the documentation of the results is suggested in [25], which proposes that a positive reply of the model checker be accompanied by two additional simulated paths: an interesting witness path, which attempts to demonstrate that the specification can be satisfied non-vacuously, and a non-interesting witness path, which is a path that is not covered by the specification. These additions can demonstrate the meaning of the specification in the model, and the behavior that is beyond the scope of the specification.

5.3. Requirements in standards

In the nuclear domain, IEC 60880 [30] mentions model checking as a complementary method for software verification. The more generic IEC 61508 [31] recommends the use of model-based testing, semi-formal and formal methods based on the intended safety integrity level. The use of software verification tools to support the verification process of safety-critical software is generally recommended (but not required) in software standards. The reason for such mild requirements for using formal methods is probably that until recently their use has not been feasible for industrial sized systems.

In our case studies, the found errors were not discovered prior to model checking. Thus there is definite advantage in the use of formal methods. However, it is difficult to mention any specific requirements that should be added to the current safety standards based on our case studies. The reason is that the standards are so generic by their nature. Because of the conservative view in standards, the introduction of new techniques is difficult in general. However, now that the potential of model checking has been demonstrated, its use should be somehow encouraged.

Our opinion is that the integration of model checking techniques to the current certification process can be enhanced by using the safety case approach, allowing model checking as an alternative method for compliance in the standards, and by adding requirements on how the methods should be used in a systematic manner.

The safety case approach (see e.g. [42]) for certification enables the use of formal methods based evidence, since the claim–argument–evidence structure of the safety case is not dictated by regulation. If the standards-based approach for certification is used, the standard could encourage the use of formal methods by relieving the requirements set for other verification methods if a particular property is shown by e.g. model checking. In the aviation domain standard DO-178B [41], such alternative methods for compliance are allowed for some objectives. Finally, the common position of nuclear regulators [24] gives more specific practical requirements on how formal methods should be utilized, but these requirements are not obligatory. It would be useful if the key standards in safety-critical domains

would give similar guidance on a systematic and well-defined process of using formal methods.

6. Conclusions

Complex digital systems are increasingly being used to implement safety-critical systems. This presents new challenges for verification. Systems with multiple inputs, memories inside the system design and time-dependent behavior are impossible to test or simulate exhaustively. To address this problem, model checking has been applied to analyze several safety-critical industrial systems.

This work shows that model checking is an applicable and scalable method for the analysis of medium-sized industrial systems. Two case studies were presented, in which an error was detected. The scalability of model checking derives mostly from efficient algorithms but modeling methodology plays a big part in achieving sufficient scalability and thus applicability. The main contribution of the paper is the development of systematic methodology for modeling function block based system designs using two model checking tools: NuSMV and UPPAAL. Our modular methods for function block based models decrease the modeling effort. In addition, the abstractions made on the environment models simplify the verification of the system, while resulting in safe model checking results.

There are clear benefits in using model checking. Our experience shows that the method can effectively be used to find hidden erroneous behaviors that are typically not taken into account in system design or test specification. Based on the found errors, we recommend that more attention is put into testing various timed sequences of inputs rather than increasing combinatorial coverage in tests. The test cases should focus more on boundary values of systems, manual actions, and hardware failures.

Despite the clearly demonstrated benefits of using model checking, it is still not widely in use for verification of safety-critical systems. In our opinion, this is due to the seemingly high effort in the adoption and use of the method, and the lack of confidence in the results of the analysis.

In order to reduce the effort required to perform model checking, we recommend improving the model checking process by automated model creation. The measures taken to ensure correctness of the used tools, modeling abstractions, the checked specifications, and the model checking process should be systematically addressed in the documentation.

Regarding the requirement in standards, the current standards-based certification process could be enhanced by allowing model checking as an alternative method for compliance in the standards, and by adding requirements on how the methods should be used in a systematic manner. An alternative certification approach based on a safety case would also allow relatively novel methods to be used more extensively.

To further improve the applicability of the method, we plan to develop modeling methodology for large distributed systems. Our future research subjects include applying compositional model checking techniques and assume-guarantee reasoning to enable the analysis of large complex systems, developing more systematic methodology for asynchronous distributed systems and expanding our models modularly by adding fault models to them.

References

[1] Kaivola R, Ghughal R, Narasimhan N, Telfer A, Whittemore J, Pandav S, et al. Replacing testing with formal verification in Intel CoreTM i7 processor execution engine validation. CAV 2009: Lecture Notes in Computer Science 2009;5643:414–29.
[2] Clarke E, Grumberg O, Peled D. Model checking. The MIT Press; 1999.
[3] Baier C, Katoen J-P. Principles of model checking. MIT Press; 2008.
[4] Cimatti A, Clarke E, Giunchiglia E, Giunchiglia F, Pistore M, Roveri M., et al. NuSMV2: an OpenSource tool for symbolic model checking. In: Proceedings of the international conference on computer-aided verification. Copenhagen, Denmark; 2002.
[5] Cavada R, Cimatti A, Jochim CA, Keighren G, Olivetti E., Pistore M, et al. NuSMV 2.5 user manual. FBK-IRST 2010 [see also 〈http://nusmv.fbk.eu/〉].
[6] Behrmann G, David A, Larsen KG. A tutorial on UPPAAL. In: Bernardo M, Corradini F, editors. Formal methods for the design of real-time systems (revised lectures), of lecture notes in computer science, vol. 3185; 2004. p. 200–37 [see also 〈http://www.uppaal.com/〉].
[7] Burch J, Clarke E, McMillan K, Dill D, Hwang L. Symbolic model checking: 10^20 states and beyond. Information and Computation 1992;98(2):142–70.
[8] Fix L. Fifteen years of formal property verification in Intel. 25 Years of model checking. Lecture Notes in Computer Science 2008;5000:139–44.
[9] Ben-David S, Eisner C, Geist D, Wolfsthal Y. Model checking at IBM. Formal Methods in System Design 2003;22(2):101–8.
[10] Jhala R, Majumdar R. Software model checking. ACM Computing Surveys 2009;41:4.
[11] Ball T, Cook B, Levin V, Rajamani S. SLAM and static driver verifier: technology transfer of formal methods inside Microsoft. In: Proceedings of the fourth international conference on integrated formal methods. Lecture notes in computer science, vol. 2999; 2004. p. 1–20.
[12] Holzmann GJ, Smith MH. Automating software feature verification. Bell Labs Technical Journal (BELL) 2000;5(2):72–87.
[13] David A, Yi W. Modelling and analysis of a commercial field bus protocol. In: Proceedings of the 12th Euromicro conference on real time systems; 2000. p. 165–72.
[14] Rushby J. Using model checking to help discover mode confusions and other automation surprises. Reliability Engineering and System Safety 2002;75(2): 167–177.
[15] Bengtsson J, Griffioen WOD, Kristoffersen KJ, Larsen KG, Larsson F, Pettersson P, et al. Automated verification of an audio-control protocol using UPPAAL. Journal of Logic and Algebraic Programming 2002;52–53:163–81.
[16] Miller SP, Anderson EA, Wagner LG, Whalen MW, Heimdahl MPE. Formal verification of flight critical software. In: Proceedings of the AIAA guidance, navigation and control conference and exhibit. San Francisco; August 15–18, 2005.
[17] Ortmeier F, Schellhorn G, Thums A, Reif W, Hering B, Trappschuh H. Safety analysis of the height control system for the Elbtunnel. Reliability Engineering and System Safety 2003:259–68 [safety, reliability and security of industrial computer systems].
[18] Koh KY, Seong PH. SACS2: a dynamic and formal approach to safety analysis for complex safety critical system. In: Proceedings of the sixth American nuclear society international topical meeting on nuclear plant instrumentation, control, and human–machine interface technologies. Knoxville, Tennessee; April 2009.
[19] Yoo J, Cha SD, Jee EA. Verification framework for FBD based software in nuclear power plants. In: Proceedings of the fifteenth Asia-Pacific Software Engineering Conference; 2008. p. 385–92.
[20] Yoo J, Jee E, Cha SD. Formal modeling and verification of safety-critical software. IEEE Software 2009;26(3):42–9.
[21] Koo SR, Seong PH, Yoo J, Cha SD, Youn C, Han H. NuSEE: an integrated environment of software specification and V&V for PLC based safety-critical systems. Nuclear Engineering and Technology 2006;38(3).
[22] Holzmann GJ. The SPIN model checker: primer and reference manual. Addison-Wesley; 2003.
[23] Hunt Jr. WA, Swords S. Centaur technology media unit verification. CAV 2009: Lecture Notes in Computer Science 2009;5643:353–67.
[24] EUR 19265 Rev. 2010. Licensing of safety critical software for nuclear reactors. Common position of seven European nuclear regulators and authorised technical support organisations.
[25] Chockler H, Kupferman O, Vardi MY. Coverage metrics for temporal logic model checking. Formal Methods in System Design 2006;28(3):189–212.
[26] Bloem R, Cimatti A, Greimel K, Hofferek G, Könighofer R, Roveri M, et al. RATSY—a new requirements analysis tool with synthesis. CAV 2010: Lecture Notes in Computer Science 2010;6174:425–9.
[27] IAEA. Software for computer based systems important to safety in nuclear power plants. IAEA safety guide NS-G-1.1; 2000.
[28] Namjoshi KS. Certifying model checkers. Lecture Notes in Computer Science 2001;2102:2–13.
[29] Peled D, Zuck L. From model checking to a temporal proof. In: SPIN '01: proceedings of the eighth international SPIN workshop on model checking of software; 2001. p. 1–14.
[30] International Electrotechnical Commission. IEC 60880. Nuclear power plants – I&C systems important to safety – software aspects for computer based performing category A functions; 2006.
[31] International Electrotechnical Commission. IEC 61508. Functional safety of electrical/electronic/programmable electronic safety-related systems; 2009.
[32] Björkman K, Frits J, Valkonen J, Lahtinen J, Heljanko K, Niemelä I, et al. Verification of safety logic designs by model checking. In: Proceedings of the sixth American nuclear society international topical meeting on nuclear plant instrumentation, control, and human–machine interface technologies. Knoxville, Tennessee; April 2009. ISBN: 978-0-89448-067-6.
[33] Lahtinen J, Björkman K, Valkonen J, Frits J, Niemelä I. Analysis of an emergency diesel generator control system by compositional model

checking—MODSAFE 2010 work report. VTT working papers 156. Espoo: VTT Technical Research Centre of Finland; 2010 [see also 〈http://www.vtt.fi/inf/pdf/workingpapers/2010/W156.pdf〉].
[34] Frits J. Model checking embedded control software. Research report TKK-ICS-R28. Espoo, Finland: Aalto University School of Science and Technology, Department of Information and Computer Science; March 2010.
[35] Pakonen A, Lahtinen J, Kuutti V-P, Karhela T. Integrating model checking with safety-critical I&C software design. In: Proceedings of the seventh international topical meeting nuclear plant instrumentation, control and human–machine interface technologies. Las Vegas (NV); 7–11 November 2010.
[36] Alur R, Courcoubetis C, Dill DL. Model checking for real-time systems. In: Proceedings of the fifth annual IEEE symposium on logic in computer science. IEEE Computer Society Press: Philadelphia (PA); 4–7 June 1990. p. 414–25.
[37] Alur R, Dill DL. A theory of timed automata. Theoretical Computer Science 1994;126(2):183–235.
[38] Björkman K, Valkonen J, Ranta J. Verification of automated changeover switching unit by model checking. In: Proceedings of the seventh international topical meeting on nuclear plant instrumentation, control and human–machine interface technologies (NPIC&HMIT 2010). Las Vegas (NV); November 7–11, 2010. p. 1719–28.
[39] Valkonen J, Koskimies M, Pettersson V, Heljanko K, Holmberg J-E, Niemelä I, et al. Formal verification of safety I&C system designs: two nuclear power plant related applications. Enlarged Halden programme group meeting. In: Proceedings of man–technology–organisation session. Loen, Norway; 18–23 May, 2008.
[40] STUK Regulatory Guides on Nuclear Safety. Guide YVL 5.5, instrumentation systems and components at nuclear facilities. Helsinki: Radiation and Nuclear Safety Authority; 2002.
[41] Requirements and technical concepts for aviation, Washington, DC. DO-178B: software considerations in airborne systems and equipment certification; December 1992. Also issued as EUROCAE ED-12B in Europe.
[42] Kelly TP, McDermid JA. Safety case construction and reuse using patterns. In: Daniel P, editor. SAFECOMP97: the 16th international conference on computer safety, reliability and security. York, UK: Springer; 7–10 September 1997. p. 55–69.
[43] Ball T, Levin V, Rajamani SK. A decade of software model checking with SLAM. Communications of the ACM 2011;54(7):68–76.

Publication II

Jussi Lahtinen, Kim Björkman, Janne Valkonen, Ilkka Niemelä. Emergency diesel generator control system verification by model checking and compositional minimization. In *8th Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS 2012)*, Znojmo, Czech Republic. Antonín Kučera, Thomas A. Henzinger, Jaroslav Nešetřil, Tomás Vojnar and David Antoš (Eds), p. 49 – 60, NOVPRESS 2012.

Emergency Diesel Generator Control System Verification by Model Checking and Compositional Minimization

Jussi Lahtinen, Kim Björkman, Janne Valkonen, and Ilkka Niemelä

VTT Technical Research Centre of Finland, Systems Research, P.O. Box 1000, FI-02044 Espoo, Finland.
Department of Information and Computer Science, School of Science, Aalto University, P.O. Box 15400, FI-00076 Aalto, Finland.
{jussi.lahtinen,kim.bjorkman,janne.valkonen}@vtt.fi
ilkka.niemela@aalto.fi

Abstract. Digital instrumentation and control (I&C) systems containing programmable logic controllers are challenging to verify. They enable complicated control functions and the state spaces (number of distinct values of inputs, outputs and internal memory) of the designs easily become too large for comprehensive manual inspection. Model checking is a formal method that can be used for verifying that systems have been correctly designed. A number of efficient model checking systems are available which provide analysis tools that are able to determine automatically whether a given state machine model satisfies the desired safety properties. However, model checking of large complex systems is often quite infeasible. In this paper, we present a compositional minimization technique for abstracting large modular function block based systems. We have applied the abstraction technique to the verification of a safety-critical emergency diesel generator control system. The system is so large that the non-abstract model could not be model checked within reasonable resources. Using the abstraction technique we managed to verify several universal properties of the system and were able to discover errors in the system designs. The abstraction technique is intended as a basis for an iterative abstraction refinement framework.

Key words: model checking, verification, validation, compositional minimization, diesel generator

1 Introduction

Verification of digital instrumentation and control (I&C) systems is challenging because programmable logic controllers enable complicated control functions and the state spaces (number of distinct values of inputs, outputs and internal memory) of the designs easily become too large for comprehensive manual inspection. Design verification is a key task in the design flow, because it can eliminate tricky design errors which are hard to detect later in the development process and are

very expensive to repair, often leading to a major redesign and reimplementation cycle. Typically, verification and validation (V&V) activities rely heavily on subjective evaluation, which covers only a limited part of the possible behaviours of the system, and therefore more rigorous formal methods are required.

Model checking [1] is a formal method that can be used for verifying the correctness of system designs. It has been used in verifying the correct behaviour of e.g. hardware and microprocessor designs, data communications protocols and operating system device drivers. A number of efficient model checking systems are available which provide analysis tools that are able to automatically determine whether a given state machine model satisfies given specifications. Model checking can also handle delays and other time-related operations, which are crucial in safety I&C systems and challenging to design and verify.

Model checking has been successfully applied to the verification of individual nuclear domain safety I&C systems, see e.g. [2]. In the nuclear domain, however, it is common to cope with hardware failures by implementing several diverse systems that execute the same physical function using different design, software and/or hardware. Because of this, it may be necessary to examine these systems simultaneously, because they might influence the same physical parameters. This requirement leads to very large models that are too complex to model check within reasonable resources.

This paper introduces the development of a compositional verification approach for model checking large system designs. The approach is utilized in the analysis of a case study concerning the control system of an emergency diesel generator. In our technique the system is divided into modules. Based on the specification, some system functionality may be irrelevant and can be left out of the model. Abstractions of the model are created by replacing a subset of the modules with non-deterministic interface modules similar to the compositional minimization approach [3]. A more detailed description of the work can be found in a technical report [4].

The rest of the paper is structured as follows. Section 2 introduces related work. Section 3 provides background information on model checking methodology. Section 4 describes the emergency diesel generator control system and some of its main requirements. Section 5 introduces compositional minimization as an approach for large modular function block based systems. Finally, Section 6 sums up the results and findings concerning the emergency diesel generator case, and Section 7 concludes the paper.

2 Related Work

The general idea in compositional verification is to break down the specification into several specifications describing the behaviour of individual components or modules of that system. Checking these local specifications is usually more feasible, and if the conjunction of the local properties implies the original specification, it is possible to deduce that the entire system satisfies the specification

as well. Many of the techniques [5, 3] require that the system is composed of interconnected modules.

Many approaches to compositional verification exist. These techniques include compositional minimization [3], assume-guarantee reasoning [6, 7] (including circular reasoning techniques [8]), partitioned transition relations [9] and lazy parallel composition [5].

In compositional minimization [3] the system is abstracted using reduced versions of some of the system's modules. The idea behind the compositional minimization is that not every part of the logic is necessarily required in order to verify a property. The reduced modules, or 'interface modules', are abstracted away from their intrinsic functionality, so that the modules' behaviour is completely non-deterministic, making the verification significantly more efficient. In this work we have applied the concept of compositional minimization to the analysis of function block diagrams by model checking. We have also developed semi-interface modules that preserve some of the modules' functionality.

