Symbolic Verification of Petri Net Based Models

PhD dissertation by

András Vörös

Advisor

Tamás Bartha, Ph.D. (BME)

András Vörös
http://mit.bme.hu/~vori/

2018

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Méréstechnika és Információs Rendszerek Tanszék

Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

H-1117 Budapest, Magyar tudósok körútja 2.

Declaration of own work and references

I, András Vörös, hereby declare, that this dissertation, and all results claimed therein are my own work, and rely solely on the references given. All segments taken word-by-word, or in the same meaning from others have been clearly marked as citations and included in the references.

Nyilatkozat önálló munkáról, hivatkozások átvételéről

Alulírott Vörös András kijelentem, hogy ezt a doktori értekezést magam készítettem és abban csak a megadott forrásokat használtam fel. Minden olyan részt, amelyet szó szerint, vagy azonos tartalomban, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Budapest, 2018. 06. 13.

Vörös András

Acknowledgements

I was really lucky to work together with amazing colleagues, students and friends, on various research and industrial projects.

First and foremost I would like to express my gratitude to my advisor, Tamás Bartha, you have been a tremendous mentor to me. I would like to thank you for encouraging my research and for allowing me to grow as a research scientist. Your advice on both research as well as on my career have been priceless. I would especially like to thank András Pataricza, István Majzik, Dániel Varró and Zoltán Micskei for their advice, suggestions and all their help in the last decade. Your help goes beyond this dissertation.

Special thanks all my colleagues and former colleagues at the research group, especially Dániel Darvas, Gábor Szárnyas, Vince Molnár, Tamás Tóth, Ákos Hajdu, Imre Kocsis, László Gönczy, Kristóf Marussy, Attila Klenik, János Oláh, Dávid Honfi, Oszkár Semeráth, Márton Búr, Ákos Horváth, István Ráth and Gábor Bergmann.

I would like also thank to my collaborators, with whom it was a great pleasure to work together. Some important results could not be born without their help: Attila Jámbor, Tamás Szabó and Zoltán Mártonka.

I would like to thank the financial support of the R3-COP, R5-COP research projects and also the support of the MTA-BME Lendület Cyber-Physical Systems Research Group.

Words cannot express how grateful I am to my whole family for all their love, support and help.

Köszönetnyilvánítás

Nagyon szerencsés voltam, hogy az elmúlt években fantasztikus kollégákkal, diákokkal és barátokkal dolgozhattam együtt.

Először és mindenekelőtt szeretném kifejezni hálámat konzulensemnek, Bartha Tamásnak. Szeretnék köszönetet mondani a támogatásért, amit az elmúlt években kaptam, és hogy bármilyen nehéz pillanatban mellettem álltál, számíthattam a segítségedre.

A kutatócsoportban csodálatos vezetőim voltak, szeretnék köszönetet mondani Pataricza Andrásnak, Majzik Istvánnak, Varró Dánielnek és Micskei Zoltánnak tanácsaikért, vezetésükért, tanításaikért, amelyek jóval túlmutatnak ezen a disszertáción.

Köszönöm szépen minden munkatársamnak (és volt munkatársamnak) a kutatócsoportban a segítséget! Különösen köszönöm a disszertáció írásban és az évek során nyújtott rengeteg támogatást Darvas Dánielnek, Szárnyas Gábornak, Molnár Vincének, Tóth Tamásnak, Hajdu Ákosnak. Emellett köszönöm Kocsis Imrének, Gönczy Lászlónak, Marussy Kristófnak, Klenik Attilának, Oláh Jánosnak, Honfi Dávidnak, Semeráth Oszkárnak, Búr Mártonnak, Horváth Ákosnak, Ráth Istvánnak és Bergmann Gábornak az elmúlt időben nyújtott szakmai és emberi segítséget. Nagyszerű volt együtt dolgozni Veletek! Köszönettel tartozom továbbá volt hallgatóimnak, akik nélkül több eredmény sem születhetett volna meg, köszönöm Nektek is a segítséget, élmény volt együtt dolgozni: Jámbor Attila, Szabó Tamás és Mártonka Zoltán.

Munkámat az R3-COP és az R5-COP kutatási projektek, valamint az MTA-BME Lendület Kiberfizikai rendszerek kutatócsoport támogatta.

Köszönöm szépen a családom támogatását. Édesapámnak, aki a példaképemként mindig motiválta a munkámat, édesanyámnak, aki még a legnehezebb pillanatokban is a szeretetével elhalmozott és támogatott, öcsémnek, nagymamámnak, hogy hittek bennem. Kedvesemnek, hogy mellettem állt, biztatott és támogatott. Nagyon sokat köszönhetek Nektek.

Summary

Ensuring the correctness of critical systems is a challenging task: various verification techniques are used to find errors or to prove correctness. Such verification methods are testing, monitoring (i. e., runtime-verification), model checking, static analysis and other design time verification techniques. Formal verification can be used even at design time to precisely analyse the models of the systems and evaluate their correctness. However, the complexity of real systems and the resulting computational demand often hinders successful verification. The problem arises especially often for asynchronous, concurrent and distributed systems, whose models face the state space explosion problem.

In my work, I aim to support the efficient modelling and verification of asynchronous systems with data dependent behaviour. I propose a framework that provides Petri net based modelling languages to support the development of the formal representation of the arising engineering problems. In the framework, ordinary Petri nets are used to capture the asynchronous behaviour of safety-critical systems and coloured Petri nets (CPN) provide a convenient means to express data-dependent behaviour. Furthermore, Petri nets can be used to efficiently represent and handle systems even with infinite state spaces. The proposed approach contains powerful verification algorithms: next-state computation and representation help efficiently handle variants of Petri net models. Saturation and abstraction based state space exploration techniques can handle complex models with huge state spaces. Various temporal logic specifications are supported: efficient CTL and LTL model checking algorithms are devised in the framework to support the specification and analysis of complex properties. Exploring the (partial) state space of infinite-state systems and counterexample generation is supported by bounded saturation based algorithms and Counterexample-Guided Abstraction Refinement. The unique combination of the above mentioned algorithms in a model checking approach yields the novelty of the framework.

The devised approach manifested in the PetriDotNet framework, which was used in various industrial and academic research projects.

The introduced approach needed many algorithmic contributions. In my thesis, I introduce the following new algorithms:

- I created an efficient algorithm for the verification of coloured Petri net models based on disjunctive-conjunctive decomposition. I also introduce a new temporal decomposition algorithm, which could further improve the performance of the saturation algorithm for complex Petri net based models.
- I developed a parallel saturation based state space exploration algorithm that extends the former approach with a new locking and synchronisation mechanism, yielding better resource utilisation compared to the former algorithm from the literature.
- I introduce a new model checking algorithm supporting the regular safety subset of the LTL language. The new algorithm is based on a novel synchronous product computation method which is significantly more efficient for asynchronous models than other approaches.

The algorithmic results became the building blocks of the new model checking approach envisioned in the PetriDotNet framework.

With the help of various case studies I illustrate the efficiency of my algorithmic developments. The CPN verification algorithm was the first which could fully verify a new safety logic of the Paks Nuclear Power Plant. Various benchmark models are used to evaluate the new algorithms and I compare their performance to those of existing model checking frameworks. The result of my work helped the PetriDotNet framework – which consists of modelling support and various verification algorithms – to become capable of solving industrial and research problems.

Összefoglalás

Kritikus rendszerek helyességének biztosítása nehéz feladat, e célból különböző megközelítések léteznek a hibák megtalálása, vagy a helyesség bizonyítása érdekében. Ilyen úgynevezett verifikációs megközelítések többek között a tesztelés, monitorozás (azaz futásidejű ellenőrzés), továbbá a modellellenőrzés, statikus analízis, stb. A formális verifikáció lehetőséget ad akár már a tervezés során, hogy precíz analízist hajtsunk végre és a tervek helyességét vizsgáljuk.