3 Model Checking

Model checking [1, 10, 11] is a computer-aided verification method developed to formally verify the correct functioning of a system design model by examining all of its possible behaviours. The models used in model checking are quite similar to those used in simulation, as basically the model must describe the behaviour of the system design for all sequences of inputs. However, unlike simulation, model checkers examine the behaviour of the system design with all input sequences and compare it with the system specification. In model checking, at least in principle, the analysis can be fully automated with computer-aided tools. The specification is expressed in a suitable language, temporal logics being a prime example, describing the permitted behaviours of a system. Given a model and a specification as inputs, a model checking algorithm determines whether the system has violated its specification. If none of the behaviours of the system violate the given specification, the (model of the) system is correct. Otherwise, the model checker will automatically give a counter-example execution of the system demonstrating how the specification has been violated.

We have used the model checker NuSMV [12, 13], which was originally designed for hardware model checking. NuSMV is a state-of-the-art symbolic model checker that supports synchronous state machine models where the real-time behaviour must be modelled using discrete time steps. NuSMV supports model checking using both Linear Temporal Logic (LTL) and Computation Tree Logic (CTL) [1] making it quite flexible in expressing design specifications. The model checking algorithms employed in this work are based on symbolically representing and exploring the state space of the system by using Binary Decision Diagrams (BDDs) [14, 15]. In addition, SAT-based (Propositional Satisfiability) bounded model checking [16] is also supported by NuSMV [17] for finding bugs in larger designs.

4 Description of the Emergency Diesel Generator Control System

4.1 Emergency Diesel Generator Control

The case study is based on high-level design documentation of the control system that does not take into account the redundant implementations of the system and related voting logic that will be realised in the final system. This work focuses on the control logic of the diesel generator system that is represented as function block diagrams.

The purpose of the emergency diesel system is to provide reserve power in case the primary power supply fails. Emergency diesel systems are used in e.g. nuclear power plants where it is essential that safety systems are constantly connected to a power supply. In case of a black out or a disturbance in the main power supply, the diesel generators can be quickly turned on to keep the necessary devices available.

The inputs of the diesel control system logic include voltage and frequency measurements, operator commands, check-back signals and measurements of the conditions of the diesel generator. The outputs of the logic are control signals for the diesel generator, the breakers, cooling systems and load protection signals for several pumps and other devices, for which power can be supplied by the diesel generator.

In addition to the function block diagrams, parts of the system environment, i.e. the expected diesel functionality, the busbar and the related breakers are also included in the model. Figure 1 illustrates the high level architecture of the electrical connections of the system. One of the control system's objectives is to connect the busbar to an available power supply. Typically only one power source is connected to the busbar.

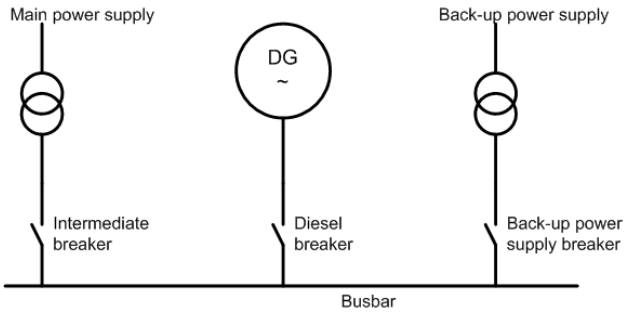


Fig. 1. The high level architecture of the electrical connections.

There is a large amount of logic related to the diesel generator control. The system analyzed here covers 10 different control functions, including functions

related to the activation of the diesel generators, operation of the diesel generators, protection of the diesel generators, and voltage and frequency regulation. In addition, some functions are diverse implementations of other functions. In most cases each function was modelled as a separate module. Due to non-disclosure agreements, the more detailed system descriptions are not presented.

4.2 System Requirements

Based on general system requirements, a list of detailed requirements was created and verified. However, the detailed requirements and temporal logic specifications are not covered in this paper due to confidentiality issues. The checked specifications were based on the following general system requirements:

1. If there is a reason to start the diesels, they will be started, and they will eventually feed power to the busbar.
2. When the diesels are started, a specified starting sequence is followed.
3. Loads are connected to the diesels according to a specified loading sequence.
4. The diesels take a few seconds to reach their operating speed. No loads should be connected to the generators during this time.
5. The connections to the busbar are controlled by several breakers as illustrated in Figure 1. The breakers should be operated in a safe fashion.
6. The control of the diesels is realised by several diverse systems. The prioritisation of the different systems' signals must be correct.
7. There should be no race conditions.

5 Abstracting Large Modular Models

It is possible that only a small part of the model is needed in order to verify a particular specification. It should be possible to easily leave some parts of the model outside examination. We achieve this through compositional minimization.

In our approach, function block diagrams are modelled as follows. A function block library is first written that consists of descriptions of all individual function blocks. In addition to Boolean operations, function blocks may realize more complicated functions using memories, such as flip-flops or various timers. The function blocks handle signal status information as well. Every signal carries a status bit that is used to mark the signal faulty. A faulty signal can affect the function block operation.

A function block diagram is modelled as a module that has a set of inputs, a set of outputs, and a set of function blocks that are instantiated from the function block library. A very simple example function block diagram is presented in Figure 2. It has three inputs, two outputs, and two AND function blocks.

The corresponding NuSMV model consists of a function block description of the AND function block, and the description of the function block diagram that instantiates the AND blocks. These two modules are presented below. In addition to the presented modules the main module of the model is needed.

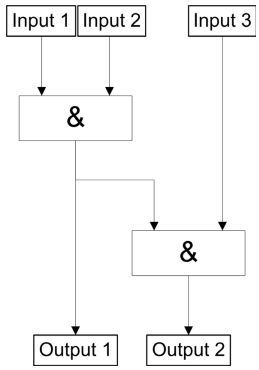


Fig. 2. An example function block diagram.

```
MODULE AND_FB(input1, input1_status, input2, input2_status)
DEFINE
output1 := input1 & input2;
output1_status := input1_status | input2_status;

MODULE example_diagram(input1, input1_status, input2,
input2_status, input3, input3_status)
VAR
AND1 : AND_FB(input1, input1_status, input2, input2_status);
AND2 : AND_FB(AND1.output1, AND1.output1_status,
input3, input3_status);
DEFINE
output1 := AND1.output1;
output2 := AND2.output1;
output1_status := AND1.output1_status;
output2_status := AND2.output1_status;
```

The function block diagrams are modelled as modules as described above. In our compositional minimization approach a subset of these modules are replaced with abstract versions. The abstract version can be either a completely non-deterministic full-interface module or a semi-interface module that contain parts of the original logic of the modules. An abstraction of the system model is created by selecting the abstraction-level of each module that describes a function block diagram. The full-interface abstraction and the semi-interface abstraction are described below.

A full-interface module contains no function blocks, and the outputs of the module are defined simply as free non-deterministic variables. Definition of the output variables as free variables is a complete over-approximation of the module, i.e. no restrictions on the behaviour of the module are set. A set of full-interface modules can be manually written in parallel with the model construction process.

Below is the full-interface module of the function block diagram in the running example.

```
MODULE example_diagram_interface(input1, input1_status, input2,
input2_status, input3, input3_status)
VAR
output1 : boolean;
output2 : boolean;
output1_status : boolean;
output2_status : boolean;
```

In semi-interface modules some function blocks are non-abstract, and some are approximated by 'interface function blocks'. Interface function blocks are dummy function blocks, the outputs of which are defined as free variables. For easy utilisation, an interface function block library was created.

The semi-interface module is created as follows. The abstraction parameters are a set of module outputs and the depth n of the abstraction. A program slicing method is then used to select a subset of function blocks that remain non-abstract. Other function blocks are transformed into interface function blocks. The program slicing method handles the function block diagram as a dependency graph, in which the signals between function blocks indicate one-way dependencies. Starting from the defined outputs, the program slicing method travels that dependency graph for n steps in a breadth-first manner, and selects the function blocks that are encountered. In our running example a semi-interface module could be created by replacing one of the AND blocks with an instance of an interface AND block. The model code for the interface AND block is below.

```
MODULE AND_FB_interface(input1, input1_status, input2,
input2_status)
VAR
output1 : boolean;
output1_status : boolean;
```

In our previous models, e.g. in [2], abstracting away from some functionality of a large system required a lot of manual effort. This work is reduced to selecting an abstraction level for each module. The model can then be generated using a simple computer script.

An example of a model configuration is shown in Figure 3. The model in the figures consists of five modules. Each of these modules has a set of function blocks. The function blocks are depicted as boxes, where grey boxes stand for non-abstract function blocks and white boxes stand for interface function blocks. The figure presents a configuration of one non-abstract module, one semi-interface module and three full-interface modules.

As the abstract model can be easily compiled, the remaining questions are: 1) How is the abstraction used to deduce whether a property is true in the accurate non-abstract model? 2) How can the correct configuration of modules be found, that both allows a property to be verified and is computationally manageable?

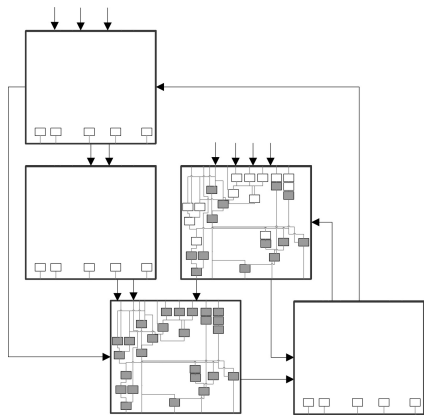


Fig. 3. A model configuration of one non-abstract module, one semi-interface module and three full-interface modules.

The first question can be answered when the examined system property is a universally quantified property, i.e. properties of LTL and ACTL*. If a universal property is true in a model configuration, in which some of the modules are replaced by interface modules, the same property is also true in the accurate non-abstract model. Interface modules are over-approximations, i.e. they have more behaviour than the regular non-abstracted modules. If a model configuration that uses these interface modules cannot produce undesired behaviour (violate the checked property), then the accurate model also cannot violate the property.

If a universal property is false in a model configuration containing interface modules, it should be determined whether the violation of the property is feasible in the non-abstract model, or if the violation is caused by the interface modules. If the property is violated because of the behaviour of the interface modules, a new refined configuration of interface modules and non-abstracted modules should be selected for further analysis.

Regarding the second question, an automatic method for the selection of the model configuration (similar to [18]) can be created, and thus the verification of large systems can in many cases be automated based on an iterative algorithm. In this work the model configuration is selected manually, but the abstraction technique is intended as a basis for an automatic iterative abstraction refinement technique. We have already experimented with various automatic methods but this work is not part of this paper.

6 Results and Findings

The emergency diesel generator control system was modelled with the NuSMV model checking tool. The NuSMV model consists of nine modules representing

the ten system functions (two functions were merged in to one module), a function block library and four modules representing the environment of the system. In addition, interface modules were created for each module (14 modules). For the purpose of using semi-interface modules, an interface function block library was also created. The NuSMV model has 2200 lines of code, including tests and comments. This does not include code for the interface modules (680 lines) or the interface function block library (230 lines).

Several properties of the diesel control system were analysed by model checking. The non-abstract NuSMV model is so complex that properties cannot be checked on the non-abstract model using a practical amount of time or memory. Fortunately, most of the examined properties in this case study can be verified by using model configurations in which only one or two non-abstract modules are present, and other modules are replaced with interface modules. The required model configuration is case-specific. Some properties, however, require the inclusion of several modules. When several modules are required for verification, semi-interface modules can be used to further avoid state space explosion. In what follows, we present an example of a true property of the system, and an example of an error that was found in the system design.

6.1 True Property Verification

One of the system requirements states that there should be no race conditions. For a particular device controlled by the system this means that it should not be possible to drive the device on and off at the same time by different functions. The device can be controlled by two functions that communicate with each other. These functions were modelled as two modules: $Module1$ and $Module2$. The systems operate erroneously if they produce conflicting control signals that are not marked as faulty (by their respective status bits). The resulting temporal logic formula was then:

```
G !(Module1.Device_ON & not Module1.Device_ON_Fault_status &
Module2.Device_OFF & not Module2.Device_OFF_Fault_status).
```

The property cannot be verified in the full model because of the state explosion problem. In order to verify the specification we created a model configuration in which the two related modules were included in addition to two other modules describing the behaviour of the diesel devices ($Module3$ and $Module4$). Other modules of the model were replaced with their respective interface modules. Later it was noticed that a smaller model configuration suffices to prove the property: the diesel behaviour ($Module3$ and $Module4$) is irrelevant to the proof, and these modules can also be replaced with interface modules. The verification times and state spaces in these different model configurations are presented in Table 1. The state space for the full non-abstract model could not be calculated in reasonable time. The runs were performed on a PC with Inter Core i7 Q740 processor and 3 GB of RAM. For model checking, NuSMV version 2.5.4 was used.

Table 1. Verification times and state space sizes in various model configurations

Model	Time	Mem (MB)	Reachable state space / Full state space
Full model	> 2 h	> 124	- / -
Modules 1, 2	29 s	54	6.5×10^{80} / 1.7×10^{117}
Modules 1, 2, 3, 4	690 s	68	1.5×10^{79} / 2.2×10^{119}

6.2 Found Errors

The analysis of some system properties resulted in counter-examples on the abstract models. In order to check whether these counter-examples were spurious, we ran bounded model checking on the full model using the counter-example length as the bound. Counter-examples that could be produced in the full model were interpreted as real system design errors. Some of these violations could be explained by the generality of the design documentation, i.e. the level of detail used in the design documentation did not fully include signal status handling. Other findings were related to the timing issues of the logic. In what follows, an error in which two consecutive operational sequences interfere with each other is examined in more detail.

A design error was found, in which a certain control sequence of the diesel is disrupted and restarted rapidly. This results in unwanted behaviour since the first sequence has not ended properly before the second sequence starts.

Part of the logic causing an overlapping sequence is illustrated in Figure 4. The logic consists of a set-reset flip-flop, two TON timer blocks (8 s, 30 s), a time pulse function block (10 s), AND-block and an OR-block. The logic intends to carry out a starting sequence of alternating signals given to a device. The sequence is specified so that first the ON-signal is given for 8 seconds, and then the OFF-signal is given for 10 seconds. After the OFF-signal, the ON-signal is given again for 12 seconds.

The intended sequence may be interrupted by the Reset signal but the interrupt should occur in a safe way. In particular, it is expected that the ON-signal is not given continuously for long periods of time, since this might be harmful to the device. To check for unnecessarily long control periods we implemented an observer variable in the model that counted the time that the ON-signal had been set. Then we checked whether the counter variable could reach a value of greater than or equal to 20 seconds using the temporal formula:

```
G (counter < 20)
```

When this property was examined by model checking, a counter-example was found in which the ON-signal is continuously set for 22 seconds. This behaviour occurs when the starting sequence is reset and quickly restarted just after 8 seconds after the first Start signal. This way the time pulse block is not reset, which interferes with the newly restarted system behaviour. In particular, the time pulse will not be re-initiated because the time pulse function block does not detect the rising edge from the 8 s TON-block.

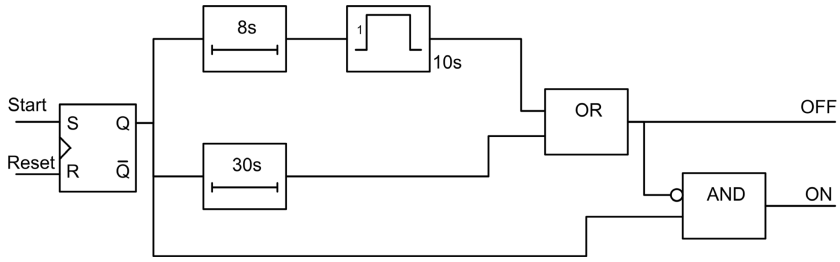


Fig. 4. Part of the logic related to overlapping sequences.

7 Conclusions

In this paper we have introduced an abstraction technique for model checking large modular function block based systems, in which abstractions of the model are created by replacing a subset of the modules with non-deterministic interface modules. We have applied the technique and the NuSMV model checking tool to the analysis of an emergency diesel control system.

The developed abstraction technique allows the model checking of large systems that otherwise can not be examined as quickly and smoothly. In the case study we were able to verify several safety properties of the system that could not be verified without the abstraction. We also found errors in the system design.

The compositional verification technique used here significantly reduces the manual work required for modelling, and reduces the required verification time of the model. The technique can be further automated to ease the analysis of large systems. The work described here is intended as a basis for an iterative abstraction refinement tool that can automatically select a suitable model configuration that is computationally feasible but at the same time describes the system to be analyzed with enough detail to enable verification of the selected property. The results of the model checking tool can be used for selecting such a configuration.

The benefit of the method is dependent on the checked temporal property. Only universal properties can be examined with the current methodology. Also, all properties may not be checked with the method because some properties are dependent on a large portion of the logic and, thus, verifying the property requires the inclusion of too many modules.

Future work includes the development of the iterative abstraction refinement tool, extending the methodology to cover liveness properties, and developing asynchronous modelling methodology. Our current NuSMV modelling technique is based on the assumption that the communication between different functions is synchronized. The assumption is, however, false since the functions are often implemented on separate decentralised processors. We have previously analysed asynchronous phenomena using timed automata (see e.g. [4]). Similar methodology would be useful also in the context of our NuSMV models.

References

1. Clarke, E.M., Grumberg, O., Peled, D.: Model checking. MIT Press (2001)
2. Lahtinen, J., Valkonen, J., Björkman, K., Frits, J., Niemelä, I., Heljanko, K.: Model checking of safety-critical software in the nuclear engineering domain. Reliability Engineering & System Safety (2012) Available online.
3. Clarke, E.M., Long, D.E., McMillan, K.L.: Compositional model checking. In: LICS. (1989) 353–362
4. Lahtinen, J., Björkman, K., Valkonen, J., Frits, J., Niemelä, I.: Analysis of an emergency diesel generator control system by compositional model checking. VTT Working Papers 156, VTT Technical Research Centre of Finland (2010)
5. Berezin, S., Campos, S.V.A., Clarke, E.M.: Compositional reasoning in model checking. In de Roever, W.P., Langmaack, H., Pnueli, A., eds.: COMPOS. Volume 1536 of Lecture Notes in Computer Science., Springer (1997) 81–102
6. Rushby, J.: Formal verification of McMillan's compositional assume-guarantee rule. Technical report, University of Minnesota, Minneapolis (2001)
7. Dams, D., Gerth, R., Leue, S., Massink, M., eds.: Theoretical and Practical Aspects of SPIN Model Checking, 5th and 6th International SPIN Workshops, Trento, Italy, July 5, 1999, Toulouse, France, September 21 and 24 1999, Proceedings. In Dams, D., Gerth, R., Leue, S., Massink, M., eds.: SPIN. Volume 1680 of Lecture Notes in Computer Science., Springer (1999)
8. McMillan, K.L.: Circular compositional reasoning about liveness. In Pierre, L., Kropf, T., eds.: CHARME. Volume 1703 of Lecture Notes in Computer Science., Springer (1999) 342–345
9. Burch, J.R., Clarke, E.M., Long, D.E.: Symbolic model checking with partitioned transistion relations. In: VLSI. (1991) 49–58
10. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching-time temporal logic. In Kozen, D., ed.: Logic of Programs. Volume 131 of Lecture Notes in Computer Science., Springer (1981) 52–71
11. Queille, J.P., Sifakis, J.: Specification and verification of concurrent systems in CESAR. In Dezani-Ciancaglini, M., Montanari, U., eds.: Symposium on Programming. Volume 137 of Lecture Notes in Computer Science., Springer (1982) 337–351
12. Cavada, R., Cimatti, A., Jochim, C.A., Keighren, G., Olivetti, E., Pistore, M., Roveri, M., Tchaltsev, A.: NuSMV 2.5 User Manual. FBK-irst. (2010)
13. FBK-IRST, Carnegie Mellon University, University of Genova and University of Trento: Nusmv model checker v.2.5.4 (2012)
14. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Trans. Computers **35**(8) (1986) 677–691
15. McMillan, K.L.: Symbolic model checking. Kluwer (1993)
16. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In Cleaveland, R., ed.: TACAS. Volume 1579 of Lecture Notes in Computer Science., Springer (1999) 193–207
17. Biere, A., Heljanko, K., Junttila, T.A., Latvala, T., Schuppan, V.: Linear encodings of bounded LTL model checking. Logical Methods in Computer Science **2**(5:5) (2006) 1–64
18. Clarke, E.M., Gupta, A., Strichman, O.: SAT-based counterexample-guided abstraction refinement. IEEE Trans. on CAD of Integrated Circuits and Systems **23**(7) (2004) 1113–1123

Publication III

Jussi Lahtinen, Tuomas Kuismin and Keijo Heljanko. Verifying large modular systems using iterative abstraction refinement. *Reliability Engineering & System Safety*, Vol. 139, p. 120 – 130, Elsevier 2015.

Reliability Engineering and System Safety 139 (2015) 120–130

Contents lists available at ScienceDirect

Reliability Engineering and System Safety

journal homepage: www.elsevier.com/locate/ress

Verifying large modular systems using iterative abstraction refinement

Jussi Lahtinen [a,*], Tuomas Kuismin [b], Keijo Heljanko [b]

[a] VTT Technical Research Centre of Finland Ltd., Systems Research, P.O. Box 1000, FI-02044 Espoo, Finland
[b] Helsinki Institute for Information Technology HIIT and Department of Computer Science, School of Science, Aalto University, P.O. Box 15400, FI-00076 Aalto, Finland

ARTICLE INFO

Article history:
Received 7 June 2013
Received in revised form
5 March 2015
Accepted 6 March 2015
Available online 14 March 2015

Keywords:
Model checking
Verification
Validation
Iterative abstraction refinement

ABSTRACT

Digital instrumentation and control (I&C) systems are increasingly used in the nuclear engineering domain. The exhaustive verification of these systems is challenging, and the usual verification methods such as testing and simulation are typically insufficient. Model checking is a formal method that is able to exhaustively analyse the behaviour of a model against a formally written specification. If the model checking tool detects a violation of the specification, it will give out a counter-example that demonstrates how the specification is violated in the system. Unfortunately, sometimes real life system designs are too big to be directly analysed by traditional model checking techniques. We have developed an iterative technique for model checking large modular systems. The technique uses abstraction based over-approximations of the model behaviour, combined with iterative refinement. The main contribution of the work is the concrete abstraction refinement technique based on the modular structure of the model, the dependency graph of the model, and a refinement sampling heuristic similar to delta debugging. The technique is geared towards proving properties, and outperforms BDD-based model checking, the k-induction technique, and the property directed reachability algorithm (PDR) in our experiments.

1. Introduction

Digital instrumentation and control (I&C) systems are increasingly used in the nuclear engineering domain. The exhaustive verification of these systems is challenging, and verification methods such as testing and simulation are typically insufficient.

Model checking [1,2] is a formal method that is able to exhaustively analyse the behaviour of a model against formally written specifications. If the model checking tool detects a violation of a specification, it will give out a counter-example that demonstrates how the specification is violated in the system.

In this work, we are primarily using the model checker NuSMV 2.5.4 [3], which was originally designed for synchronous digital hardware model checking. The NuSMV model checker does not have a notion of continuous time but instead the timing elements in our models are modelled with discrete time steps using explicit counter variables. In NuSMV, the formal correctness specification can be written as a simple state invariant clause that should hold in each individual reachable state of the system, or in a more complex specification language such as the Linear Temporal Logic (LTL) and the Computation Tree Logic (CTL) [1,2]. In addition to NuSMV, we also utilise another model checking tool called ABC/ZZ by Niklas Eén [4]. Since our models are written in the NuSMV modelling language, we translate the NuSMV models into the AIGER format [5] used in ABC/ZZ.

The specifications used in this work are formalised as state invariant specifications, but we are exploiting a procedure compatible with our approach that reduces LTL property model checking into state invariant model checking [6], thus enabling all LTL properties to be model checked.

The classical algorithm for model checking state invariant specifications is based on symbolically representing and exploring the reachable state space of the system by using Binary Decision Diagrams (BDDs) [1], which are a highly efficient data structure for representing and doing operations with large state spaces. Another way to check state invariants is to use a propositional satisfiability solving (SAT)-based approach. This line of work employs a propositional satisfiability solver in a bounded model checking (BMC) procedure [7,6], which looks for counter-examples shorter than a user provided maximum length, called the bound. An advanced variant of this procedure we employ in this work is called *k-induction* [8,9]. In that approach the state invariants are proved using induction, and the base step and the induction step of the proof are basically reduced to bounded model checking problems. Another SAT-based technique for checking safety properties is the IC3 algorithm by Bradley [10], also known as property directed reachability (PDR). The technique inductively searches for an invariant that holds in the initial state and implies the examined specification.

* Corresponding author.
E-mail address: jussi.lahtinen@vtt.fi (J. Lahtinen).

http://dx.doi.org/10.1016/j.ress.2015.03.012
0951-8320/

The traditional model checking algorithms including the ones described above have been successfully used to analyse individual nuclear domain safety I&C systems, see e.g. [11,12], as well as satellite onboard software designs [13]. However, in the nuclear domain it is common to cope with hardware failures by implementing several subsystems that execute the same physical function using design diversity in software and/or hardware. It may be necessary to examine these diverse subsystems simultaneously, e.g. because the specifications may in fact cover their combined behaviour, and to also additionally check that the diverse subsystems have no unintended interactions. Unfortunately, the currently available classical model checking methods by themselves do not always scale to analysing these large and complex combined systems. In our experience, the Binary Decision Diagram (BDD)-based techniques by themselves have proven to be inadequate in the analysis of our models in some of these larger system configurations. The alternative SAT-based bounded model checking (BMC) techniques [7,6] can often find counter-examples in many large systems. Some BMC techniques such as the k-induction technique [8,9], and the PDR algorithm [10] can prove properties, but in our experience the necessary CPU time and memory needed for a proof can make the verification impractical for many real life designs.

One classical approach to avoid the scaling problem is to use abstraction. Intuitively, *abstraction* is the act of simplifying a model with the intention of making the verification of the model more efficient, whereas adding more detail to the model is called a *refinement*. In systems where multiple diverse subsystems are present, the whole system functionality is rarely needed to verify a system property. Depending on the exact specification some subsystems or parts of subsystems may be irrelevant for proving the specification, and can be abstracted. The abstractions we use in this work are over-approximations. Over-approximation techniques tend to relax constraints, e.g., by allowing a variable to also have values that are not realistic. Over-approximation leads to a model that has more behaviour but can be less complex to analyse. Because of the possible unrealistic behaviour in the model, over-approximation can lead to spurious counter-examples. [14] Since the over-approximated model has more behaviour than the concrete model, the correctness of the resulting abstract model implies also the correctness of the full model when universal properties such as state invariants are examined.

Unfortunately, creating such an abstract model for each checked specification is non-trivial and requires a lot of manual work, becoming tedious and thus also error prone. For the best efficiency gains, the abstraction is tailored for each specification separately.

In this paper, we describe how these kinds of over-approximating abstractions can be created *fully automatically* by using an iterative abstraction refinement technique exploiting the modular structure of the system. Our technique will (i) significantly reduce the amount of manual work needed to create these abstractions, (ii) prove the system correctness based on verification runs automatically performed on these abstractions, and (iii) reduce the overall computational effort required for model checking, enabling the model checking of larger system models. Our approach is designed to be efficient at finding proofs for properties, as we expect most of the designs at this stage of inspection to actually be correct. For all properties that hold, the algorithm will find some abstraction of the system.

In our technique, we require that the system is structured into modules. Abstractions of the model are created by automatically replacing a subset of the modules with stubs that can at each time point non-deterministically give any value from any of their outputs. These simplified modules are called *interface modules*. This approach is similar to the compositional minimisation [15] technique.

In iterative abstraction refinement an initial abstraction is first generated and model checked. If the examined property produces a counter-example, the model is refined and the resulting new model is verified again. The process is continued until the property is proved or no further model refinement is possible.

In the model checking step of our abstraction refinement technique three model checking algorithms (BDD-based, k-induction, PDR) are run in parallel in a portfolio-based manner, similar to what is described in [16].

The abstractions in our technique are refined using a two-phase procedure. First, in the preliminary refinement phase, we obtain a computationally manageable subset of the modules in which the previously found counter-example becomes infeasible. This part of the refinement procedure is based on traversing the dependency graph of the modules. After the preliminary refinement phase we attempt to minimise the size of the needed model refinement using an iterative sampling procedure similar to delta debugging. Delta debugging [17,18] is originally a technique for isolating failure causes of software errors automatically. The technique works by systematically narrowing down failure-inducing circumstances until a minimal set remains. In our work we use the same principles in order to minimise the size of the refinement. The feasibility of the candidate refinements is repeatedly checked during both model refinement phases. These feasibility checks are performed using k-induction [8,9] to see whether the spurious counter-example of length k has been removed.

We have tested our technique, and report experimental results from verifying two different systems with it. The first system is a fictional case study that consists of two diverse safety systems. The fictional system is used to demonstrate the technique in practice, and to show more detailed examples of the system implementation. The second system is an actual industrial emergency diesel generator control system that is used for evaluating the performance of our technique on a real life system. The system is safety-critical, as emergency diesels are used e.g. in nuclear power plants to provide electricity in case of power failures. In both case studies we have compared our technique against three other model checking approaches: classical BDD-based model checking, SAT-based k-induction, and PDR-based model checking. In the comparisons these approaches used the full concrete model for verification. The results show that for most of the properties our technique is able to find a proof of correctness of the system more efficiently than the other three approaches.

2. Verified systems

We have tested our algorithm by applying it on two case study systems. The first is a fictional safety system we have constructed for demonstration purposes. The other is a model of an actual emergency diesel generator. The purpose of the emergency diesel model is simply to provide some additional benchmark information on the performance of the algorithm on a real-life industrial model.

2.1. Fictional system description

We have created a fictional system model for demonstration purposes. The NuSMV model of the system is available online [19]. The fictional model consists of two safety systems. Safety system 1 reads temperature measurements and controls an actuator device (a pump) if the measurements exceed a certain limit. Safety system 2 has pressure measurements as input, and it controls two other actuator devices. The systems are redundant, and the purpose of the pumps is to cool down a process so that the temperature and pressure of the underlying process remain sufficiently low.

The basic functionality in both of the safety systems is such that if the measurements exceed the given limits, the safety system is initiated and this fact is memorised. The safety systems are also associated with particular timing sequences that are given to the actuators when they are started.

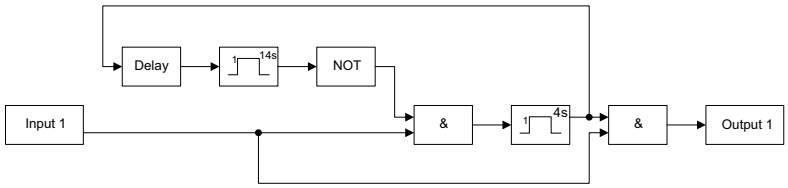


Fig. 1. The function block diagram associated with module 11 of the fictional system.

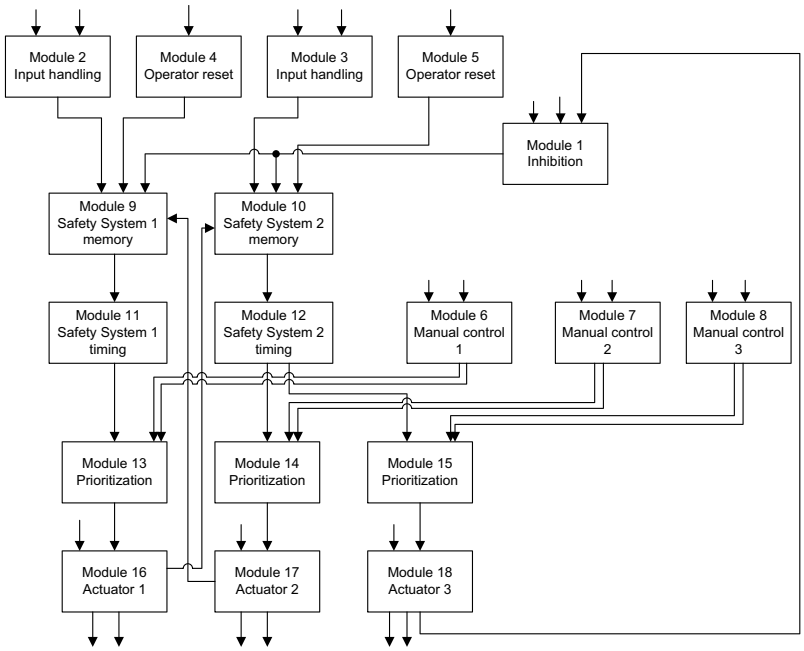


Fig. 2. An overview of the fictional model consisting of two different safety systems. The figure represents the dependency graph of the modules in the system.

The system includes the manual control of the actuators as well. The safety systems can cease to operate if the relevant input is no longer above the limit and the operator presses a manual reset button. In addition, the operator can stop and start the actuators manually using push-buttons. Since this can lead to conflicting commands to the devices, prioritisation logic is associated with each actuator. Manual stop has the highest priority, manual start has the second highest priority, and the start commands from the safety systems have the lowest priority.

We have divided the system into 18 modules so that each module encompasses a single function of the system. Each of the modules is defined by a function block diagram. As an example, the function block diagram realising the behaviour of module 11 is illustrated in Fig. 1. The functionality of the other modules is defined in a similar manner. The dependency graph of the fictional model is illustrated in Fig. 2. The dependency graph tells how information flows between modules. For example, module 11 represents a function block diagram that receives its input from the output of module 9. The output of module 11 is used by the module 13 as input. Safety system 1 is implemented in modules 2, 4, 6, 9, 11, 13 and 16. Safety system 2 is implemented in modules 3, 5, 7, 8, 10, 12, 14, 15, 17 and 18.

Both safety systems are associated with a certain timing sequence (realised in modules 11 and 12). For example, the timing sequence specified for safety system 1 in module 11 is to drive the actuator for 4 s, and then wait for 10 s. If the input of module 11 is still set after this the sequence is driven again. Module 12 drives the actuators (modules 17 and 18) for 30 s once when the command to initiate the safety system 2 is received.

In our model, the actuators (modules 16, 17 and 18) can fail non-deterministically, and cannot be repaired after a failure. An actuator starts functioning after it has received a start command continuously for a specified amount of time. The actuator also outputs a feedback signal that implies whether a start command is being received.