Azonban a valós rendszerek összetettsége, és az ebből fakadó számítási igény gyakran megakadályozza a sikeres verifikációt. Elosztott, aszinkron és konkurens rendszerek esetén ez a probléma különösen gyakran felmerül az úgynevezett állapottér robbanás jelensége miatt.

Munkám során adatfüggő viselkedésű aszinkron rendszerek hatékony modellezését és ellenőrzését céloztam meg. Ezen ellenőrzések támogatására egy megközelítést dolgoztam ki és egy keretrendszert javasolok, amely Petri net alapú modellezési nyelveket biztosít a felmerülő mérnöki problémák formális reprezentációjára. A keretrendszer az egyszerű Petri háló formalizmust biztosítja az aszinkron biztonság-kritikus rendszerek tervei, és a színezett Petri háló formalizmust (CPN) az adatfüggő viselkedés hatékony modellezésére. Emellett Petri hálók segítségével akár végtelen állapotterű rendszerek hatékony reprezentálása és kezelése is lehetséges.

Az általam javasolt megközelítés modellezési és verifikációs megoldásokat kombinál oly módon, hogy a keretrendszer alkalmas legyen komplex problémák megoldására. A különböző Petri háló variánsok támogatására az állapotátmenet reláció hatékony dekompozícióját lehetővé tevő megközelítéseket mutatok be, kihasználva az egyes modellezési formalizmusok jellemzőit. Dolgozatomban új algoritmusokat mutatok be a Petri háló alapú modellek szélesebb körének támogatására, azaz ellenőrzésére. Bonyolult modellek komplex viselkedéseit és ezen rendszerek állapotterét úgynevezett szaturációs algoritmusokkal kezeli a keretrendszer, amely emellett lehetőséget biztosít a követelmények hatékony megfogalmazására a CTL és az LTL specifikációs nyelvek segítségével is. A CTL és LTL nyelven megfogalmazott követelmények ellenőrzésére pedig hatékony ellenőrző algoritmusok kerültek kifejlesztésre. A (részleges) állapottér feltérképezésére, továbbá a végtelen állapotterű rendszerek kezelésére és a hatékony ellenpélda generálásra korlátos modellellenőrző algoritmusok állnak rendelkezésre, továbbá az ellenpélda-alapú absztrakció finomítás (Counterexample-Guided Abstraction Refinement, CEGAR) módszerét alkalmazom. Ezen algoritmusok innovatív kombinációja jelenti a keretrendszer egyik újdonságát.

A kutatásaim során kidolgozott megközelítés a PetriDotNet keretrendszerben került megvalósításra, és különböző ipari és tudományos kutatási esettanulmányok során pedig kiértékelésre. A bemutatott megközelítés hatékonyságához több algoritmikus fejlesztéssel járultam hozzá. Ezek az alábbiak:

- Hatékony algoritmust dolgoztam ki a színezett Petri háló modellek verifikálásához diszjunktív-konjunktív dekompozíció alkalmazásával. Bemutatok dolgozatomban egy új, lusta algoritmust a színezett Petri háló modellek komplex állapotátmeneti függvényeinek kezelésére. Az új algoritmusok szignifikánsan javították a színezett Petri háló alapú modellek verifikációs teljesítményét.
- Kidolgoztam egy párhuzamos szaturáció alapú állapottér felderítő algoritmust, amely a korábbi megközelítéseket egy új zárolási és szinkronizációs mechanizmussal bővíti. Az új algoritmus jobb erőforrás-kihasználtságot, és ezáltal gyorsabb futási időket eredményez, mint az irodalomból ismert korábbi algoritmus.
- Bemutatok egy új modellellenőrzési algoritmust, amely az LTL specifikációs nyelv reguláris részhalmazát támogatja. Az új algoritmus egy új szinkronszorzat számítási megközelítésen ala-

pul, amely az aszinkron modellekre jelentősen hatékonyabb, mint más megközelítések.

Az általam kidolgozott új algoritmusok a PetriDotNet keretrendszer modellellenőrzési megközelítésének építőelemei lettek.

Dolgozatomban különféle esettanulmányok segítségével bemutatom az új algoritmusok hatékonyságát. Az általam kidolgozott színezett Petri háló verifikációs algoritmus volt az első, amely a Paksi Atomerőmű egy bizonyos biztonsági logikájának a helyességét egyben bizonyítani tudta. Az új algoritmusok értékelését különböző benchmark-modellek segítségével is elvégeztem és összehasonlítottam már meglévő megközelítésekkel.

Dolgozatom eredményei hozzájárultak, hogy a PetriDotNet modellező és formális verifikációs keretrendszer mind ipari, mind kutatási problémák megoldására alkalmassá váljon.

Contents

Chapter 1

Introduction

1.1 Preliminaries and Objectives

Ensuring the correctness of systems is a long-standing requirement in the engineering disciplines. Engineers have been using various techniques to analyse their projects and find design problems before the implementation. Various means from the field of mathematics and physics helped to build more stable buildings, more robust and stronger machines and so on. The design workflow depicted on Figure 1.1, which has worked for many centuries in the engineering disciplines is also applied in the design of computer-based ICT (Information and Communication Technology) systems. However, the analysis of complex ICT systems requires new techniques and algorithms [BKL08] compared to the traditional engineering domains like the mechanical engineering domain or architecture.

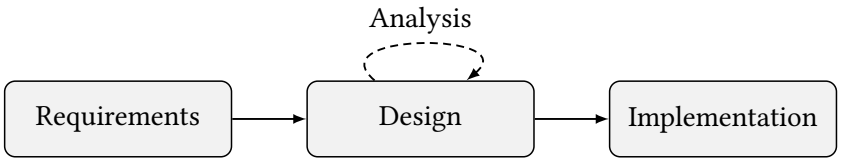


Figure 1.1: Development process

In my dissertation, I focus on the correctness analysis of critical ICT systems, and specifically the logical correctness checking, i.e. the verification of such systems. In my work I have investigated how the development can be supported by modelling languages, verification algorithms, and a framework, making all these techniques available to the computer engineers. The outcome of the verification process will help the engineers producing systems with better quality and fewer errors.

1.1.1 Target of the Dissertation

A system is safety-critical if its failure could result in loss of life or significant damage. There are many well-known safety-critical areas such as medical devices, aircraft flight control and nuclear systems. Ensuring the correctness of these systems is especially important, in which advanced verification techniques play a significant role.

Safety-critical systems are inherently distributed, components responsible for various functions in these systems cooperate to keep up the proper operation. The distributed characteristics of the components and their interaction results in intricate system level behaviour. This fact raises the main challenge: the resulting behaviour is not only difficult to understand and to modify, but also to analyse.

Due to the technological development, recent safety-critical systems are becoming more and more complex, raising challenges in the modelling, development and also in the verification. In my thesis, I aim to provide solutions to support the modelling and verification of safety-critical systems. As no single approach can cover all aspects of ICT systems, in my work I focus on the verification of asynchronous systems, such as communication and distributed systems.

Verification analyses if the model of the system fulfils the given correctness criteria. Various kinds of requirements [BKL08] are expected to be fulfilled by the system:

- Safety requirements express that the system does not reach an error/dangerous state.
- Boundedness properties express restrictions to the resource usages and other aspects of the system.
- Liveness properties expect the system to respond to a request after finite delay and also avoid deadlock, for example, a client will finally send an answer to a request.
- Reversible systems can reach certain states again and again.
- Persistence requirements express that some property will finally hold in the system after a transient phase. For example, in a distributed system, the connection will be established, and it remains in that state stably.

In my dissertation, I aim to support a rich set of properties to specify the correctness requirements.

1.1.2 Verification Techniques in Systems Engineering

In this section, a typical engineering workflow based on the widely known V-model is used to show the role of verification techniques at the various phases of development (the V-model received its name as it forms a V shape). This workflow is used as a general guideline in the development of safety-critical systems. Many variants of this workflow were developed by the industry tailored to the special needs of the different sectors. The V-model defines the elementary steps and draws a general workflow for the design and implementation of the system as depicted on Figure 1.2. In addition, the workflow defines verification and validation steps in the development to ensure that the correct design is developed and it fulfils the requirements. In the following, the various verification and validation techniques are summarised that ensure the correctness at the different phases of the development. The V-model defines the verification goals to ensure correctness of:

- the design with regard to the requirements and
- the implementation with regard to the design.

The systems engineering process depicted on Figure 1.2 starts by designing the requirements, which are then refined in the next, so-called system design phase. This phase defines the main functionalities of the system. In the next phase, designing the architecture provides the necessary decomposition to be able to construct the component level design. At each step, the designer refines the outcome of the former steps by providing more details. At the final step of the left wing of the V-model, one can produce the implementation for each component from the design models. Implementation has to be tested and verified against coding and other implementation errors. After the component level validation, system integration builds the smaller pieces together where extensive integration testing is executed to validate that the components work properly together. Finally, system validation ensures if the outcome is the system, which is desired by the customer.

Various analysis techniques serve the verification and validation of the system design and implementation. In the following these analysis techniques, such as (computer) simulation, testing and formal verification are shortly summarised:

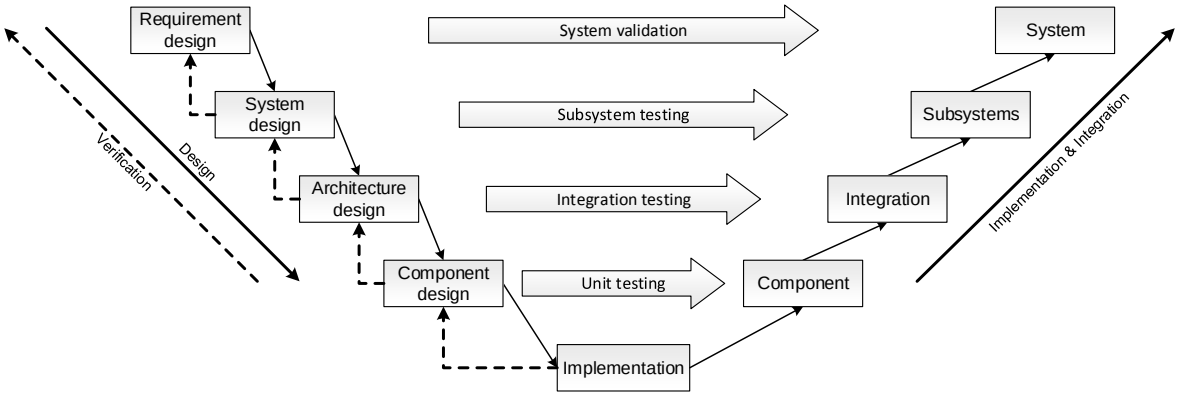


Figure 1.2: Verification during system engineering

Simulation is the process of executing the model of a system. Simulation is a design-time activity to assess the dynamic aspects of systems.

Testing is an activity in which a system or component is executed under specified conditions, the results are observed or recorded, and evaluated according to the specification [Ins10; Mic13]. Systems are tested at various phases of the development [Ana+13] from component level implementation [GA14] up to system level integration.

Formal verification is the procedure of proving or disproving the correctness of a system with respect to a certain formal specification or property [CES86]. Formal verification is based on the mathematics of computation. Both design models and also implementation [Bey17] can be verified. However, as the models are becoming more detailed and the design approaches the implementation level, the computational complexity increases.

Simulation is the elementary task of inspecting and analysing the system behaviour. The prerequisite is the model of a real or imagined system, which shall be designed and then experiments are conducted on the model during simulation to reproduce the possible behaviour. The purpose of simulation experiments is to understand the behaviour of the system or evaluate strategies for the operation of the system. Simulation uses an abstract model (a computational model) for execution. Simulation provides analysis capabilities at an early stage of the development when only models are available.

The testing procedure can be carried out at various levels of abstraction. In one hand, testing the model by simulating it and evaluating the behaviour can provide feedback for the developers at an early stage of the design. Simulation is used as the elementary procedure to test models. On the other hand, testing the implementation provides inputs and observes the reactions of the concrete system. Testing is one of the most widely used verification approaches [MSB11].

In general, testing analyses the runs of the system by providing inputs, simulating the behaviour and examining the reaction (output), by comparing it to an explicitly stated (provided as assertions) or implicitly assumed (such as no crash should occur) expected behaviour. Testing is efficient in finding problems, and it has many advantages. Testing the model of the system relies on simulation, which can be computed efficiently. Besides, testing is easy-to-use for the developers: no additional knowledge is required, it works on the model of the system. Testing can also be applied at the implementation level so the revealed problems do not come from the inaccurate modelling but they are real problems in the implementation. In addition, when exhaustive verification is not possible, testing can still help locating problems.

On the other side, neither simulation nor testing can be complete in the sense that they usually can not explore all the behaviour of a system so neither simulation nor testing can prove correctness alone.

Formal verification extends their strengths with mathematically established proofs based on the exhaustive traversal of all the possible behaviour.

Finding errors is one side of the problem: the need to be able to prove correctness naturally raised. This need led to the development of formal verification techniques to support the engineers with tools providing certainty about the correctness of their design.

The combination of the various verification methods can ensure the high quality of computer-based systems. These techniques can be used in different phases of the systems engineering process, and they together constitute a powerful tool to find errors at an early stage of the development.

1.2 Formal Verification

Formal verification is the analysis of hardware, software and systems that provides mathematically established proofs for correctness or existing errors. Formal verification is performed on the abstract representation (model) of the system or directly on the source code.

Nowadays, the application of formal methods is gaining high importance in the development of modern ICT and especially safety-critical systems. Standards, like IEC 61508 also recommend the application of formal techniques in the development process.

1.2.1 Applying Formal Verification in System Design

The traditional application of verification is depicted on Figure 1.3 [13]. The goal is to verify the correctness of the system by checking if the engineering model fulfils the requirements [LMM99; Cse+02]. Applying formal verification in the system development process consists of the following steps.

1. Engineering models and requirements are developed.
2. Formal models and formal requirements are created.
3. The verification procedure is executed.
4. Results are interpreted and back-propagated to the engineering levels. Corrections are made, and verification is run again if needed.
5. Implementation is derived from the engineering models.

The input of the procedure is the engineering model, which is usually described by a domain expert in the language of design tools, for example, SysML or UML. The requirements also come from the engineers. Both the engineering models and the requirements have to be formalized and transformed into the input language of some verification tools. Formal models provide a mathematically precise way to describe the system: this enables the application of formal verification techniques on the systems' design. The result of the verification is interpreted on the level of the formal model and formal specification: this result has to be back-propagated to the engineering domain to be understood by the engineers. This back-propagation procedure is depicted on the figure with dashed lines pointing towards the engineering level.

Supporting the whole verification process involves all the aforementioned steps. In this thesis, I focus on how to support the formal modelling, and I will introduce novel verification algorithms to enhance the verification process.

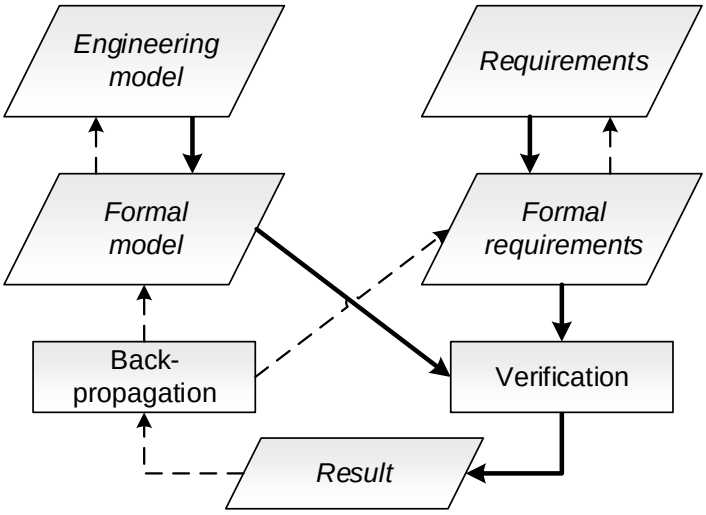


Figure 1.3: High-level view of the verification process

1.2.2 Formal Modelling

Formal modelling is the process of developing a formal representation of the system under analysis in a formal modelling language. The resulting model can be analysed by various techniques to prove its correctness or find design errors.