The model has some internal feedback loops as well. The simultaneous start of actuators 1 and 2 is forbidden. This mimics a real life situation where simultaneous use would result in excessive power usage. For this reason, the rising edge of the start signal of actuator 1 prevents the simultaneous start of safety system 2, and vice versa. Both safety systems can also be inhibited if some related equipment has failed. Module 1 reads measurements indicating equipment failures, including the failure of actuator 3. For example, if the actuator 3 is broken, both safety systems are prevented from giving the start command. Manual controls are not prevented.

The main requirements of the fictional system are described in Table 1.

2.2. Real world diesel generator control system

For evaluating real world performance we use a model of an actual emergency diesel generator control system. The emergency diesel

Table 1
The main requirements of the fictional system.

Requirement	Description
Requirement 1	If the temperature measurements surpass a limit value, safety system 1 is started
Requirement 2	If the pressure measurements surpass a limit value, safety system 2 is started
Requirement 3	Manual OFF command in module 6/7/8 stops the actuator 1/2/3
Requirement 4	Manual ON command in module 6/7/8 starts the actuator 1/2/3, unless the manual OFF command is received
Requirement 5	Whenever the inhibit signal of module 1 is true, the safety systems are not given a new start command in modules 9 and 10
Requirement 6	If the temperature measurement does not exceed the limit value, and operator reset is received from module 4, the memory in module 9 is reset
Requirement 7	If the pressure measurement does not exceed the limit value, and operator reset is received from module 5, the memory in module 10 is reset
Requirement 8	The priority of control commands in modules 13, 14 and 15 is Manual STOP, Manual START, Automation Start
Requirement 9	The start of the safety systems shall not occur simultaneously due to excessive power usage

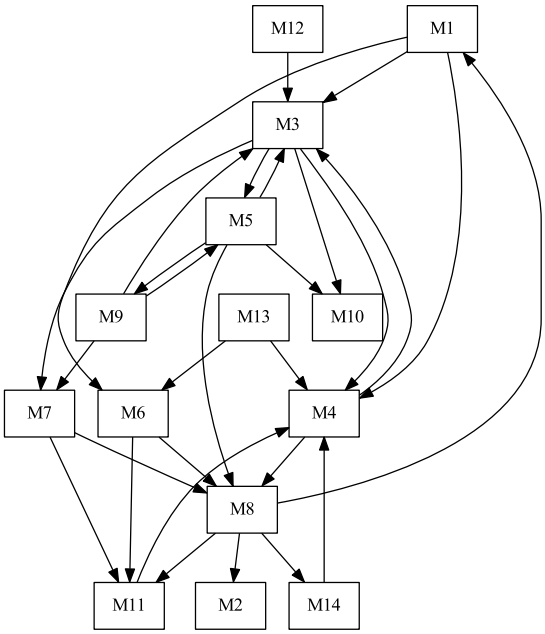


Fig. 3. The dependency graph of the emergency diesel generator control system model.

system provides reserve power in case of power loss. Emergency diesel systems are used in e.g. nuclear power plants where it is essential that safety systems are constantly connected to a power supply. In case of a black out or a disturbance in the main power supply, the diesel generators can be quickly turned on to keep the necessary systems operational.

There is a large amount of logic related to the diesel generator control. The analysed system covers 10 different control functions, including functions related to the activation, operation, and protection of the diesel generators, and voltage and frequency regulation. In addition, some functions are diverse implementations of another function that intend to increase the system's tolerance to failures. Diverse functions perform the same physical function using a different software or hardware design.

The control logic of the system was represented as function block diagrams. We modelled the control logic in the NuSMV modelling language. The model was created manually based on a functional description of the vendor-specific function blocks used in the design, and several function block diagrams describing the system functions. Each of the system's functions, as well as the system environment, was written as a separate module, resulting in 14 modules. The dependency graph of the model is presented in Fig. 3. Details of the emergency diesel system are not presented here due to non-disclosure agreements. A more detailed description of the case study together with a manual compositional verification approach for it can be found in a technical report [20].

3. Abstractions of large modular systems

3.1. Modelling methodology

We leave out the hardware and system level aspects, and focus on modelling the design logic of the system. In our example cases the design logic is implemented using function blocks. A function block diagram consists of a set of function blocks and the connections between the function blocks. Each function block diagram has a set of inputs, some internal state, and calculates its outputs based on these two. Inputs of a function block diagram can be connected to outputs of other function block diagrams, or to the environment of the system. For example, module 11 (presented in Fig. 1) has one input, one output, two pulse function blocks, an AND gate, a NOT gate, and one delay function block.

A function block based system is modelled so that a separate function block library is first created that has all the declarations for the function block types used in the system. A model of a function block diagram then creates instances of these function blocks and makes the connections between the inputs and outputs of the function blocks. We allow the environment of the system to behave freely and independently of the system under verification. Further details on our modelling methodology can be found in [12,21].

3.2. Partition into modules

Our technique works with systems that can be easily partitioned into modules. We also require that the modules are on a single level of hierarchy, and that a module does not contain another module.

In the modelling phase there are several possible ways to split the system into modules. In our example cases the modules correspond to the function block diagrams of the system. The level of coarseness of the models comes from the design of the industrial systems. In the diesel generator control system the partition of the system into function block diagrams was dictated by the actual design process. In the fictional system, we tried to create a modular partition with a similar level of coarseness so that it would resemble real life industrial designs. The modular structure is also visible in the NuSMV model of the system. Each module used in the technique is implemented using one or more NuSMV modules. In our implementation of the technique we use annotations in the model to indicate which NuSMV modules correspond to the modules used for abstraction in our technique. It would be interesting to see whether changing the level of coarseness of the modules can have influence on the performance of our technique.

However, in this paper, we do not discuss how the partition into modules should be performed.

3.3. Abstracting the model through compositional minimisation

In our abstractions we exclude some parts of the model from examination by doing compositional minimisation. In the compositional minimisation technique [15] the system is abstracted using reduced versions of some of the system's modules. We call such a reduced module version an *interface module*. It has the same outputs as the concrete module it substitutes, but no internal state. No restrictions are set to the outputs: they are completely non-deterministic, and this is the abstraction we use for all modules. The compositional minimisation technique we use is discussed in more detail in our previous work [22,23].

An abstraction of the system is created by selecting either the concrete version or the interface version of each module. Using interface modules instead of concrete ones can simplify the state space of the model significantly.

The abstractions discussed above are such that the abstract model simulates the concrete model, i.e. the interface modules are over-approximations of the modules. Because of this universal properties (properties of LTL and ACTL$*$) that are true in the abstraction are also true in the concrete model. For example, suppose we wanted to verify a universal property that refers to variables from module 9 and module 14 in Fig. 2. Through trial and error, we select a set of modules whose functionality is significant with respect to the property, say modules 9, 17 and 14, and replace all other modules with an interface module. If the abstract model created based on this selection can be used to prove that the universal specification is true, then the whole system must also function according to the specification. In what follows, for simplicity we only focus on the verification of state invariant properties. State invariant properties express that a condition holds for all reachable states. State invariants are useful for checking safety properties. We can check, for example, that two signal values controlling a particular device in opposite directions can never occur at the same time. Universal properties are more general, as they may also give requirements for path fragments.

4. Modular iterative abstraction refinement

Our algorithm finds a suitable level of abstraction by selecting modules that are necessary to prove a property. By suitable, we mean that the abstraction is detailed enough to verify the analysed property, but coarse enough so that it can be model checked in reasonable time. We achieve this through iterative abstraction refinement. Our technique is able to produce counter-examples as well, but it specialises in finding proofs quickly. In order to get faster verification results for both true and false specifications, it is useful to run our algorithm simultaneously with another one that is good at finding counter-examples quickly. For example the traditional BMC engine is such an algorithm [24]. Our technique uses three existing model checking algorithms as subroutines: an incremental BMC-based k-induction algorithm, a BDD-based algorithm, and the property directed reachability (PDR) algorithm.

4.1. Implementation

Our general iterative abstraction refinement loop is illustrated in Fig. 4. We examine an abstract version of the model, possibly get a counter-example, and refine the abstraction by adding new modules to the current abstract configuration so that the spurious counter-example is not present. We then minimise the refinement using a minimisation heuristic. The general algorithm is as follows:

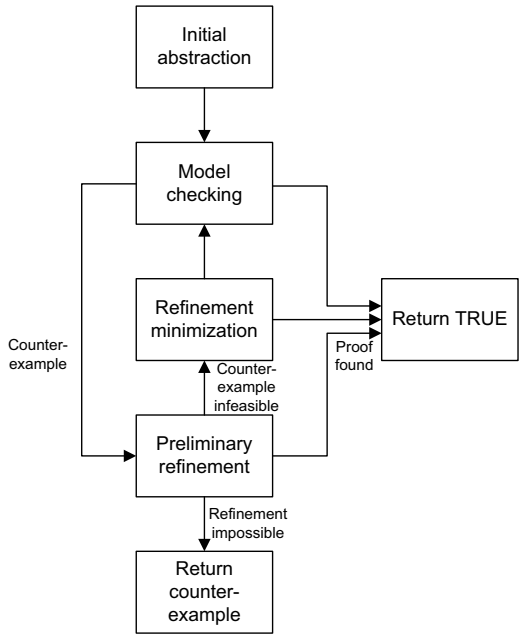


Fig. 4. The general algorithm for verifying large modular systems.

1. Choose the initial configuration of modules based on the state variables appearing in the state invariant property.
2. Model check the current abstraction of modules by running the k-induction, the BDD-based method and the PDR algorithm in parallel and terminate when one of them finishes. If a proof is found return “true”. Otherwise a counter-example is produced, and refinement is attempted based on it.
3. Based on the dependency graph of the system modules, create a preliminary refinement of the model by iteratively adding new modules until the counter-example becomes infeasible. If the model cannot be refined any further (all the modules on which the property is dependent on have been included), return “false”, and report the abstract counter-example from the feasibility check.
4. Minimise the refinement by sampling subsets of the preliminary refinement and checking the feasibility of the samples. After successful minimisation go to step 2.

The feasibility of the candidate refinements is checked in steps 3 and 4 of the algorithm. The feasibility check is performed by checking the original specification using k-induction, and the counter-example length as the bound k. The feasibility check can have three different outcomes:

1. The k-induction algorithm manages to prove the property during the feasibility check.
2. k-induction cannot find a counter-example within the bound.
3. A counter-example is found.

If the property is proved during the feasibility check, we return “true”. If a counter-example cannot be found within the bound, the refinement has been successful. If a counter-example is still found, the refinement is not concrete enough.

Note that because our approach is geared towards obtaining proofs we assume that the initial counter-example is spurious, and do not check its feasibility before refinement. The general intention of the algorithm is to begin with as much abstraction as possible, and

then iteratively add modules to the configuration until the abstraction satisfies the property. If the property is false, the abstraction will contain all the modules in the cone-of-influence of the property, and the resulting counter-example is returned as a result.

For all properties that hold, the algorithm will find some abstraction of the system. Note that this abstraction might be the original non-abstract system in the worst case. Furthermore, the abstractions computed by the algorithm are subset minimal, that is, no single module added to the new refinement can be made abstract without making the property feasible. This is the case because the refinement minimisation algorithm will try to remove each individual module separately from the refinement before the final minimised refinement is returned. In what follows, the individual steps of the algorithm are described in more detail.

4.1.1. Initial configuration

The initial configuration is selected by extracting the variables from the examined specification and determining the modules whose outputs these variables are. These modules represent the initial configuration, and concrete versions of these modules are used in the model. Other modules of the model are replaced by their respective interface modules.

4.1.2. Model checking

The second step is model checking the current configuration of modules. In this step we use several model checking algorithms in parallel as illustrated in Fig. 5.

In our implementation, BDD-based state invariant checking, and the k-induction algorithm are run using NuSMV, while the property directed reachability (PDR) algorithm is run using the ABC/ZZ model checker [4].

We have used the incremental k-induction algorithm, since the algorithm is also capable of proving properties using an induction based approach. A standard BMC engine could have also been used instead of k-induction, but the proofs of k-induction improve the performance of our approach. The BDD-based model checking algorithm may require a lot of memory and time when the size of the model increases, which is why the other model checking algorithms have been included. The BMC and PDR algorithms can find counter-examples faster, and thus reduce the overall run-time of the algorithm. The approaches are often complementary to each other.

The ABC/ZZ tool uses AIGER [5] format models as input. In order to be able to use the PDR algorithm implemented in the ABC/ZZ tool some model transformations are necessary. We first flatten the NuSMV model using the flattening feature implemented in NuSMV and then run the `smvtoaig` tool that is in the AIGER tool package that creates a corresponding model in the AIGER format. We also transform the state invariant into an LTL property so that the model together with the LTL property translates correctly into an AIGER format model that can be model checked by the PDR model checker. The LTL formula is created simply by adding the LTL globally operator G before the state invariant.

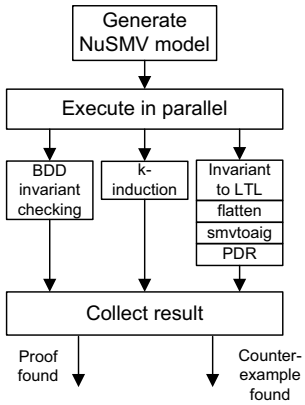


Fig. 5. Parallel model checking used in the algorithm.

4.1.3. Preliminary refinement

In case of a spurious counter-example, the objective is to find a new abstraction (i.e. a configuration of concrete modules and interface modules) that is more detailed than the current configuration of the model and makes the current counter-example infeasible. The original specification is then checked again on the refined abstraction.

The preliminary refinement phase is presented as pseudo-code in Algorithm 1. The function *RefineConfiguration* has as input the set *Current* of modules that are concrete in the current abstraction, and the length of the recently received counter-example *CElength*. The algorithm returns the set of new modules *Refinement* that are added to the current model, and a string indicating whether no further refinement is possible or if a proof is found while checking the feasibility of the refinement.

Algorithm 1. Preliminary refinement.

```
1:  procedure RefineConfiguration (Current,CElength)
2:    Configuration ← Current
3:    Refinement ← ∅
4:    while True do
5:      newRefinement ← ∅
6:      for e ∈ getNeighbourModules(Configuration) do
7:        if e∉Configuration then
8:          newRefinement ← newRefinement ∪ e
9:        end if
10:     end for
11:     if len(newRefinement) = = 0 then
12:       return [Refinement, "no refinement"]
13:     end if
14:     Configuration ← Configuration ∪ newRefinement
15:     Refinement ← Refinement ∪ newRefinement
16:     CEinfeasible, proved ← checkFeasibility
        (Configuration, CElength)
17:     if CEinfeasible then
18:       if proved then return [Refinement,"proof"]
19:       end if
20:       break    ▷ Refinement found
21:     end if
22:   end while
23:   return [Refinement, ""]
24: end procedure
```

The algorithm uses the dependency graph of the model to identify new modules to be added to the model.

The function *getNeighbourModules* traverses the dependency graph one step in the backwards direction starting from all the nodes representing the modules in the current abstraction, and returns the modules representing these neighbour nodes.

A new model configuration *Configuration* is generated in which the modules in the refinement are concrete, and all other modules are interface modules. If all of the relevant edges of the dependency graph have been traversed, and no new modules can be added, the property is false in the concrete model. The counter-example that was generated in a previous step is the final counter-example. Note that

these counter-examples may have some abstract modules that are not capable of influencing the variables in the property.

When refinement candidates are being created, the feasibility of the previous counter-example is checked using the function *checkFeasibility*. The function checks the original specification using k-induction with the bound k set to the length of the counter-example. If a counter-example can be found within the bound, the refinement has not been successful, and we continue with the preliminary refinement phase. If no counter-examples can be found within the bound, the refinement has been successful, and we move on to refinement minimisation.

We do not use counter-example information, other than its length, to guide the refinement process. We use the counter-example length during the feasibility checks. We simply check that counter-examples of similar length (or shorter) are not possible. One specific advantage is that all counter-examples of length up to the bound k are eliminated in one check. Another benefit is that k-induction may prove the property during the feasibility checks. According to our experience further analysis of the counter-example is too demanding when compared to the achieved benefits. This simple approach leads to better performance in our algorithm.

Algorithm 2. Refinement minimisation.

```
1:  procedure MINIMISATION (Current,CElength,Refinement)
2:    n ← 2
3:    NewRefinement ← Refinement
4:    while True do
5:      if len(NewRefinement) < 2 then
6:        return [Current ∪ NewRefinement, ""]
7:      end if
8:      subsets ← partitionSet(NewRefinement, n)
9:      complements ← getComplements(subsets, NewRefinement)
10:     CEinfeasible ← False
11:     for c ∈ subsets ∪ complements do
12:       Configuration ← Current ∪ c
13:       CEinfeasible,
      proved ← checkFeasibility(Configuration, CElength)
14:       if CEinfeasible then
15:         if proved then return [Configuration, "proof"]
16:         end if
17:         if c ∈ subsets then
18:           n ← 2      ▷ c ∈ subsets
19:         else
20:           n ← max(n − 1, 2)  ▷ c ∈ complements
21:         end if
22:         NewRefinement ← c
23:         break
24:       end if
25:     end for
26:     if CEinfeasible then continue
27:     end if
28:     if n < len(NewRefinement) then
29:       n ← min(len(NewRefinement), 2n)
30:       continue
31:     else
32:       return [Current ∪ NewRefinement, ""]
33:     end if
34:   end while
35: end procedure
```

4.1.4. Refinement minimisation

After the first suitable refinement has been found, we perform refinement minimisation. The minimisation algorithm is represented as pseudo-code in Algorithm 2.