1.2.2.1 Development of the Formal Representation

There are two main directions to develop the formal models: they can be developed manually, or formal models are automatically generated from the engineering models by using model transformations. Verification engineers develop formal models from the system description. Increasing the expressiveness of the formal modelling language supports the efficient development of the formal models. It means a smaller abstraction gap between the engineering and formal modelling level and also provides more information for the underlying verification engines.

Many approaches try to support the automatic generation of formal models from engineering models, but they rarely provide a proper solution for the problem in their own as the generated formal models might contain too many details preventing successful verification [Dar17],[13],[12].

1.2.2.2 Formal Modelling Languages

There are many formalisms to represent the system under analysis. As systems possess various characteristics, formal representations have to be able to express these properties and exploit them for verification.

Finite state automaton and their extensions are popular as they are easy to use and an automaton can naturally represent certain problem domains. Additionally, networks of automata provide a compact representation for distributed systems. Programming language-like formalisms are popular for their expressiveness and as they are similar to those languages that software engineers are used to. Petri nets and related models constitute an expressive class of formal modelling languages: they are popular for their simplicity, but Petri nets still possess high expressive power. There are two main types of Petri nets:

Petri net based modelling languages solve the problem of graphical and formal representation of concurrent and distributed systems. Petri nets naturally handle the inherent asynchronism of such systems. Petri nets can represent both finite and infinite state systems. Moreover, various subclasses and extensions exist to support the modelling and analysis of concurrent systems.

Coloured Petri nets (CPN) extend Petri nets with various data types and variables, and additional guard expressions are used to refine the possible behaviour of the systems further. Coloured Petri nets can raise the abstraction level to help the efficient development of formal models.

Ordinary Petri nets are well-suited to model control flow and data dependent behaviour. Petri net based models can have finite state space which means that a finite number of states are reachable from the initial state. Various subclasses exist for representing the various problems. Finite state machines and networks of finite state machines are expressed with finite Petri nets. The marked graph is a special subclass of Petri nets being able to represent decision-free parallel activities. For these Petri net subclasses, efficient verification methods exist [Mur89; BKP17].

On the other hand, Petri net based models can also represent infinite state systems, with an infinite number of reachable states. Such ordinary Petri nets are used to model concurrent multi-threaded programs with finite data structures. In general, the expressive power of Petri nets equals the expressive power of Vector Addition Systems [EN94].

In coloured Petri nets, various data types can be used as colour types, and guard expressions can be evaluated on them. Coloured Petri nets combine the inherent concurrency of Petri nets with data dependent behaviours expressed with colour types and other language elements. Coloured Petri nets are also a proper means to describe parametric systems. Compared to ordinary Petri nets where the structure of the net encodes the modelled behaviour, in coloured Petri nets, a wider range of language elements support the modelling.

1.2.3 Formal Requirements

Formal requirements capture the requirements of the design phase in a formally interpreted precise language. In this thesis, our goal is to verify the behaviour of systems, so we restrict the introduction to the formal requirement languages being able to express behavioural properties.

Formal requirements are usually expressed with the help of temporal logic. Various temporal logics exist, the two most common are Computation Tree Logic (CTL) and Linear Temporal Logic (LTL). They have different semantics and expressive power. For example, deadlock freedom can be only expressed with CTL while fairness properties are only expressible with LTL. It is desirable for a model checker to support both formalisms. However, only a few model checkers provide support for both of them.

1.2.4 Formal Verification Techniques

There are various formal verification techniques to ensure the correctness of systems [DKW08]. Formal verification does not rely on the concrete execution of the software/system, so these techniques are often referred to as static analysis techniques. Widely used static analysis techniques are – among others – abstract interpretation, model checking and theorem proving.

Abstract interpretation techniques derive properties from the structure of the models or the source code. Abstract interpretation iterates through the program and approximates the possible behaviours without executing the calculations of the program. Over-approximation and various abstractions tailored to the domain provide efficiency. However, accurate results can not be provided due to the coarse approximations.

Model checking analyses a model representation of the system. Model checking algorithms exhaustively explore the possible states of the system and verify the requirements given as temporal logic formulae. Model checking algorithms produce a counterexample if a property violation is found.

Theorem proving based verification reduces the verification problem to solving first-order or higher-order logic problems. First-order theorem provers might work fully automatically, higher-order logic provers are mainly interactive. Both the property and the system representation have to be expressed as a logic problem in the input language of the chosen theorem prover.

From the various approaches, there are semi-automatic procedures like interactive theorem proving, and on the other side, there are fully automatic techniques such as model checking and abstract interpretation. There is a huge gap also in precision: abstract interpretation examines safety properties being reduced to the reachability checking of some erroneous states. Model checking extends the set of analysis questions, and it is able to answer liveness or even complex fairness and timed properties. Theorem proving can prove even properties expressed in higher-order logic, however only for certain (very restricted subset of) systems. In general, static analysis techniques (like abstract interpretation) are known to be computationally cheaper but less precise, on the other side model checking is more precise for the more computational cost.

The approaches use different formalisms to design the system representation: the verification engineers can choose between directly verifying the implementation or execute the analysis on a higher level of abstraction. The first approach provides information directly from the implementation: this advantage is however very expensive as the verification of programs is algorithmically rarely tractable. Abstract models are usually easier – however in practice still difficult – to verify, but implementation and coding problems are not detected at this higher level.

Formal verification is getting more and more widely used for industrial problems [Cal+15; Adi+15; DMB16; LS09; Kle+09; SD10; BP12; Kai+09]. From the various techniques, model checking provides a good trade-off between precision, expressiveness and computational costs [DKW08].

1.2.4.1 Model Checking

Model checking is an automated formal verification technique: given a formal model representing the system, and a formal specification, a model checking algorithm traverses the possible behaviours of the formal model and decides if the formal specification is fulfilled. The state space is represented in the internal data structures of the model checker and used for the analysis of the formal properties. When the formal specification is fulfilled, the correctness of the design is proven. Otherwise, the model checker produces a counterexample, which shows how the system can reach an incorrect/undesired situation.

There are two main families of model checking algorithms: *explicit techniques* use traditional graph algorithms to explore the states one-by-one. On the other side, *symbolic model checking* algorithms apply special encoding of the state space and the transition relation. Explicit state model checking can be fast but often faces the so-called state space explosion problem: even small models can have huge state spaces, which can not fit into the memory of modern computers. Symbolic algorithms try to solve the state space explosion problem by avoiding the explicit representation of the state graph and using a compact representation instead. One of the symbolic approaches is saturation, which was devised for the verification of asynchronous, concurrent systems.

Model checking is a difficult problem in general: even small systems can have huge state space due to the large number of interactions of asynchronous, concurrent or distributed systems, or caused

by the data content of the state variables. In many cases, formal models have infinite state spaces, which have to be traversed and represented by the model checking algorithms. Explicit model checking algorithms store the states and transitions one-by-one and traditional graph traversal algorithms are used: the memory requirements of storing huge state space graphs often prevent their application. On the contrary, symbolic model checking algorithms handle sets of states together instead of manipulating them individually and clever encodings help to fit the state space representation into the memory.

Symbolic state space representation. In symbolic model checking, characteristic functions are used to encode sets of states and decision diagrams can be used for efficient storage. A decision diagram is a directed acyclic graph, representing a Boolean or multi-valued function. Various reduction rules ensure that decision diagrams are a canonical and compact representation of a given function or set, which makes it a proper means to store set of states. Traditional symbolic algorithms encode the reachable states and also the next state functions in decision diagrams. State variables are mapped to the variables of the decision diagram and state vectors are stored in the decision diagram.

The other option is the application of *SAT-based* techniques to manipulate the characteristic function of the symbolic representation and efficient solvers help the state space traversal. Induction can help to find proofs for correctness or bounded state space exploration searches counterexamples.

Symbolic model checking algorithms can manipulate a huge set of states together, but their efficiency highly relies on the used encoding. Finding a good encoding can be a complex task.

Efficient state space traversal. In model checking, the state space has to be traversed. During the exploration, states have to be stored or memorised, to avoid redundant exploration, redundant computations. Decision diagrams offer a compact representation and storage for the state space, but the construction of the state space representation i. e., the exploration strategy of the state space has to be chosen. The states of synchronous hardware systems are traditionally explored by breadth-first (BFS) traversal in symbolic model checking. However, BFS is used to be inefficient for asynchronous systems [CMS05]. On the other hand, depth-first (DFS) traversal does not fit the traditional decision diagram based symbolic algorithms as symbolic algorithms are not able to handle states individually.

Ciardo and his colleagues developed a special iteration strategy to solve this problem, the so-called saturation iteration algorithm, which combines BFS and DFS strategies tailored to the structure of the decision diagram representation of the state space. Saturation is efficient[CMS05] for asynchronous and GALS (Globally Asynchronous Locally Synchronous) systems. Saturation was developed for Petri nets, and some extensions also support model checking of various properties.

Checking temporal logic specifications. CTL and LTL are widely used temporal logics with different expressiveness and different verification algorithms. CTL model checking is reduced to compute greatest and least fixed points of the next-state functions in the state space iteratively. This approach is called structural model checking and decision diagram based symbolic approaches efficiently solve the problem of traversing and storing the possible states. On the other side, LTL model checking requires different algorithms as it is reduced to the checking of language inclusion of the property automaton and the state space of the model. Model checking LTL properties is usually composed of two challenges: one must compute the synchronous product of the state space and the automaton model of the desired property, then look for counterexamples that is reduced to finding strongly connected components (SCCs) in the state space of the product. Checking LTL properties is computationally a harder problem than checking CTL properties in general.

1.2.5 Target Problem of the Dissertation

In this dissertation, I focus on the modelling and verification of concurrent, asynchronous systems, which constitute a significant part of the set of safety-critical systems. These are typically discrete-event systems (DES), so I show how such systems can be efficiently modelled and verified.

Summarising the verification challenges in general: there is a need to precisely (formally) represent the system and the requirements, efficient model checking algorithms are required to solve the verification problem, and we need tool support with the aforementioned capabilities.

1.3 Overview

In this section, I overview the challenges in the model checking process. In [BKL08], authors defined the following phases in the application of model checking in systems engineering:

- Modelling phase:
  - Model the system using the description language of the model checker.
  - Perform sanity checks by simulation
  - Formalize the requirements using a property specification language.
- Running phase: run the model checker and check the validity of the property in the model.
- Analysis phase:
  - If the property is satisfied, check the next property.
  - If the property is violated, then:
    1. Analyse the generated counterexample;
    2. Refine the model, design or the property;
    3. Repeat the entire procedure.
  - Out of memory or time-out necessitates the reduction of the model or the application of a different model checking algorithm.

Formal verification is often desirable though complex task and each phase of the model checking process has their own challenges. Developing formal models is time-consuming, and the verification of real industrial problems is computationally hard. The huge gap between engineering and formal models are difficult to bridge by automated techniques. On the other hand, verification engineers might develop the proper formal models from engineering models. However, this process is time-consuming. Even if the models are available in a formal representation, the requirements have to be also expressed formally, which needs a rich set of formal requirement languages. After all, the high computational cost of formal verification often prevents its successful application.

1.3.1 Model Checking of Asynchronous Systems

Formal verification is a computationally difficult problem: small systems still have huge state spaces, which has to be traversed by the verification algorithms. This is especially true for the asynchronous models of distributed systems: the various overlapping of the components' behaviour yields a huge number of possible behaviour. Advanced techniques are required to handle this explosion. The number of possible states grows exponentially with the growing number of components in a distributed system, even up to the Cartesian product of the states of the individual components. As formal verification has to be exhaustive, the large number of states poses huge challenges for the verification algorithms.

As it can be seen, formal verification of distributed systems is computationally expensive so choosing the proper approach is crucial. Saturation provides an efficient solution for the model checking of Petri net models: my work is based on saturation-based techniques.

Due to the computationally extensive nature of model checking, the question naturally arises that how modern multi-core processor could be exploited for further enhancing the performance of model checking. Efficient model checking approaches such as symbolic algorithms use complex data structures and iteration strategies making the parallelisation task difficult. The reason for symbolic model checking being inherently sequential is that fixed-point computation and detection needs the results of the previous steps. This problem is especially true for saturation, which was also discussed in the literature [CZJ09]. With the newer and newer advances of model checking, the challenge of exploiting multi-core computers in saturation-based model checking is raised.

1.3.2 Formal Modelling

Petri nets are a popular modelling language to describe the behaviour of concurrent and asynchronous systems, but many application domains require a more expressive formalism: in such cases, coloured Petri nets provide efficient means to describe asynchronous systems with data dependent behaviours.

Coloured Petri nets (CPN) extend ordinary Petri nets with various data types that can be used as colour types, and guard expressions can be evaluated on them. However, this also yields challenges for the verification algorithms. These intricate language elements of coloured Petri nets are difficult to be handled by the model checking algorithms. This issue should be addressed in a formal verification tool analysing coloured Petri net models.

Many efficient techniques and tools exist for the verification of Petri net models. The drawback of the application of CPNs is the lack of efficient verification techniques, what we also faced in our research. The reason for that is twofold: by choosing decision diagram based techniques, one can efficiently represent the state space of Petri net models, but complex guard expressions are not efficient to be encoded in decision diagrams. On the other side, techniques based on advanced solver technologies such as SAT and SMT being able to handle intricate guard expressions efficiently are not good at verifying concurrent systems. These facts lead to the situation that coloured Petri net based symbolic verification tools are not available. The most commonly used tool for verifying CPN models is CPNTools [JKW07], which provides techniques based on explicit state traversal and representation. Such explicit techniques rarely scale to real-life problems.

1.3.3 Formal Requirements

From the wide range of requirement languages, formal verification relies on formal languages such as CTL or LTL: these are the most widely used formal specification languages being able to express many kinds of properties of interest. CTL model checking is reduced to compute greatest and least fixed points of the next state functions in the state space iteratively. This approach is called structural model checking and decision diagram based symbolic approaches efficiently solve the problem of traversing and storing the possible states. On the other side, LTL model checking is reduced to solving language containment problem. Model checking LTL properties is usually composed of two challenges: one must compute the synchronous product of the state space and the automaton model of the desired property, then look for counterexamples that is reduced to finding strongly connected components (SCCs) in the state space of the product. Symbolic model checking approaches exist for the SCC computation. However, the efficient construction of the synchronous product is still an open question, especially when using saturation-based algorithms. Related work in this field uses tradi-

tional binary decision diagram encoding of the composite state space and encodes the synchronous next state relation in a big, monolithic decision diagram. Saturation-based approaches avoid the explicit computation of the synchronous product [Thi15] by dividing the iteration order into smaller parts. This approach might break the iteration strategy of saturation, which may decrease the efficiency. Synchronous product computation is not yet integrated into saturation-based traversal for LTL model checking in a way that it would fully exploit the efficiency of the iteration strategy.