The function *Minimisation* has as input the set *Current* of modules that are concrete in the current abstraction, the length of the recently received counter-example *CElength*, and the set of modules *Refinement* in the preliminary refinement. The algorithm returns a new set of modules *Configuration* that includes all modules in *Current* and a locally minimal set of modules in *Refinement* that suffice to make the refinement feasible, and a string indicating if the property could be proved during the feasibility check.

In refinement minimisation we iteratively sample subsets of the modules in the preliminary refinement and check counter-example feasibility on the resulting model. The feasibility checks are performed similarly as in preliminary refinement using the function *checkFeasibility*. If a suitable subset of modules is found, the minimisation procedure is restarted using the found subset as a starting point. The approach leads to a locally minimal subset of modules. Note that our refinement approach is cumulative, since the minimisation is applied only to the new modules in the preliminary refinement. The modules from previous iterations cannot be removed in the minimisation.

The subset sampling is based on the delta debugging technique described in [17,18]. The purpose of the original technique is to generate a simple test case that captures the variable assignments that cause a particular failure. In delta debugging based refinement minimisation, we first partition the set of modules into two parts, giving us the initial granularity of two, and check refinements based on both sets. If the minimisation is not successful, we increase the granularity, and divide the set of modules into four parts. After this we check these four sets and their complement sets. If none of these subsets is suitable, we again increase granularity. The process is continued until the granularity reaches the size of the module set.

The function $partitionSet(set, n)$ partitions a set into n parts. For example, $partitionSet([1,2,3,4,5,6],3)$ returns $[[1,2],[3,4],[5,6]]$. The function $getComplements(partitions, set)$ is used to produce the complements of these sets. $getComplements([[1,2],[3,4],[5,6]],[1,2,3,4,5,6])$ returns the set $[[3,4,5,6],[1,2,5,6],[1,2,3,4]]$.

To simplify the presentation, the pseudo-code in Algorithm 2 misses some of the functionality in our implementation. In our implementation, we also maintain a history of checked refinements in order to avoid redundant feasibility checks. We also randomise the order of the modules in *NewRefinement* in the beginning of the while loop so that the order does not have influence on the result of the minimisation.

5. Related work

Our verification approach is based on iterative abstraction refinement, which has been used in various model checking related contexts. The generic verification strategy is based on the classical four steps: (i) generating an initial abstraction, (ii) model checking the property on the abstraction, (iii) checking possible counter-examples on the concrete model, and (iv) refining the abstraction when needed. Iterative abstraction refinement was first introduced as the localisation reduction by Kurshan et al. [25]. This technique is similar to ours in the way abstractions are made. The abstractions are non-deterministic but they are performed on the variable level instead of using abstraction on the module level as we do. The technique relies heavily on the dependency graph of the variables.

After localisation reduction, many variations to the generic iterative refinement loop have been suggested [26–31]. The variations differ in the way the abstraction is used and refined, and in the way the spurious counter-examples are handled. The CEGAR technique by Clarke et al. [27] uses a more general existential abstraction technique based on predicate abstraction that divides the variables into abstract variable clusters. In this approach spurious counter-examples are checked by symbolic BDD-based simulation, and the refinement step

uses the counter-example to partition some previous equivalence class of the abstraction so that the counter-example becomes invalid. Later research, e.g. [14] has suggested that the use of SAT-based methods can be more effective in abstract counter-example validation.

Our technique has been inspired by the original CEGAR techniques but instead of using predicate abstraction and computing abstract transition relations using predicates, we use a lighter-weight approach where each submodule of the system can be replaced by its abstract counterpart, which seems to be a more scalable technique for large scale industrial safety systems.

McMillan et al. [30,32] have introduced the "Proof-based Abstraction" approach where the counter-example itself is not used in the feasibility checking or refinement steps. Instead, they use a SAT-solver to prove that counter-examples up to a certain bound k are not possible. The idea is to use BMC techniques for refinement and BDD-based techniques to prove the refined model. This same basic approach is used in our technique. However, the way we refine the model is different as we are exploiting the modular structure of the system to be verified. In our refinement step we find a set of modules that become concrete and thereby remove all counter-examples up to the length of the current one. In addition, in our technique a side result of the feasibility check may be that the k-induction finds a proof for the property. Another difference is that in the proof-based abstraction technique the refinement is not cumulative, while our technique creates cumulative refinements.

There are other iterative refinement techniques that focus on modules and compositionality. The work in [33] is in the context of explicit model checking of state graphs. They use a component refinement method based on parallel composition.

Finally, our algorithm runs several verification engines in parallel. The use of multiple verification engines is not new, see e.g. [34,31]. For example in [31], two different model checkers are used in the abstraction refinement loop. They use a SAT-based technique to check and concretise spurious counter-examples, and a BDD-based model checker to verify the abstractions. The refinement technique is based on information from multiple counter-examples. The use of several model checking engines in a parallel portfolio has also been studied, e.g. in [24], but we are not aware of other approaches than ours related to parallel use of several different model checking algorithms for iterative abstraction refinement.

6. Results

A prototype implementation of the technique was created in the Python programming language. The Python implementation is available online [19]. Some example properties were verified on the fictional model. In addition, we tested 100 randomly generated true invariants on a model of a real world emergency diesel generator control system. The verification runs were performed on a computer with an Intel Xeon X5560 processor with 16 cores running at 2.80 GHz. In the model checking phase of our technique, three cores are used concurrently. A memory limit of 4 GB and a timeout of 1800 s were used in the tests.

6.1. Verification on the fictional model

A set of 20 state invariants was formalised based on the list of requirements for the fictional model (Table 1). Most of the invariants proved to be quite easy for the traditional invariant checking algorithms. However, some state invariants were very difficult for these algorithms. As an example we present five formalised properties with an emphasis on the more difficult properties:

- Property 1 (based on requirement 1 in Table 1): When the measurements in module 2 (temperature) are over the designated limits, and the prevention signal from module 1 is not set, and the feedback signal from module 17 does not indicate that the diverse safety system is starting, safety system 1 is initiated by module 9. As a state invariant, this can be written as

  (((temp1 > 250) & (temp2 > 250) &
  ! MOD1.output1 & ! MOD9.feedstop) − > MOD9.output1)

- Property 2 (based on requirement 3 in Table 1): If a manual OFF command has been given at the previous time point (module 6), the device (module 16) is not on. This requires that module 6 keeps track of the history of the manual OFF command in its internal state. As a state invariant,

  ((MOD6.prevoff) − > ! MOD16.output1)

- Property 3 (based on requirement 9 in Table 1): Unless manual commands are used, actuators 1 and 2 (modules 16 and 17) cannot be started exactly at the same time. As a state invariant,

  ! (! MOD6.wasused & ! MOD7.wasused &
  MOD16.risingedge & MOD17.risingedge)

- Property 4 (based on requirement 9 in Table 1): If actuator 1 (module 16) receives a starting command but has not yet started, module 10 cannot initiate the diverse safety system. As a state invariant,

  ! (! MOD16.output3 & ! MOD16.output1 & MOD13.output1 &
  MOD16.output2 & MOD10.output1 & ! MOD10.wasprev)

- Property 5 (based on requirement 9 in Table 1): If actuator 2 (module 17) receives a starting command but has not yet started, module 9 cannot initiate the diverse safety system. As a state invariant,

  ! (! MOD17.output3 & ! MOD17.output1 & MOD14.output1 &
  MOD17.output2 & MOD9.output1 & ! MOD9.wasprev)

The above properties were verified using four techniques: BDD based invariant checking, the BMC based k-induction technique, property directed reachability (PDR), and our algorithm. The bound used in the k-induction technique was 10 000. We used such a large bound because we prefer running out of memory to not reporting a result. The verification results are in Table 2.

The verification times in Table 2 for PDR do not include the time to transform the NuSMV model to the AIGER format. The results show that our algorithm manages to verify all properties rather

Table 2
Verification times for model checking the state invariants using four techniques: BDD based invariant checking, the BMC based k-induction technique, property directed reachability (PDR), and our algorithm.

Property	BDD	k-induction	PDR	Algorithm
Property 1	3.25 s	1.6 s	7.27 s	0.49 s
Property 2	TIMEOUT	1.71 s	5.48 s	0.47 s
Property 3	TIMEOUT	1.99 s	7.83 s	8.11 s
Property 4	TIMEOUT	MEMOUT	16.95 s	0.21 s
Property 5	TIMEOUT	MEMOUT	52.80 s	0.41 s

quickly, while the BDD-based technique, and the k-induction technique cannot verify all properties within the given resource limits. Our technique is also faster than PDR, though the PDR algorithm could also verify all of the properties. Our technique is the fastest in properties 1, 2, 4 and 5. The k-induction technique is the fastest for verifying property 3.

6.2. Verification on the emergency diesel control system

The technique was evaluated by using the real world diesel control system as a benchmark. The performance of our technique was compared with other model checking algorithms. Since our technique focuses on finding proofs efficiently, we have used only true state invariants in the comparison. We verified 100 randomly generated true state invariant specifications on the diesel model using four approaches: our technique, k-induction, property directed reachability (PDR), and a BDD-based state invariant checking method. In the BDD-based method and the k-induction method the cone-of-influence reduction (NuSMV option -coi) was also used. The BDD based state invariant checking algorithm was also run with dynamic variable ordering (NuSMV option -dynamic). Both of these options resulted in better running times. The PDR algorithm was run using the command `bip ,live -k=inc -eng=pdr2`. The BDD approach, the k-induction approach, and the PDR approach used the full concrete model.

The random state invariants were generated so that two or three variables were randomly selected from the set of the modules' outputs and non-deterministic variables of the main module. Negations were randomly placed in front of the chosen variables, and the operators joining the chosen variables were randomly selected (AND/OR). After this the random state invariant was model checked using the different model checking methods until each state invariant was proved either true or false. Finally, only the true state invariants were picked out for the results. The following is an example of a random state invariant property:

```
((not MOD14.output1 OR MOD2.output2) AND
  not main_module_input1)
```

The results of the comparison are in Figs. 6–8.

The k-induction algorithm was able to verify 55% of the properties rather quickly, but for the rest of the properties (45%) the required bound for a proof was so large that the technique ran out of memory. The BDD-based method was able to verify 81% of the properties within the 1800 s timeout. 13% of the generated random state invariants were such that neither the BDD-based technique nor the k-induction technique could verify them within the given resource limits. The PDR technique was able to solve all properties within the given resources. Fig. 8 does not show the time to generate the AIGER format model from the NuSMV model. On average this time was 17 s. The model transformation time is, however, included in the PDR verification runs performed as part of our technique since these model transformations depend on the model configuration and cannot be calculated beforehand.

Fig. 7 contains a distinct vertical grouping of data points. The set of points indicates that many properties have been solved in approximately 15 s by the k-induction method. We assume that this is because a certain amount of time is needed for building and initialising the model, after which the properties may be quickly

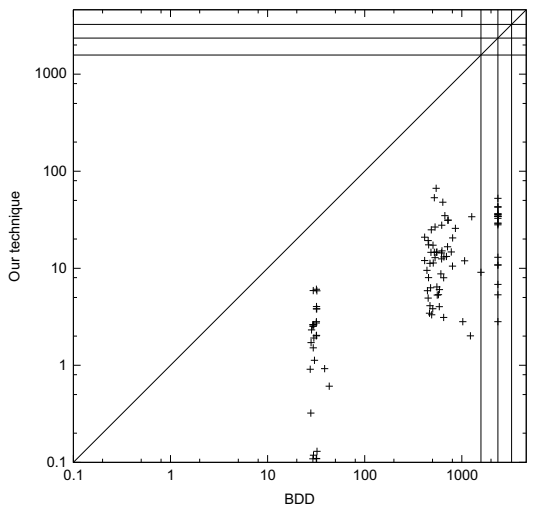


Fig. 6. Run-times (in seconds) of BDD model checking and our technique.

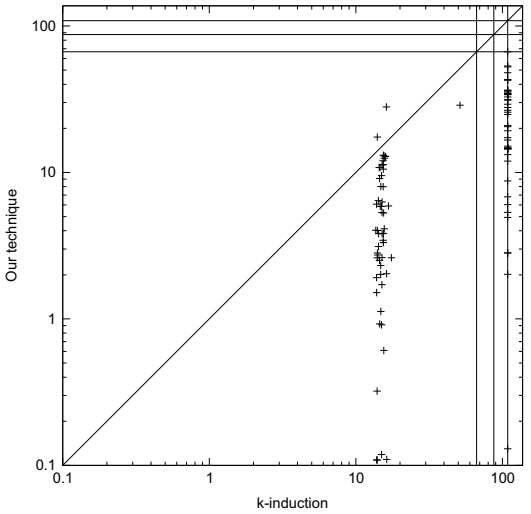


Fig. 7. Run-times (in seconds) of k-induction and our technique.

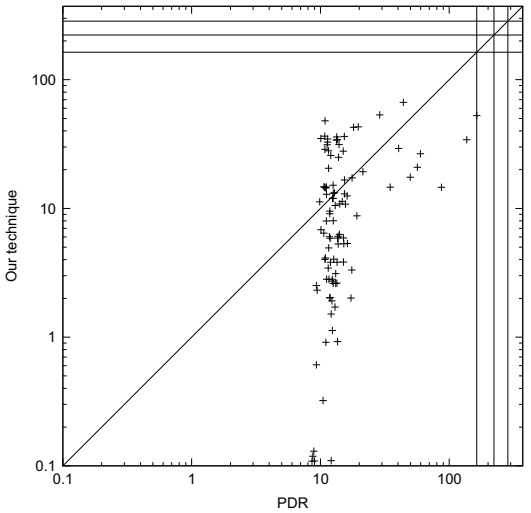


Fig. 8. Run-times (in seconds) of PDR and our technique.

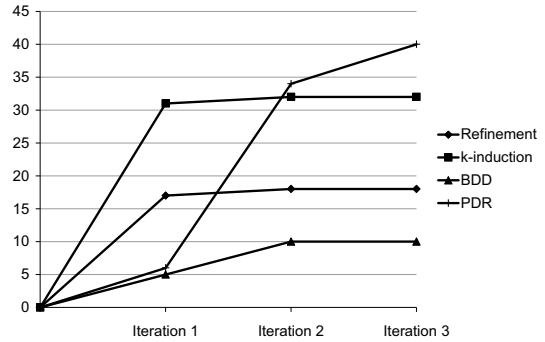


Fig. 9. The number of proofs found by k-induction, BDD, PDR, and refinement phase k-induction, and the cumulative number of proofs found on different iterations of the algorithm.

proved within a small bound. A similar phenomenon is apparent in Figs. 6 and 8 as well.

Our technique could verify all of the properties within the given resources. In cases where the other techniques were able to provide a proof, our technique was typically faster. In 68% of the cases the properties were such that the proof could be found faster using our technique than by using the other three algorithms. In 1% of the cases, the k-induction technique was the fastest. The PDR-based technique was the fastest in 31% of the cases.

When our technique is analysed in detail, we note that 82 properties are proved in the model checking phase of the algorithm, and 18 properties are proved in the preliminary refinement phase of the algorithm. The distribution of the model checking phase proofs was such that 32 properties are proved by the k-induction subroutine, 10 properties are proved by the BDD-based subroutine, and 40 of the properties are proved by the PDR-based subroutine. The proofs of the preliminary refinement phase are also generated by the k-induction subroutine as only that subroutine is used for checking the feasibility of the refinement. The cumulative frequency graph in Fig. 9 shows the number of proofs obtained by the different subroutines (k-induction, BDD, PDR, and refinement phase k-induction) and the distribution of these proofs on the different iterations of the algorithm. Three iterations of the algorithm sufficed to prove all properties.

The technique seems to work so that the k-induction method handles most of the trivial properties in the first iteration. If a property is not trivial for k-induction, the refinement phase usually results in an abstraction that is detailed enough for a proof by the PDR-based method or the BDD-based method in the second iteration. The average for the number of modules needed in a proof was 3.4 in the system consisting of 14 modules.

7. Conclusions

This paper presents a technique for model checking large modular systems. The main contributions are the development of an iterative abstraction refinement technique based on simple modular abstraction, and a refinement technique based on the dependency graph of the modules, and heuristic similar to delta debugging. The technique uses BDD-based model checking, k-induction, and PDR-based model checking concurrently to verify a particular abstraction, and k-induction to check the feasibility of abstract counter-examples. The technique can be used to analyse state invariant properties.

Our iterative verification technique is similar to other iterative refinement based verification techniques, but it differs in the way that it uses module structure to decompose the system in a natural way, so that abstractions of the system are simpler.

We have applied the technique to the verification of two systems: a fictional model that consists of two diverse safety systems, and an emergency diesel generator control system. We have compared the performance of the technique to some standard model checking algorithms. The results show that our technique is a feasible approach for the verification of large systems. When applied on the real life industrial emergency diesel control system, our technique outperformed the compared methods in 68% of the tested random true state invariants.

It is probable that the performance of our technique is dependent on the structure of the model's dependency graph. Both systems that were used in the tests comprised several individual subsystems, and could we divided into modules in a natural way. The resulting dependency graphs were quite balanced. For example, there were no modules that were dependent on overly many other modules. Such structures in the dependency graph could lead to the inclusion of all the model's modules when using our technique. Safety-critical systems may suit well with our technique because the various diverse and redundant functions of these systems inherently avoid unnecessary dependencies between other functions. We plan to test the performance of our technique on other classes of system models in future work.