1.3.4 Objectives

My goal is to introduce a model checking approach for the verification of Petri net based models of complex systems. The proposed approach supports the formal modelling by providing coloured Petri nets as a high-level formalism to represent the system. Temporal logics such as CTL and LTL supports the development of formal specifications. Saturation is used to explore the state space and the next-state function and stores them symbolically. Temporal logic model checking processes this representation of the possible behaviours and evaluates the CTL or LTL specification. The result of the procedure is an error trace to show the problem in the system. Otherwise, the model of the system is assumed to be correct. In the following section, I summarise the challenges concerning the individual steps of the verification process.

1.3.4.1 Summarizing the Challenges

Challenge 1: Verification of complex systems High-level modelling languages are needed to model complex systems. High-level models of complex systems require rigorous verification techniques, so the existing verification approaches and algorithms have to be extended to overcome the challenges.

Saturation was introduced for the analysis of Petri-nets and their variants/simple extensions. However, in practice, higher level languages provide a better means to describe complex, real-life systems. Coloured Petri nets are a popular formalism, but saturation-based algorithms have not yet been extended for their analysis. Complex data structures of Coloured Petri nets have prevented the application of efficient saturation-based symbolic model checking algorithms in this field.

Challenge 2: Increase the efficiency of model checking algorithms New techniques are needed to increase the efficiency of model checking algorithms and decrease runtime requirements.

Parallelization is a common approach to improve the performance of algorithms. However, saturation is inherently sequential, so it is difficult to parallelise [CZJ09]. The reason behind is the fact that saturation heavily relies on the results of former computations. Indeed, the parallel manipulation of a decision diagram is a difficult problem on its own, which is the prerequisite to develop parallel saturation-based algorithms. Exploit the computational power of modern multi-core computers in saturation-based algorithms is a huge challenge.

Challenge 3. Verification support for various requirements Research and industrial case-studies revealed the need for a wide range of specification languages to support the various types of requirements of the use-cases.

CTL and LTL temporal logics have different strength and weaknesses, so it is important for a model checker to support both formalisms from the usability point of view. Saturation was traditionally used for CTL model checking as the traditional approaches for LTL model checking are difficult to implement in symbolic settings. LTL model checking is reduced to automata based model checking,

and saturation-based model checking approaches have to be extended to support automata based formal specifications. Automaton based specification provide the semantics for high-level specification languages such as LTL. LTL model checking requires solving an additional problem during the state space generation: namely the synchronous product computation with an automata representation of the property under analysis. This problem is easy to solve by explicit state space traversal algorithms, but intricate synchronisation constraints often prevent the efficient application of symbolic methods. This lead to that former saturation-based synchronous product generation approaches do not compute synchronisation constraints symbolically instead they try to divide the problem into locally solvable parts. However, this breaks the iteration might degrade it to a breadth-first like iteration.

Challenge 4: Tool support for formal modelling and verification. The wide range of industrial problems necessitates a formal modelling and verification framework with various modelling languages and verification algorithms. As no single formalism or algorithm can support the many aspects of the use-cases, a configurable framework is needed, which can be fine-tuned to handle the verification problems.

Therefore the goal of the dissertation was to define a framework addressing these challenges and develop the necessary algorithms for supporting the verification procedure.

Beside the algorithmic developments, there is a need for tool support for the envisioned verification framework. This involves modelling, verification and counterexample generation of complex systems. There has not been any tool yet for combining the aforementioned algorithms together in one tool to the efficient support of verification of various Petri net based models.

Chapter 2

Background

In this chapter, I introduce the theoretical background of my work. The basic definitions related to Petri nets are from [4][16]. The introduction of decision diagrams is from [6]. The overview for model checking and saturation is based on [5] and [20].

2.1 Petri Nets

Petri nets are graphical models for concurrent and asynchronous systems, enabling both structural and dynamical analysis. Formally, a Petri net [Mur89] is a tuple $PN = (P, T, E, W, m_0)$ where:

- P is the set of *places*,
- T is the set of *transitions*, with $P \neq \emptyset \neq T$ and $P \cap T = \emptyset$,
- $E \subseteq (P \times T) \cup (T \times P)$ is the set of *arcs* and
- $W : E \to \mathbb{Z}^+$ is the weight function assigning weights $w^-(p_j, t_i)$ to the edge $(p_j, t_i) \in E$ and $w^+(p_j, t_i)$ to the edge $(t_i, p_j) \in E$.

A *marking* of a Petri net is a mapping $m : P \to \mathbb{Z}_0^+$. The initial marking is denoted by m_0. A place p contains k tokens under a marking m if $m(p) = k$.

A transition $t_i \in T$ is *enabled* in a marking m, if $m(p_j) \geq w^-(p_j, t_i)$ holds for each $p_j \in P$ with $(p_j, t_i) \in E$. An enabled transition t_i can *fire*, consuming $w^-(p_j, t_i)$ tokens from places $p_j \in P$ with $(p_j, t_i) \in E$ and producing $w^+(p_j, t_i)$ tokens in places $p_j \in P$ with $(t_i, p_j) \in E$. The firing of a transition t_i in a marking m is called an *event* and denoted by $m[t_i\rangle m'$ where m' is the marking after firing t_i.

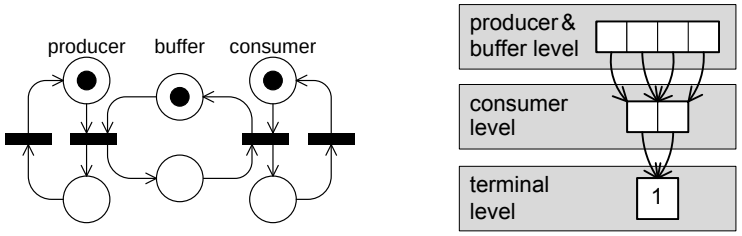
A word $\sigma \in T^*$ is a *firing sequence*. A firing sequence is *realizable* in a marking m and leads to m', (denoted by $m[\sigma\rangle m'$), if either $m = m'$ and σ is an empty word, or there exists a realizable firing sequence $w \in T^*$, a transition $t_i \in T$, and a marking m'' such that $m[w\rangle m''[t_i\rangle m'$ and $\sigma = \{w, t_i\}$. The *Parikh image* of a firing sequence σ is a vector $\wp(\sigma) : T \to \mathbb{Z}_0^+$, where $\wp(\sigma)(t_i)$ is the number of the occurrences of t_i in σ.

Reachability problem. A marking m' is *reachable* from m if there exists a realizable firing sequence $\sigma \in T^*$, for which $m[\sigma\rangle m'$ holds. The set of all reachable markings from the initial marking m_0 of a Petri net PN is denoted by $R(PN, m_0)$. The aim of the *reachability problem* is to check if $m' \in R(PN, m_0)$ holds for a given marking m'.

Define a *predicate* as a linear inequality on markings of the form $Am \geq b$, where A is a matrix, and b is a vector of coefficients [EM00]. The aim of the *submarking coverability problem* is to find a reachable marking $m' \in R(PN, m_0)$, for which the given predicate $Am' \geq b$ holds.

The reachability problem is decidable [May81], but it is at least EXPSPACE-hard [Lip76].

Reachability graph. The *state space* of a Petri net is the set of states reachable from the initial state through firings of transitions. The state space can be either finite or infinite. The reachability graph is constructed by traversing the states and connecting them by edges representing the transition firings, i. e., the steps in the state space. Figure 2.1a depicts a simple example Petri net model of a producer-consumer system. The producer creates items and places them in the buffer, from where the consumer consumes them. For synchronising purposes the capacity of the buffer is one, so the producer has to wait till the consumer takes the item from the buffer. This Petri net model has a finite state space of 8 states.



(a) The Petri net of the producer-consumer model

(b) State space representation with MDD

Figure 2.1: Producer-consumer example

2.2 Decision Diagrams

Decision diagrams are widely used for verification. In this section, a basic definition is given.