We also realise that the efficiency of our technique relies on the length of the spurious counter-examples. The feasibility checking of longer counter-examples takes a lot more effort as the k-induction technique is used. This could be a potential problem for some types of systems.

In addition to verifying state invariants and general LTL properties, verifying safety subsets of temporal logics such as the syntactic LTL safety properties [35] and the IEEE 1850 Property Specification Language (PSL) safety properties [36] can be very efficiently reduced into model checking state invariant properties, and thus handled even more efficiently than generic LTL properties.

In the case studies, the models were manually created based on function block diagrams and other design documentation. To increase confidence in the correctness of the model and in the verification results, an automatic method for generating the model is needed. Such a method requires that the system is well-specified and in a machine-readable format. Automatic translation approaches based on the IEC 61131-3 standard exist, see e.g. [37] and [38]. In the diesel generator control system case study, however, only a textual description of the functionality of the vendor-specific function blocks was available for modelling, making an automatic translation method infeasible.

Acknowledgements

Funding from the SAFIR2014 programme (The Finnish Research Programme on Nuclear Power Plant Safety 2011–2014), and Academy of Finland projects 139402 and 277522 are gratefully acknowledged.

References

[1] Clarke EM, Grumberg O, Peled D. Model checking. Cambridge, Massachusetts, US: MIT Press; 2001 ISBN 978-0-262-03270-4.
[2] Baier C, Katoen JP. Principles of model checking. Cambridge, Massachusetts, US: MIT Press; 2008 ISBN 978-0-262-02649-9.
[3] Cimatti A, Clarke EM, Giunchiglia E, Giunchiglia F, Pistore M, Roveri M, et al. NuSMV 2: an opensource tool for symbolic model checking. In: Brinksma E, Larsen KG, et al., editors. CAV; Lecture notes in computer science, vol. 2404. Berlin, Heidelberg: Springer; 2002. p. 359–64 ISBN 3-540-43997-8.
[4] Eén N. The ABC/ZZ verification and synthesis framework. URL ⟨https://bitbucket.org/niklaseen/abc-zz⟩; 2014.
[5] AIGER: a format, library and set of utilities for And-Inverter Graphs (AIGs). URL ⟨http://fmv.jku.at/aiger/⟩; 2014.
[6] Biere A, Heljanko K, Junttila TA, Latvala T, Schuppan V. Linear encodings of bounded LTL model checking. Log Methods Comput Sci 2006;2(5:5):1–64.

[7] Biere A, Cimatti A, Clarke EM, Zhu Y. Symbolic model checking without BDDs. In: Cleaveland R, editor. TACAS; Lecture notes in computer science, vol. 1579. Springer; 1999. p. 193–207 ISBN 3-540-65703-7.
[8] Sheeran M, Singh S, Stålmarck G. Checking safety properties using induction and a SAT-solver. In: Hunt WA, Jr., Johnson SD, editors. FMCAD; Lecture notes in computer science, vol. 1954. Berlin, Heidelberg: Springer; 2000. p. 108–25. ISBN 3-540-41219-0.
[9] Eén N, Sörensson N. Temporal induction by incremental SAT solving. Electr Notes Theor Comput Sci 2003;89(4):543–60.
[10] Bradley AR. SAT-based model checking without unrolling. In: Jhala R, Schmidt DA, editors. VMCAI; Lecture notes in computer science, vol. 6538. Berlin, Heidelberg: Springer; 2011. p. 70–87 ISBN 978-3-642-18274-7.
[11] Yoo J, Jee E, Cha SD. Formal modeling and verification of safety-critical software. IEEE Softw 2009;26(3):42–9.
[12] Lahtinen J, Valkonen J, Björkman K, Frits J, Niemelä I, Heljanko K. Model checking of safety-critical software in the nuclear engineering domain. Reliab Eng Syst Saf 2012;105:104–13. http://dx.doi.org/10.1016/j.ress.2012.03.021.
[13] Gan X, Dubrovin J, Heljanko K. A symbolic model checking approach to verifying satellite onboard software. Sci Comput Program 2013. http://dx.doi.org/10.1016/j.scico.2013.03.005 Available online.
[14] Clarke EM, Gupta A, Strichman O. SAT-based counterexample-guided abstraction refinement. IEEE Trans CAD Integr Circuits Syst 2004;23(7):1113–23.
[15] Clarke EM, Long DE, McMillan KL. Compositional model checking. In: LICS. IEEE Computer Society; 1989. p. 353–62. ISBN 0-8186-1954-6.
[16] Sterin B, Een N, Mishchenko A, Brayton R. The benefit of concurrency in model checking. In: Proceedings of the international workshop on logic synthesis, IWLS'11; 2011. p. 176–82.
[17] Zeller A. Isolating cause-effect chains from computer programs. In: SIGSOFT FSE; 2002. p. 1–10.
[18] Zeller A, Hildebrandt R. Simplifying and isolating failure-inducing input. IEEE Trans Softw Eng 2002;28(2):183–200.
[19] Lahtinen J, Kuismin T, Heljanko K. CEGAR algorithm and fictional model. ⟨http://users.ics.aalto.fi/tlauniai/cegar-2013/⟩; 2014 [accessed September 2, 2014].
[20] Lahtinen J, Björkman K, Valkonen J, Frits J, Niemelä I. Analysis of an emergency diesel generator control system by compositional model checking. VTT Working Papers 156.VTT Technical Research Centre of Finland; 2010.
[21] Pakonen A, Mätäsniemi T, Lahtinen J, Karhela T. A toolset for model checking of PLC software. In: IEEE 18th conference on emerging technologies & factory automation (ETFA); 2013. p. 1–6. http://dx.doi.org/10.1109/ETFA.2013.6648065.
[22] Lahtinen J, Launiainen T, Heljanko K. Model checking methodology for large systems, faults and asynchronous behaviour. VTT Technology 12, VTT Technical Research Centre of Finland; 2012.
[23] Lahtinen J, Björkman K, Valkonen J, Niemelä I. Emergency diesel generator control system verification by model checking and compositional minimization. In: Kučera A, Henzinger TA, Nešetřil J, Vojnar T, Antoš D, editors. MEMICS 2012; 2012. p. 49–60. ISBN 978-80-87342-15-2.
[24] Sterin B, Een N, Mishchenko A, Brayton R. The benefit of concurrency in model checking. In: IWLS'11; 2011. p. 176–82.
[25] Kurshan RP. Computer-aided verification of coordinating processes: the automata-theoretic approach. Princeton, NJ, USA: Princeton University Press; 1994 ISBN 0-691-03436-2.
[26] Balarin F, Sangiovanni-Vincentelli AL. An iterative approach to language containment. In: Proceedings of the 5th international conference on computer aided verification. CAV '93. London, UK: Springer-Verlag; 1993. p. 29–40. ISBN 3-540-56922-7.
[27] Clarke E, Grumberg O, Jha S, Lu Y, Veith H. Counterexample-guided abstraction refinement for symbolic model checking. J ACM 2003;50(5):752–94. http://dx.doi.org/10.1145/876638.876643.
[28] Das S, Dill DL. Successive approximation of abstract transition relations. In: Proceedings of the 16th annual IEEE symposium on logic in computer science. LICS '01. Washington, DC, USA: IEEE Computer Society; 2001, p. 51–58.
[29] Clarke EM, Gupta A, Kukula JH, Shrichman O. SAT based abstraction-refinement using ilp and machine learning techniques. In: Proceedings of the 14th international conference on computer aided verification. CAV '02. London, UK: Springer-Verlag; 2002. p. 265–79. ISBN 3-540-43997-8.
[30] McMillan K, Amla N. Automatic abstraction without counterexamples. In: Garavel H, Hatcliff J, editors. Tools and algorithms for the construction and analysis of systems, Lecture notes in computer science, vol. 2619. Berlin, Heidelberg: Springer; 2003. p. 2–17 ISBN 978-3-540-00898-9.
[31] Glusman M, Kamhi G, Mador-Haim S, Fraer R, Vardi MY. Multiple-counterexample guided iterative abstraction refinement: an industrial evaluation. In: Proceedings of the 9th international conference on tools and algorithms for the construction and analysis of systems. TACAS'03. Berlin, Heidelberg: Springer-Verlag; 2003. p. 176–91. ISBN 3-540-00898-5.
[32] Amla N, McMillan K. Combining abstraction refinement and SAT-based model checking. In: Grumberg O, Huth M, editors. Tools and algorithms for the construction and analysis of systems; Lecture notes in computer science, vol. 4424. Berlin, Heidelberg: Springer; 2007. p. 405–19. http://dx.doi.org/10.1007/978-3-540-71209-1_31 ISBN 978-3-540-71208-4.
[33] Zheng H, Yao H, Yoneda T. Modular model checking of large asynchronous designs with efficient abstraction refinement. IEEE Trans Comput 2010;59(4):561–73. http://dx.doi.org/10.1109/TC.2009.187.
[34] Wang D, Jiang PH, Kukula J, Zhu Y, Ma T, Damiano R. Formal property verification by abstraction refinement with formal, simulation and hybrid engines. In: Proceedings of the 38th annual design automation conference. DAC'01. New York, NY, USA: ACM; 2001. p. 35–40. ISBN 1-58113-297-2. http://dx.doi.org/10.1145/378239.378260.
[35] Latvala T. Efficient model checking of safety properties. In: Proceedings of 10th International SPIN workshop on model checking software, Portland, OR, USA, May 9–10, 2003, Lecture notes in computer science, vol. 2648. Berlin, Heidelberg: Springer; 2003. p. 74–88.
[36] Launiainen T, Heljanko K, Junttila TA. Efficient model checking of PSL safety properties. IET Comput Digit Tech 2011;5(6):479–92.
[37] Yoo J, Cha S, Jee E. Verification of PLC programs written in FBD with VIS. Nucl Eng Technol 2009;41(1):79–90.
[38] Soliman D, Thramboulidis K, Frey G. Transformation of function block diagrams to UPPAAL timed automata for the verification of safety applications. Annu Rev Control 2012;36(2):338–45.

Publication IV

Jussi Lahtinen. Verification of fault-tolerant system architectures using model checking. In *1st International Workshop on Development, Verification and Validation of Critical Systems (DEVVARTS)*, Lecture Notes in Computer Science, volume 8696, p. 195 – 206, Springer 2014.

Publication V

Jussi Lahtinen. Automatic test set generation for function block based systems using model checking. In *9th International Conference on the Quality of Information and Communications Technology (QUATIC 2014)*, Guimarães, Portugal, p. 216 – 225, IEEE 2014.

Automatic test set generation for function block based systems using model checking

Jussi Lahtinen
VTT Technical Research Centre of Finland
P.O. Box 1000
Espoo, Finland
Email: jussi.lahtinen@vtt.fi

***Abstract*—Many nuclear instrumentation and control (I&C) systems are designed using a function block diagram description of the system. Strict requirements pertain to the verification of these systems. Different verification techniques, including structure-based testing, are demanded by standards and the regulators. Unfortunately, the traditional structure-based test techniques intended for software code are not directly applicable to function block diagrams. However, coverage criteria for function block diagrams have recently been developed. In this work we have used these coverage criteria and developed a technique for generating structure-based test sets for function block based designs. The test set is automatically generated but the technique requires that a model checking model of the system is available. The technique utilises model checking to determine the concrete test cases. We have also described how tests can be generated so that multiple test requirements can be fulfilled at once, thus decreasing the number of generated test cases. We have implemented our approach as a proof-of-concept tool, and demonstrated the technique on a case study system.**

I. Introduction

Digital instrumentation & control (I&C) systems are increasingly being used for implementing safety-critical applications such as nuclear power plant safety systems. These systems have to be adequately verified using methods including simulation, formal methods, and predominantly testing. The ISO/IEC 29119-4 [1] standard divides test techniques into three categories: specification-based, structure-based, and experience-based testing. In the nuclear automation domain, specification-based and structure-based testing techniques are commonly used. Specification-based testing means that the tests are derived from the requirement specification of the system, while structure-based tests are derived directly from the structure of the system. The use of both testing techniques is required by the nuclear regulators, e.g. the USNRC Regulatory Guide 1.171 [2]. Another motivation for structure-based testing is that other forms of testing are specification-centric, and their success depends a lot on whether the requirements of the system have been sufficiently specified. Structure-based testing can provide test cases that help identify omissions in the system requirement specification.

Structure-based test design techniques can be subdivided into control flow based and data flow based techniques. A program can be modelled as a control flow graph, in which all the possible execution sequences are represented as paths of the graph. The control flow test techniques such as statement testing, branch testing and decision testing define coverage with respect to this graph. Data flow testing, on the other hand, focuses on points at which variables receive values and points at which these values are used, and defines coverage with respect to these events.

Many nuclear I&C systems are designed using a function block representation of the system. Function Block Diagram (FBD) as defined in the IEC standard 61131-3 [3] is a commonly used graphical programming language for programmable logic controllers, in which the design consists of inputs, outputs, and a set of simple elementary function blocks such as AND, OR, or timer function blocks, and the connections between these components. An example function block diagram following the graphical notations used in this paper can be seen in Fig. 1.

The IEC 61131-3 standard is not always strictly followed in real applications and other vendor-specific implementations are typical. For example, vendor-specific function blocks are used in the design phase of application functions in the AREVA's TXS platform. The application functions are then converted into C code using an automatic code generator.

Applying structure-based testing to automatically generated code is undesirable as the test cases can become non-intuitive and difficult to understand. One alternative to this is to determine the structure-based tests on the level of the function block diagram. However, the traditional structure-based test techniques are not directly applicable.

Function block diagrams are fundamentally different from code when it comes to testing, and the traditional definitions of code coverage do not apply. Especially the control flow methodology is not applicable. In code, only part of the code is covered in a single test case. In function block diagrams the whole system is usually[1] "covered" on every time step, i.e. all function blocks have some input, and produce some

[1]Different function block based design paradigms exist. The function block diagrams as defined in IEC 61499 are executed in an event-based manner. For these function blocks the control flow testing techniques might be more suitable.

output.

In order to be able to perform structure-based testing one must first define a test coverage criterion that describes the degree to which a particular system is tested by a set of tests. Jee et al. [4], [5] have developed some novel coverage criteria that can be used as a basis for planning structure-based tests for function block diagrams. The coverage criteria are based on interpreting the system as a data flow diagram, and generating a set of test requirements that the tests have to fulfil.

These coverage criteria offer a good basis for planning structure-based tests. On trivial systems tests can be manually composed. On more complex system designs that contain memories, timers and feedback loops, however, it can be quite difficult to come up with a test case that fulfils a given test requirement. Furthermore, in larger systems the number of test requirements can be rather high. It would be practical to design the tests so that a single test case will fulfil multiple test requirements simultaneously, and that the number of tests is small. Doing this manually is infeasible, and an automatic technique for determining the test cases can be of considerable use.

Model checking [6] is a formal method that can be used for analysing the behaviour of a system exhaustively. In model checking, a model of the system is written and the system specifications are formalised in a suitable language, e.g. temporal logic. A model checking tool then analyses the model against the temporal logic clause in a way that takes all possible system behaviours into account. If it is possible to violate the specification in the model, the model checking tool gives a concrete counter-example as output that demonstrates on variable-level how the violation might occur. This ability to produce concrete counter-examples can also be useful for generating test cases. The classic way to use model checking for test generation is to take the negation of some system requirement and formalise that in temporal logic. When the resulting formula is analysed using the model checking tool a counter-example will be produced that is according to the original requirement. The inputs and expected outputs of the test case can then be read from the counter-example.

In our previous work (see e.g. [7]) we have used model checking for analysing and verifying properties of safety-critical nuclear domain systems. In this paper we combine model checking and test case generation. We use the coverage criteria designed for function block based systems and introduce a novel approach for generating a test set that has maximum coverage according to these criteria. The test requirements established by using the coverage criteria are transformed into suitable temporal logic formulas, and analysed against a model of the system. We also describe how tests can be generated so that multiple test requirements can be fulfilled at once, thus decreasing the number of generated test cases. For model checking, we employ both

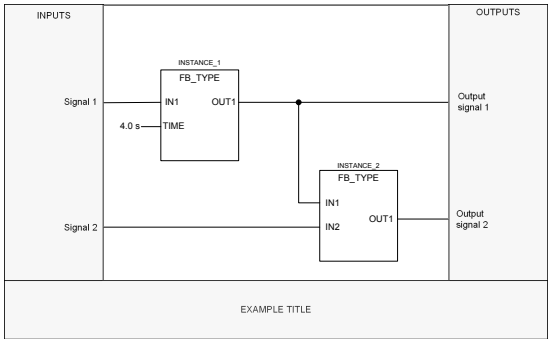


Figure 1. An example of a function block diagram (FBD)

Binary Decision Diagram (BDD) -based [6] and bounded model checking (BMC) -based [8] algorithms provided by the model checker NuSMV 2.5.4 [9]. In BMC, only model executions of length up to k are examined, and the model checking problem is translated into a propositional satisfiability problem (SAT) that can be solved using SAT-solvers.

We have implemented our approach as Python code, and demonstrate the implementation on a small example system. An early version of the technique and the implementation are explained in more detail in a research report [10].

II. Related work

The first papers on the subject of using model checking to generate tests were by Callahan et. al [11] and Engels et. al [12]. These papers introduce the classic way of generating test cases, in which the negation of a specification is model checked in order to create a test case according to the original specification. Since these pioneering papers, the general test generation idea has been adapted in a variety of applications. An extensive survey on research combining test generation and model checking can be found in [13].

The work by Enoiu et. al [14] also discusses function block diagrams in the context of test generation using model checking. They have used the model checker UPPAAL to generate a test set for function block diagrams. They have defined their own coverage criteria for function block diagrams. However, no special heuristic algorithms to minimise the number of test cases are used in their work.

Several papers have addressed the efficiency of the test set, and the efficiency of the test generation process. For example, Ammann et. al [15] reduce the test set size simply by removing duplicate test cases and by dropping the counter-examples which are a prefix to another counter-example.

The paper by Hamon et. al [16] is concerned with methods for efficient generation of test sets. In their technique, called iterated extension, a model checking tool is modified so that

in addition to searching for counter-examples in a regular manner, the tool also searches for extensions to previously found counter-examples.

There are also techniques for generating efficient test sets that are not necessarily related to model checking. For example, automatic test pattern generators (ATPG's) use test compaction to reduce the overall size of the tests. The paper by Niermann et. al [17] desribes a number of heuristic algorithms used for compacting a set of tests generated by a sequential circuit automatic test pattern generator.

Test cases are generated for Boolean form expressions in [18]. The approach is based on fault classes for hypothesised faults. Test predicates are formed by applying the various fault classes on the Boolean expressions. The set of test predicates is optimised using a SAT solver or a SMT solver. One of their optimisation strategies, called test collecting, considers several independent faults at once in order to find a single test case that detects all of them.

In a paper by Fraser and Arcuri [19] a genetic algorithm is used in order to generate a test suite for software code in which all coverage goals are covered at the same time while size of the test suite is kept as small as possible.

Test set efficiency is also discussed in a paper by Campos et. al [20]. The objective in their work is to improve fault localization in software by generating a test set based on its ability to diagnose the fault. This is another aspect of test set efficiency that can be very useful for debugging.

Our work in contrast to the related work focuses on structure-based tests in the context of function block diagrams. In addition, we also propose a simple heuristic algorithm to reduce the size of the test set.

III. Example system description

A small function block based system, illustrated in Fig. 2, is used as a running example to demonstrate the test coverage criteria and our test generation technique. The example is a stepwise shutdown system (adapted from [21]) that has been designed as a preventive safety system to drive a process into a normal operating state without having to rapidly shut the process down. It can be triggered by a process input (e.g. high measurement value) or by the operator using a manual trip command. A 14 s control cycle is used that consists of a 4 s control followed by 10 s idle time after which the cycle is started again if the measurements are still high. In addition, the operator can add 4 s control cycles manually if the 10 s idle time seems too long. The design contains an intentional error: if the manual trip command is given during the 4 s control the system freezes until the process input disappears. The design error is left in the example so that we can see whether the generated tests will be able to detect the error.

IV. Test coverage criteria for function block diagrams

Jee et al. [4], [5] have developed three test coverage criteria for structure-based testing of function block diagrams: basic coverage (BC), input condition coverage (ICC), and complex condition coverage (CCC). To the best of our knowledge, test coverage criteria for function block diagrams prior to these did not exist. The coverage criteria are based on interpreting the function block diagram as a data flow graph, and calculating the data paths of that graph. A set of test requirements is then written based on the data paths. In what follows, we briefly go through the relevant definitions related to the coverage criteria.

The function block diagram F is defined as a tuple $F = \langle FBs, V, E \rangle$, where FBs is a set of function blocks, V is a set of variables, and E is a set of edges. An edge is defined as a connection between two function blocks or a function block and a variable. Function blocks can be defined with respect to the edges. For example, the function block AND is defined as: $e_{OUT} = AND(e_{IN1}, e_{IN2})$, where e_{OUT} is the output edge of the function block and e_{IN1} and e_{IN2} are the input edges.

A data path is defined as a finite sequence $\langle e_1, e_2 \dots e_n \rangle$ of edges where all the edges succeed one another. Since data paths are finite, any internal feedback loop in a function block diagram needs to be somehow handled. We handle feedback loops by creating an alternative version of the function block diagram in which all feedback loops are disconnected. New input signals are added to replace the feedback signals. This alternative version is used for determining the data paths and test requirements of the system.

The example system has a single feedback loop. Fig. 3 illustrates the example system with the feedback loop disconnected. The loop-inducing edge has been replaced with a new input $Feedback$. In addition, we have left the DELAY function block out of examination in the example system in order to simplify the example. DELAY function blocks are problematic as the methodology does not currently provide proper means to handle them adequately (see Section VI).

Once the feedback loop is removed it is straight-forward to calculate the data paths. The example system has eight data paths. Note that the time parameters of the PULSE blocks are also considered as inputs of the system. One of the data paths of the example system is highlighted in Fig. 3.

A function condition (FC) is the logical condition under which the output edge e_o of a function block is influenced by the value at the input edge e_i. If internal variables of the function block need to be analysed to determine the relationship between an input and an output, the condition is called a function block condition (FBC).

For example, the FCs and FBCs of the example system are presented in Table I. The conditions of the example system were determined manually following the convention

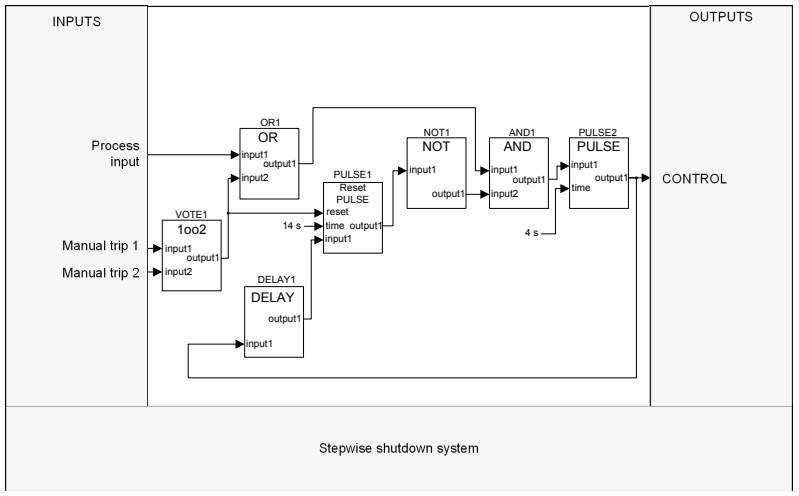


Figure 2. Stepwise shutdown system

Table I
THE FUNCTION CONDITIONS AND FUNCTION BLOCK CONDITIONS REQUIRED FOR THE ANALYSIS OF THE EXAMPLE SYSTEM

Function block	**FCs or FBCs**
AND	$FC_{AND}(\langle input_1, output\rangle) = \neg input_1 \vee input_2$ $FC_{AND}(\langle input_2, output\rangle) = \neg input_2 \vee input_1$
NOT	$FC_{NOT}(\langle input_1, output\rangle) = TRUE$
PULSE	$FBC_{PULSE}(\langle input_1, output\rangle) = \neg(clock > 0)$ $\wedge\neg prev \wedge \neg prevout$ $FBC_{PULSE}(\langle time, output\rangle) = clock > 0$
OR	$FC_{OR}(\langle input_1, output\rangle) = input_1 \vee \neg input_2$ $FC_{OR}(\langle input_2, output\rangle) = input_2 \vee \neg input_1$
1oo2	$FC_{1oo2}(\langle input_1, output\rangle) =$ $input_1 \vee \neg input_2$ $FC_{1oo2}(\langle input_2, output\rangle) =$ $input_2 \vee \neg input_1$
Reset PULSE	$FBC_{ResetPULSE}(\langle input_1, output\rangle) =$ $\neg(clock > 0) \wedge \neg prev \wedge \neg prevout \wedge \neg reset$ $FBC_{ResetPULSE}(\langle reset, output\rangle) = reset \vee$ $(\neg prev \wedge \neg prevout \wedge input_1) \vee (clock > 0)$ $FBC_{ResetPULSE}(\langle time, output\rangle) =$ $(clock > 0) \wedge \neg reset$

described in [4], [5]. In practice, the FBCs were manually deduced from a list of all input and output combinations of relevant variables of a function block. The function block conditions of the two PULSE function blocks refer to internal variables ($prev$, $prevout$, $clock$) used for implementing them. The variable $prev$ stores the previous value of the input, $prevout$ stores the previous value of the output, and $clock$ is a counter that is started whenever the pulse begins. For more detail, the NuSMV implementation of the example system is presented in [10].

A data path condition (DPC) is the condition along the data path under which the input value plays a role in computing the output. It can be composed as the conjunction of the function block conditions on that path. The data path condition corresponding to the data path in Fig. 3 in is: $(process\_input \vee \neg VOTE1.output) \wedge ((\neg OR1.output) \vee NOT1.output) \wedge (\neg(PULSE2.clock > 0) \wedge \neg\ PULSE2.prev \wedge \neg PULSE2.prevout)$.

The coverage criterion and the data path conditions are used to generate a set of test requirements. The BC coverage criterion (BC) is met when each DPC is fulfilled by one of the test cases.

The input condition coverage (ICC) criterion requires that for each **Boolean input of a data path**, there is a test case in which: 1) the DPC is fulfilled and the input is false; 2) the DPC is fulfilled and the input is true.

The complex condition coverage (CCC) criterion is even more demanding. It requires that for each **Boolean variable within a data path**, there is a test case in which: 1) the DPC is fulfilled and the variable is false; 2) the DPC is fulfilled and the variable is true.

A test requirement is fulfilled by a test case that at some point drives the system to a state in which the test requirement evaluates to true. If all of the test requirements can be fulfilled by one of the tests, 100 % test coverage is achieved. In many cases, however, some of the test requirements can be infeasible, due to e.g. timings in the design.

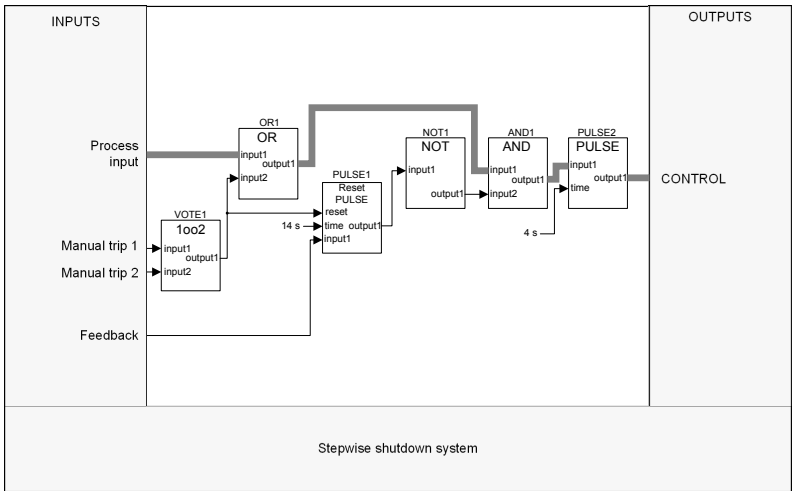


Figure 3. The figure illustrates the example system with the feedback loop disconnected. One of the data paths of the system is highlighted.

V. Test set generation using model checking

A. Using model checking for generating test cases

Our technique for using model checking to generate test cases for function block diagrams is illustrated in Fig. 4. The prerequisites for the technique are that the function conditions and function block conditions have been defined for each function block type, and that the function block diagram is modelled as a model checking model. A methodology for modelling function block diagrams already exists; see e.g. [22]. The model checking methodology suits well for designs where Boolean logic is used together with timers and memories. However, complex functionality such as PID controllers are too complex to be used for model checking. Another limitation is that the model checking tool used in this work does not support analog variables, and therefore analog variables as well as time are discretised in the model.

Once the initial information has been acquired, the data paths of the system are identified, and a set of test requirements is written. This is straight-forward work and can be done automatically if the system design is in computer readable form.

Next we need to define a test case that fulfils a given test requirement. Each test requirement can be transformed into a temporal logic clause stating that the test requirement can never be fulfilled. In linear temporal logic (LTL) the property is specified as $\mathbf{G}\neg(testRequirement)$.

The model checking tool can be used to evaluate the temporal logic clause against the system model. If a path exists to a state in which the test requirement is fulfilled, it is given as a counter-example. The system inputs used for a test case and the expected outputs of the system can be read from the counter-example.

B. Test set generation algorithm

It is possible to create test cases that fulfil multiple test requirements at once. This can be done by combining two (or several) temporal logic formulas into a single formula that covers all the associated test requirements. As an example, assume we have two test requirements: $testRequirement_1$ and $testRequirement_2$. The corresponding temporal logic formula covering these two test requirements is:

$\mathbf{G}\neg(testRequirement_1) \vee \mathbf{G}\neg(testRequirement_2)$.

The temporal logic formula is also equivalent to:

$\neg(\mathbf{F}(testRequirement_1) \wedge \mathbf{F}(testRequirement_2))$.

The formula states that no path exists, in which each test requirement is true at some time point during the test. If a path exists that fulfils both test requirements it will be output by the model checking tool as a counter-example.

It is possible to create a single temporal logic formula encompassing all of the test requirements. However, such a test may be very complex, or consist of very many time steps. Some test requirements can also be infeasible, and these cases should be detected and sorted out.

We developed an automatic test set generation algorithm with the intention of keeping the test cases simple, and the number of tests low. The algorithm is presented as pseudo code in Algorithm 1. In the procedure $GenerateTestSet$, we assume that the set of test requirements R has already been calculated, and that the system FBD has been modelled. The procedure makes calls to another function $runMC$ that creates the temporal logic formula corresponding to the set of currently examined test requirements, and performs the model checking on the model, and returns two elements: a Boolean variable $testfound$ that expresses whether a suitable test case could be found, and the counter-example file ce, if

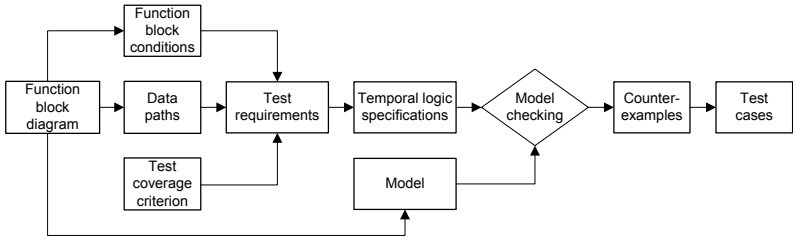


Figure 4. Test generation using model checking

one exists.

The algorithm begins with the first test requirement and determines whether a test case for that single test requirement can be found. If it can be found we attach a new test requirement to the examined set of test requirements, and find out whether a counter-example of the same length that fulfils all test requirements in the set can be found. If such a counter-example is still possible we continue by attempting to add even more test requirements to the set. If the counter-example becomes infeasible, we exclude the most recent test requirement and continue by adding one from the set of unexamined test requirements. Once all test requirements have been gone through, we have a single test case that fulfils n out of the $|R|$ test requirements. The process is then repeated with the $|R| - n$ remaining test requirement until every test requirement is covered by some test case, or it has been determined that the test requirement is infeasible.

For simplicity, the pseudo code presentation is missing some of the functionality of our algorithm. We use a bounded model checking (BMC) algorithm to quickly look for short test cases. If the BMC algorithm does not find a counter-example within its bound, we employ a BDD-based algorithm as a backup, since the BDD-based approach will find counter-examples of any length. We also store the lengths of the already found counter-examples and use them to guide the BMC algorithm: when the set of currently examined test requirements grows, the length of the test fulfilling the requirements is kept unchanged if possible. Finally, we also create a mapping from the test requirements to the test cases that fulfil them.

Algorithm 1 Test set generation algorithm

```
1:  procedure GenerateTestSet(R, FBD)
2:      Checked ← ∅                          ▷ Covered test requirements
3:      Infeasible ← ∅                       ▷ Infeasible test requirements
4:      Current ← ∅                          ▷ Current set of requirements
5:      Tests ← ∅                            ▷ Resulting test set
6:      for x ← 0, |R| − 1 do
7:          if R[x] ∈ Checked then
8:              continue
9:          end if
10:         Current ← ∅
11:         for y ← x, |R| − 1 do
12:             if R[y] ∈ Checked then
13:                 continue
14:             end if
15:             Current ← Current ∪ R[y]
16:             [testfound, ce] = runMC(FBD, Current)
17:             if not testfound then
18:                 if |Current = 1 then
19:                     Infeasible ← Infeasible ∪ R[y]
20:                     Checked ← Checked ∪ R[y]
21:                     break
22:                 end if
23:                 if |Current| > 1 then
24:                     Current ← Current − R[y]
25:                 end if
26:             else                         ▷ Test found
27:                 Checked ← Checked ∪ R[y]
28:             end if
29:             if ∃z : R[z] ∉ Checked, z > y then
30:                 continue
31:             else
32:                 Tests ← Tests ∪ ce
33:                 break
34:             end if
35:         end for
36:     end for
37: end procedure
```

C. Implementation

The test set generation technique was implemented as a prototype tool using Python. The tool requires a model checking model as input. The model is annotated so that parts referring to the inputs, outputs and function blocks of the model can be easily recognised. The tool then deduces the structure of the function block diagram based on the model. In addition to the model file, we also use a separate file for relevant information regarding the different function block types, e.g. names and types of the output signals of function blocks, and most importantly the function conditions and function block conditions related to each input-

output pair.

Function block diagrams may have diverging connections between the function blocks. By this we mean that an output of a function block is used as input in multiple other function blocks. According to the used methodology, however, a connection has to be between two function blocks or a function block and a variable. Our tool examines the structure of the FBD, and if diverging connections are detected, dummy function blocks are added to the branch points. The dummy function blocks have a single input and several outputs, and they simply forward the input signal to the outputs.