A *Multiple-valued Decision Diagram* (MDD) is a directed acyclic graph, representing a function f consisting of K variables: $f : \{0, 1, \ldots\}^K \rightarrow \{0, 1\}$. An MDD has a node-set containing two types of nodes: non-terminal nodes and two terminal nodes (0 and 1). The nodes are ordered into $K + 1$ levels. A non-terminal node is labelled by a variable index $0 < k \leq K$ that indicates to which level the node belongs (which variable it represents), and has n_k (domain size of the variable, in binary case $n_k = 2$) arcs pointing to nodes in level $k - 1$. A terminal node is labelled by the variable index 0. Duplicate nodes are not allowed, so if two nodes have identical successors in level k, they are also identical. In a quasi-reduced MDD redundant nodes are allowed: it is possible that every arc of a node points to the same successor.

These rules ensure that MDDs are canonical and compact representations of a given function or set. The evaluation of the function is based on a top-down traversal of the MDD through the variable assignments represented by the arcs between nodes. Figure 2.1b depicts an MDD used for storing the encoded state space of the example Petri net. Each edge encodes a possible local state, and the possible state configurations are the paths from the root node to the terminal node labelled *one*.

2.3 Saturation

Saturation [CZJ12] is a state space generation and model checking algorithm that proved its efficiency in the verification of asynchronous systems [CMS03]. This section overviews the basic concepts of the saturation algorithm.

2.3.1 Overview of Saturation

Saturation [CMS06] is a *symbolic algorithm* for state space generation and model checking that is particularly efficient for concurrent, asynchronous systems. Saturation explores the possible states of the model and stores the encoded state space in an MDD. *Decomposition* serves as the prerequisite for the symbolic encoding in saturation: the algorithm maps the state variables of the high-level model into symbolic variables of the decision diagram. Formally, saturation explores the reachable state space $\mathcal{S}_{rch}$ of a model $M = \langle \mathcal{S}, \mathcal{S}_{init}, \mathcal{E}, \mathcal{N} \rangle$ composed of K components (or subsystems), where:

- $\mathcal{S}$ is the possible set of *global* states. A *state variable* is defined for each component denoted by $s_1, \ldots, s_K$ with possible *local* state spaces $\mathcal{S}_1, \ldots, \mathcal{S}_K$, so that the global state space can be defined as their Cartesian product: $\mathcal{S} = \mathcal{S}_1 \times \cdots \times \mathcal{S}_K$. Each global state $\mathbf{s}$ is a K-tuple $\langle s_1, \ldots, s_K \rangle$, where each $s_k \in \mathcal{S}_k = \{0, 1, \ldots\}$ is the state of the k-th component ($1 \leq k \leq K$). The variables are mapped into symbolic variables of the encoding decision diagrams;
- $\mathcal{S}_{init} \subseteq \mathcal{S}$ is the set of initial states, $\mathcal{S}_{rch} \subseteq \mathcal{S}$ represents the set of states reachable from the initial states;
- $\mathcal{E}$ is the set of (asynchronous) events, usually transitions of a high-level model i. e., firing of a transition in a Petri net;
- $\mathcal{N} \subseteq \mathcal{S} \times \mathcal{S}$ is the next state relation defined as the union of the separate next state relations of the events as follows: $\mathcal{N} = \bigcup_{\varepsilon \in \mathcal{E}} \mathcal{N}_\varepsilon$, where $\mathcal{N}_\varepsilon$ is the next state relation of event ε. We often treat $\mathcal{N}$ as a function, defining $\mathcal{N}(\mathbf{s}) = \{\mathbf{s}' | \langle \mathbf{s}, \mathbf{s}' \rangle \in \mathcal{N}\}$ as the states that are reachable from $\mathbf{s}$ in one step (and also $\mathcal{N}(\mathcal{S})$ as an extension to sets of states).

The global state space $\mathcal{S}$ is represented as an MDD with K variables (levels), where variable x_i corresponds to the state of the ith component. A global state $\mathbf{s}$ is encoded by a trace (path) of the MDD, where $x_1 = s_1, \ldots, x_K = s_K$. Decomposition helps the algorithm to exploit the inherent *locality efficiently* of asynchronous systems.

Saturation uses a peculiar *iteration strategy*: it iterates through the MDD nodes and generates the whole state space representation using a node-to-node transitive closure. Building the MDD representation of the state space starts by building the MDD representing the initial state. Then the algorithm saturates every node in a bottom-up manner, by applying saturation recursively when new states are discovered. The result is the state space representation encoded in MDD. This way, saturation avoids that the peak size of the MDD during the iteration exceeds its final size, which is a critical problem in traditional approaches. The reader is referred for details and a running example to [CMS03].

Saturation exploits the locality inherent in concurrent systems, where a single event usually affects only a small number of components (state variables). This approach partitions the global next-state function according to the high-level model events in the system. An event ε is *independent* from the component k, if *1)* its firing does not change the state of the component, and *2)* its enabling does not depend on the state of the component. If ε depends on component k, then we call it a supporting variable: $k \in supp(\varepsilon)$. We define $Top(\varepsilon)$ as a function that returns the largest index in $supp(\varepsilon)$. Then $\mathcal{E}_k$ is the set of events: $\{\varepsilon \in \mathcal{E} | \, Top(\varepsilon) = k\}$. For the sake of convenience we use $\mathcal{N}_k$ to represent the

next state function of all the events $\varepsilon \in \mathcal{E}_k$, formally $\mathcal{N}_k = \bigcup_{\varepsilon \in \mathcal{E}_k} \mathcal{N}_\varepsilon$. Thus, the algorithm does not create a large, monolithic next state function representation. Instead, it divides the global next state function $\mathcal{N}$ into smaller parts according to the set of events $\mathcal{E}$ in the high-level model.

Symbolic encoding of the next state functions of events $\varepsilon \in \mathcal{E}_k$ relies on the following observation: $\mathcal{N}_\varepsilon(\langle s_1, \ldots, s_K \rangle)$ and $\mathcal{N}_\varepsilon(\langle s_1, \ldots, s_k \rangle) \times \{\langle s_{k+1}, \ldots, s_K \rangle\}$ are equivalent. From this fact we can derive two important properties of saturation: *1)* in the encoding of $\mathcal{N}_\varepsilon$ it is only required to encode the state changes of state variables $s_1, \ldots, s_k$, where $k = Top(\varepsilon)$, as well as *2)* it is possible to apply the individual $\mathcal{N}_\varepsilon$ functions in a finer granularity: $\mathcal{N}_\varepsilon$ is applicable not only on the full state space representation, but also on the local state space representation composed of state variables $s_1, \ldots, s_k$.

The saturation iteration strategy divides the global fixed-point computation into smaller parts, as it computes a local fixed-point with regard to a decision diagram node n_k. A node n_k is called *saturated*, if it represents a local state space computed as the fixed-point of the transitive closure of local next state relations: $\mathcal{S}(n_k) = \bigcup_{i:1 \le i \le k} \bigcup_{\varepsilon \in \mathcal{E}_i} \mathcal{N}_\varepsilon^*(\mathcal{S}(n_k))$, where $\mathcal{S}(n_k)$ is the set of states represented by node n_k [CMS03]. Building the MDD representation of the state space starts at the MDD of the initial state. Then the algorithm saturates every node in a bottom-up manner, by applying saturation recursively, if new states are discovered. In this way saturation avoids the peak size of the MDD to be much larger than the final size, which is a critical problem in traditional approaches.

2.3.2 Disjunctive and Conjunctive Partitioning

The *next-state* function $\mathcal{N}_\varepsilon$ of an event ε describes the states reachable from a given state in one step (i. e., with a single firing of a transition).