We also implemented a loop removal feature in the tool. Loops are removed automatically via a backwards depth-first-search starting from the outputs of the FBD. If an edge is encountered during the search that has been previously visited, then a loop has been detected. The loop removal procedure is used only for creating the test requirements of the system. The actual tests are generated for the original system that still has all feedback loops intact.

VI. Discussion

We have identified some issues that complicate the utilisation of the structure-based coverage criteria as defined in [4] and [5]. First, there is a slight difference in the definition of the function block conditions of the coverage criteria by Jee et. al, and the input-output condition as defined in another widely used coverage criterion called Modified Condition/Decision Coverage (MC/DC) (see IEC 29119-4). For example, according to Jee et. al the function condition for one of the inputs of an AND function block is:

$FC\langle input_1, output\rangle = (\neg input_1) \vee input_2$

The condition states that when $input_1$ is false, it has influence on $output$, and if $input_1$ is true, it influences the output only when $input_2$ is true as well. A similar input-output influence relation is defined in the MC/DC coverage criterion. In MC/DC the influence relation is somewhat different: input has influence on the output when flipping of the input value, also flips the output value. If the function condition for the AND function block was written based on this definition, it would be:

$FC\langle input_1, output\rangle = input_2$

That is, $input_1$ has influence on $output$ only when the other input is true. The difference in these two definitions is the case where both inputs are false. According to the definition by Jee et. al $input_1$ has influence on the output since $input_1$ is one of the inputs that are false. According to the MC/DC definition $input1$ does not have influence on $output$. In our opinion the MC/DC definition is more intuitive and leaves less room for interpretation.

Another issue is that the current methodology by Jee et. al does not adequately take the time dimension into account in the function block conditions. Only the instantaneous influence of an input to an output is considered. However, in some cases an input may only have a delayed influence on an output signal. A perfect example is the DELAY function block. If its function block condition is defined in strict terms we realise that the input has no direct influence on the output, and the function block condition becomes: $FBC\langle input_1, output\rangle = FALSE$. This causes all the test requirements involved with a data path that has a DELAY function block to become infeasible. This seems counter-intuitive since there is a clear dependency between the input of the DELAY and the output. However, setting the function block condition to *TRUE* could cause test cases to be generated, in which the input of a data path does not influence the output due to a time delay within the path.

The input of the DELAY function block at time point n influences the output at time point $n+1$. The methodology should be further developed so that the time aspect is somehow taken into account.

Table II
Information on the test generation for the example system

Coverage criterion	Test Requirements	Infeasible Requirements	Test cases	Achieved coverage
BC	8	0	1, 3a	100%
ICC	14	1	1, 2, 3b	92,9%
CCC	80	10	1, 2, 3c	87,5%

VII. Results

In this section we present the generated tests for the example system. In addition, we present results of using the developed tool for test generation on a small group of other function block diagrams. The tests were generated on a PC with Intel Core i7 Q740 processor and 3 GB of RAM. For model checking, NuSMV version 2.5.4 was used.

A. Example system

Test cases were generated for the example system based on the three coverage criteria (BC, ICC and CCC). Information related to the test generation process is illustrated Table II.

Applying the coverage criteria to the system resulted in eight test requirements for the BC criterion, 14 test requirements for the ICC criterion, and 80 test requirements for the CCC criterion. All BC test requirements were feasible. One ICC test requirement was infeasible because it required that the feedback signal was true while the internal memory indicating the previous control output value was false. In the actual system where the feedback loop is intact these two signals are the same signal which causes the requirement to be infeasible. Ten out of the 80 CCC test requirements were infeasible. As an example of the infeasible cases, one of the test requirements states that the output of a PULSE function block is false while the internal clock of the PULSE is running. This cannot occur in the system because the output is set whenever the clock is running.

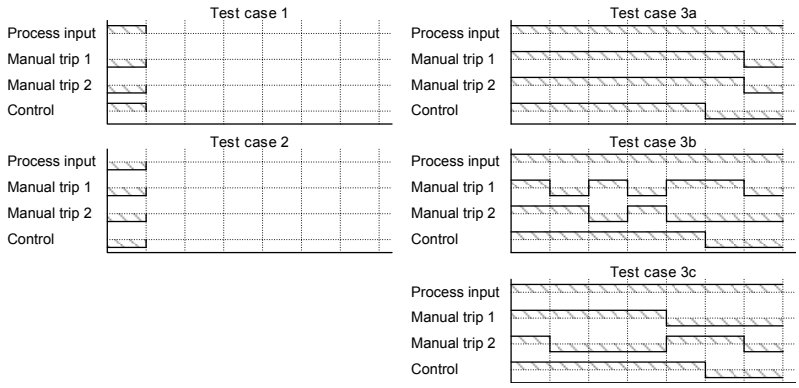


Figure 5. Timing diagrams of the generated test cases

The timing diagrams of the generated test cases are illustrated in Fig. 5. The BC criterion led to test cases 1 and 3a, ICC led to test cases 1, 2 and 3b, and CCC led to test cases 1, 2 and 3c. Test case 1 consists of a single time point at which the control output should be set when the process input is true, and the manual trip commands are false. Test case 2 consists of a single time point as well and it requires that the control output is not set when all inputs are false. All criteria led to a single longer test case (test cases 3a, 3b and 3c). In these tests the process input is true until the control output becomes false after the 4 s pulse. The three test cases have minor differences in how the manual trip commands alternate during the pulse and after it.

Three test cases suffice for fulfilling all feasible test requirements of the case study system, even when the most rigorous coverage criterion (CCC) is used. In the example system the benefit of using the CCC criterion when compared to the less rigorous ICC criterion cannot be seen. This is probably due to the simplicity of the case study system. However, the difference between BC and the more rigorous coverage criteria can be seen: BC results in two tests while ICC and CCC results in three test cases. The ICC and CCC test cases are also more intricate.

The case study system was chosen because it includes a design error: if the manual trip command is given during the 4 s control the system freezes until the input disappears. In the generated test cases the error triggering behaviour is present but the end effect of the error can not be seen. The manual trip command is indeed given during the 4s control in the test cases 3a, 3b and 3c. Unfortunately, the generated test cases are too short to show the freeze of the output, and thus the error can not be identified only based on the generated tests. With the help of the test cases, however, a system designer may notice that a scenario in which the operator gives the trip command at a wrong time has not been considered.

Table III
FUNTION BLOCK DIAGRAMS USED AS INPUT FOR TEST GENERATION

FBD name	Inputs	Outputs	Function blocks	Data paths
FBD1	3	1	6	3
FBD2	2	1	4	2
FBD3	1	1	1	1
FBD4	1	4	14	15
FBD5	4	4	12	18
FBD6	1	1	6	3
FBD7	1	2	5	6
FBD8	3	2	8	15
FBD9	2	5	10	10

B. Test generation results

Using the developed tool, we generated structure-based tests for nine exemplar function block diagrams. Some statistics of these function block diagrams are presented in Table III. For these tests we defined the input-output relations of the function blocks following the MC/DC methodology, unlike in the running example. We also did not consider parameters of the function blocks as separate inputs as was the case in the running example. The function block condition of the DELAY function block was defined as: $FBC\langle input_1, output\rangle = FALSE$ according to the MC/DC principles.

The test generation results for different coverage criteria are summarised in Tables IV, V and VI. In addition to the number of total and infeasible test requirements, the number of generated tests, the number of performed model checker executions, and total test generation time are shown. Our test generation technique is able produce compact test cases, since the number of test cases remains low even when the number of total test requirements is high.

We can also see that the number of infeasible test requirements is quite high. This is due to the use of DELAY function blocks in systems FBD4, FBD5, FBD6, FBD7,

Table IV
TEST GENERATION RESULTS WHEN THE BC TEST COVERAGE CRITERION IS USED

FBD name	Test reqs.	Infeasible test reqs.	Test cases	Model checker runs	Elapsed time
FBD1	3	0	1	3	3 s
FBD2	2	0	1	2	2 s
FBD3	1	0	1	1	1 s
FBD4	15	9	2	23	71 s
FBD5	18	12	1	29	38 s
FBD6	3	1	1	4	5 s
FBD7	6	3	2	8	10 s
FBD8	15	5	2	28	29 s
FBD9	10	7	1	15	20 s

Table V
TEST GENERATION RESULTS WHEN THE ICC TEST COVERAGE CRITERION IS USED

FBD name	Test reqs.	Infeasible test reqs.	Test cases	Model checker runs	Elapsed time
FBD1	4	0	1	4	4 s
FBD2	2	0	1	2	2 s
FBD3	2	0	2	3	3 s
FBD4	15	9	2	23	70 s
FBD5	36	24	5	119	127 s
FBD6	6	3	2	12	14 s
FBD7	12	7	3	20	24 s
FBD8	30	12	5	110	106 s
FBD9	20	14	2	42	51 s

Table VI
TEST GENERATION RESULTS WHEN THE CCC TEST COVERAGE CRITERION IS USED

FBD name	Test reqs.	Infeasible test reqs.	Test cases	Model checker runs	Elapsed time
FBD1	20	0	3	32	31 s
FBD2	12	0	3	24	22 s
FBD3	4	0	2	6	5 s
FBD4	212	136	5	712	1123 s
FBD5	262	182	5	902	999 s
FBD6	34	22	2	75	102 s
FBD7	70	45	3	113	152 s
FBD8	202	97	5	731	712 s
FBD9	140	104	2	287	347 s

FBD8 and FBD9. As noted in section VI all data paths that are involved with a DELAY function block produce infeasible test requirements.

The number of test requirements can become quite high when the CCC coverage criterion is used (Table VI). Subsequently, the model checker has to be run hundreds of times, and the examined temporal logic formulas become long. Our test generation technique can manage these FBDs with a higher number of test requirements because the individual model checker runs remain short in duration. The test generation times, however, can still be quite high for a relatively simple function block diagram.

VIII. THREATS TO VALIDITY

In our technique we create a small number of tests that achieve high structure-based coverage by fulfilling multiple test requirements simultaneously. A threat to *construct validity* is that small tests may not be desirable if there is also need to e.g. identify which test requirement is the one localizing an error in the system. It can also be quite difficult to determine the correctness of the outputs of a test case when multiple test requirements are fulfilled at once.

Threats to *internal validity* might come from errors in the implementation code, the model checking model, and the function block conditions that were manually composed. To reduce possible errors in these components, the implementation was tested on many function block diagrams other than the one used as a running example, and the intermediate products of the implementation such as data paths and data path conditions and test requirements were manually reviewed.

The most important threat to *external validity* is whether the methodology can be extended to cover also function blocks that have no direct combinatorial dependencies between the inputs and the outputs (such as the DELAY function block). In addition, our technique depends on the fact that the system can be adequately described in the modelling language of the model checking tool. Designs including complex mathematical functions cannot be exactly modelled. Also, analog variable ranges have to be discretised for the tool.

IX. CONCLUSIONS

In this work we have introduced a technique for using model checking to generate structure-based test cases for function block diagrams. We have also presented an algorithm for generating a set of test cases that has high structure-based coverage. The test set is automatically generated but the technique requires that a model checking model of the system is available. The resulting test set is efficient in the sense that the number of test cases is small and the tests are concise. We have implemented the algorithm using the Python programming language and have demonstrated the use of the technique in a small case study system, and presented test generation results on a small group of function block diagrams.

The main application of our technique are safety-critical function block based systems, but the technique can be used for other relatively simple function block based systems consisting of Boolean logic, timers and memories. Based on our experience the technique should scale to typical nuclear domain safety systems, as long as only the behaviour of the single system is included in the analysis.

Our test set generation technique currently requires a fair amount of manual work. A model checking model of the examined system has to be available, and the input-output relations of the function blocks have to be manually

analysed. In future, we plan to determine a way to generate these conditions in a more automatic fashion. This automatic generation could be based on iterative counter-example guided model checking of a single function block. Using the MC/DC definition for defining the input-output relations seems more intuitive and more suitable for automatic generation purposes. We also plan to further extend the methodology so that the time dimension is taken into account in the definition of the function block conditions.

Finally, we plan on refining the test generation algorithm as many optimisations to the algorithm are possible. The test generation process can take a considerable amount of time. Infeasible test requirements may cause many redundant model checker executions. The test generation time could be shortened by trying to detect the infeasible test requirements as soon as possible so that the number of redundant model checker executions remains low. It may also be useful to check whether a previously generated counter-example already fulfils a test requirement without running the model checker (similar to monitoring of faults in [18]).

REFERENCES

[1] IEC, *ISO/IEC 29119-4 (2013): Software and systems engineering — Software testing — Part 4: Test techniques*, 2013.

[2] USNRC, "Software Unit Testing for Digital Computer Software Used in Safety Systems of Nuclear Power Plants, Regulatory Guide 1.171," 1997.

[3] IEC, *IEC 61131-3 (2013): International Standard for Programmable Controllers — Part 3: Programming Languages*, 1993.

[4] E. Jee, J. Yoo, S. D. Cha, and D. Bae, "A data flow-based structural testing technique for FBD programs." *Information & Software Technology*, vol. 51, no. 7, pp. 1131–1139, 2009.

[5] E. Jee, S. Kim, S. Cha, and I. Lee, "Automated test coverage measurement for reactor protection system software implemented in function block diagram," in *Proceedings of the 29th International Conference on Computer Safety, Reliability, and Security*, ser. SAFECOMP'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 223–236.

[6] E. M. Clarke, O. Grumberg, and D. Peled, *Model checking*. MIT Press, 2001.

[7] J. Lahtinen, J. Valkonen, K. Björkman, J. Frits, I. Niemelä, and K. Heljanko, "Model checking of safety-critical software in the nuclear engineering domain," *Reliability Engineering & System Safety*, vol. 105, no. 0, pp. 104 – 113, 2012.

[8] A. Biere, K. Heljanko, T. A. Junttila, T. Latvala, and V. Schuppan, "Linear encodings of bounded LTL model checking," *Logical Methods in Computer Science*, vol. 2, no. 5:5, pp. 1–64, 2006.

[9] FBK-IRST, Carnegie Mellon University, University of Genova and University of Trento, "NuSMV model checker v.2.5.4," 2012. [Online]. Available: http://nusmv.fbk.eu/

[10] J. Lahtinen, J. Ranta, and L. Lötjönen, "CORSICA 2013 work report: Test set generation, FPGA model checking, and fault injection," VTT Technical Research Centre of Finland, Espoo, Finland, Research report VTT-R-00212-14, 2014.

[11] J. Callahan, F. Schneider, S. Easterbrook *et al.*, "Automated software testing using model-checking," in *Proceedings 1996 SPIN workshop*, vol. 353. Citeseer, 1996.

[12] A. Engels, L. Feijs, and S. Mauw, "Test generation for intelligent networks using model checking," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, E. Brinksma, Ed. Springer Berlin Heidelberg, 1997, vol. 1217, pp. 384–398.

[13] G. Fraser, F. Wotawa, and P. E. Ammann, "Testing with model checkers: a survey," *Software Testing, Verification and Reliability*, vol. 19, no. 3, pp. 215–261, 2009. [Online]. Available: http://dx.doi.org/10.1002/stvr.402

[14] E. P. Enoiu, D. Sundmark, and P. Pettersson, "Model-based test suite generation for function block diagrams using the UPPAAL model checker," in *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on*. IEEE, 2013, pp. 158–167.

[15] P. Ammann, P. Black, and W. Majurski, "Using model checking to generate tests from specifications," in *Formal Engineering Methods, 1998. Proceedings. Second International Conference on*, Dec 1998, pp. 46–54.

[16] G. Hamon, L. de Moura, and J. Rushby, "Generating efficient test sets with a model checker," in *Software Engineering and Formal Methods, 2004. SEFM 2004. Proceedings of the Second International Conference on*, Sept 2004, pp. 261–270.

[17] T. Niermann, R. Roy, J. Patel, and J. Abraham, "Test compaction for sequential circuits," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 11, no. 2, pp. 260–267, Feb 1992.

[18] A. Gargantini and G. Fraser, "Generating minimal fault detecting test suites for general boolean specifications," *Information and Software Technology*, vol. 53, no. 11, pp. 1263 – 1273, 2011.

[19] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276–291, 2013.

[20] J. Campos, R. Abreu, G. Fraser, and M. d'Amorim, "Entropy-based test generation for improved fault localization," in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, Nov 2013, pp. 257–267.

[21] K. Björkman, J. Valkonen, K. Heljanko, and I. Niemelä, "Model-based analysis of a stepwise shutdown logic," VTT Technical Research Centre of Finland, VTT Working Papers 115, 2009.

[22] A. Pakonen, T. Mätäsniemi, J. Lahtinen, and T. Karhela, "A toolset for model checking of PLC software," in *IEEE 18th Conference on Emerging Technologies & Factory Automation (ETFA)*, September 2013, pp. 1–6.

Aalto-DD 159/2016
VTT SCIENCE 133

ISBN 978-952-60-6959-3 (printed)
ISBN 978-952-60-6958-6 (pdf)
ISSN-L 1799-4934
ISSN 1799-4934 (printed)
ISSN 1799-4942 (pdf)

978-951-38-8448-2 (printed)
978-951-38-8447-5 (pdf)
2242-119X
2242-119X (printed)
2242-1203 (pdf)

Aalto University
School of Science
Department of Computer Science
www.aalto.fi

BUSINESS + ECONOMY

ART + DESIGN + ARCHITECTURE

SCIENCE + TECHNOLOGY

CROSSOVER

DOCTORAL DISSERTATIONS