The global next-state of event ε can be defined as a product $\mathcal{N}_\varepsilon = \mathcal{N}_{(\varepsilon,1)} \times \ldots \times \mathcal{N}_{(\varepsilon,K)}$. This encoding enables building the next-state functions locally, but it requires a *Kronecker-consistent decomposition* [CMS03]. Ordinary Petri nets are Kronecker-consistent for any partitioning of the places [CMS03], but this is not guaranteed for more general models, like well-formed CPN models.

In [CMS03] the authors used a Kronecker matrix-based representation of $\mathcal{N}_\varepsilon$. In their solution the next-state function $\mathcal{N}_{(\varepsilon,i)}$ of the event ε (firing of the corresponding transition) in the ith submodel is encoded by a *Kronecker matrix* $\mathcal{K}_{(\varepsilon,i)}$ [Buc+00]. $\mathcal{K}_{(\varepsilon,i)}$ is a binary matrix and it belongs to event ε at level i. $\mathcal{K}_{(\varepsilon,i)}$ is constructed as follows: $\mathcal{K}_{(\varepsilon,i)}[j,k] = 1 \leftrightarrow k = \mathcal{N}_{(\varepsilon,i)}(j)$. These Kronecker matrices contain only the local next-state relations. Kronecker-consistent decomposition of the next-state representation turned out to be very efficient in practice.

In [CY05] the authors introduced a new next-state representation for saturation-based algorithms to be able to analyse a more general class of models. This solution uses MDDs with $2K$ levels to symbolically encode a next-state function $\mathcal{N}$ into the relation $\mathcal{R}$ of *from* and *to* variables: $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$. The variables $\mathbf{x} = (x_1, x_2, \ldots, x_K)$ in $\mathcal{R}$ refer to the current ("from") state, and the variables $\mathbf{x}' = (x'_1, x'_2, \ldots, x'_K)$ to the next ("to") states. $\mathcal{R}$ encodes the next-state function so that from state $\mathbf{x}$ we can go to states $\mathbf{x}'$ in one step.

The algorithm avoids creating a large, monolithic next-state relation, it divides the global next-state function into smaller parts instead. The first step is the *disjunctive decomposition* according to the set $\mathcal{E}$ of e events in the high-level model: $\mathcal{R} = \bigvee_{\varepsilon \in \mathcal{E}} \mathcal{R}_\varepsilon$. The relation $\mathcal{R}_\varepsilon$ is called *disjunct* in the following. In many cases the computation of these local relation $\mathcal{R}_\varepsilon$ is still expensive. So, in the next step the algorithm partitions the $\mathcal{R}_\varepsilon$ disjuncts *conjunctively* according to the *enabling* and *updating* relations [CY05]: $\mathcal{R}_\varepsilon = \bigwedge_{en} \mathcal{R}_{\varepsilon,en}^{enable} \wedge \bigwedge_{up} \mathcal{R}_{\varepsilon,up}^{update}$, where $\varepsilon \in \mathcal{E}$ and en refers to those indices (of the variables) which contribute to the enabling of the transition while up refers to those indices (of the variables) which are updated by the transition, $en, up \in supp(\varepsilon)$. The *enabling* relation is responsible for deciding if the given event is enabled in a certain state while the *update* relation decides

to which next states the exploration can go. Each $\mathcal{R}^{update}_{\varepsilon,up}$ and $\mathcal{R}^{enable}_{\varepsilon,en}$ relation is called a *conjunct* in the following. Note that in the construction and representation of the individual conjuncts and transition relations of the individual events only those variables are considered, which contribute to the firing of the event. In the decision diagram representation, the so-called identity reduction [CY05] is used to provide a compact storage.

The *enabling* relation consists of variables necessary for deciding the enabling of the transition related to a certain event. It contains only "from" variables (in $\mathbf{x}$), and does not change the value of any "to" variables (in $\mathbf{x}'$). The *updating* relation represents the local state changes, i. e., the local next-state functions, therefore it contains variables both from $\mathbf{x}$ and $\mathbf{x}'$.

The transition relation $\mathcal{R}_\varepsilon$ explicitly defines the relation of variables affected by the event ε. The set of variables affected by $\mathcal{R}_\varepsilon$ is denoted by $\mathbf{x}_\varepsilon$ (and their corresponding next-state variables $\mathbf{x'}_\varepsilon$). For all other variables, their relation can be described as the *identity relation*[CY05], therefore the following holds for the representation $\mathcal{R}_\varepsilon$: for all x_i, where $x_i \notin \mathbf{x}_\varepsilon$, $x_i = x'_i$. So in the representation, we have to consider only variables of $\mathbf{x}_\varepsilon$, therefore $\mathcal{R}_\varepsilon(\mathbf{x})$ can be represented as $\mathcal{R}_\varepsilon(\mathbf{x}_\varepsilon)$, while assuming that the omitted variables yield the identity relation. This will decrease the storage requirements of the representation as identity reduction is applied [CY05]. When the transition relation is further decomposed into conjuncts, their representation is further simplified. Beside the identity reduction, each conjunct will refer to a subset of $\mathbf{x}_\varepsilon$, other variables are considered *don't care* in the representation.

This fine-grained decomposition approach makes it possible to handle arbitrary finite next-state functions, which is the key to handle complex events efficiently. The smaller the partitions we create, the less computation they need and also it makes their representation smaller. The limit for the size of the partitioning comes from the used high-level modelling formalism.

2.3.3 State-space Exploration Based on Saturation

The most important properties and mathematical definitions have been introduced in the former sections from which an efficient state space traversal algorithm is constructed in this section. The pseudocode of the saturation algorithm is depicted on Algorithm 1 and Algorithm 2 [CMS05; CY05]. Saturation starts the exploration from the decision diagram representation of the initial states and traverses each node of the decision diagram and saturates it by calling function $Saturate$ of Algorithm 1. During saturation, all the possible states are discovered locally by computing the next-states of each locally fireable transition. Each local step is computed by the function $RelProd$ of Algorithm 2. $RelProd$ computes the reachable set of states from a given initial state through an event/transition represented by $\mathcal{N}_\varepsilon$, so function $RelProd$ computes the next-state set, formally: $\mathcal{S}(\mathbf{x}') = \{\{\mathbf{x}'\} \mid \exists \mathbf{x}_\varepsilon : \ RelProd(\mathbf{x}_\varepsilon, \mathbf{x'}_\varepsilon) \wedge \mathcal{S}(\mathbf{x})\}$ (according to [Bur+92; CMS05; CGP99]), where $\mathcal{S}(\mathbf{x})$ represents the set of states over variables in $\mathbf{x}$. The algorithm uses a caching mechanism which is not detailed here, the interested reader is referred to [CMS05]. Function $Union$ computes the set union operation of two decision diagrams. Operation $Confirm$ depicted on Algorithm 3 updates the enable and next-state relation if a new local state (localstate input argument of the function) is discovered. Function $ModelEnable_{\varepsilon,l}(i)$ evaluates if the local state i makes event ε enabled locally and $ModelUpd_{\varepsilon,l}(i)$ computes the set of locally reachable states from local state i through the firing of event ε. The function $Build$ constructs the enable and update relations from the conjuncts. For the sake of simplicity, the function $Build$ is called after each $RelProd$ function call to update the relations. Practically, the function $Build$ needs to be called only if new states are discovered at lower levels of the MDD.

Algorithm 1. Saturate

input : s_k : node
// s_k: node to be saturated,
output : node
if $s_k = \mathbf{1}$ **then**
  return $\mathbf{1}$;
Return result from cache if possible;
$k \leftarrow Level(s_k)$; // retrieve the actual level of the MDD
$t_k \leftarrow new\ Node_k$;
foreach $i \in \mathcal{S}_k : s_k[i] \neq \mathbf{0}$ **do**
  $t_k[i] \leftarrow Saturate(s_k[i])$;
repeat
  foreach $\varepsilon \in \mathcal{E} : k \in Top(\varepsilon)$ **do**
    $\mathcal{R}_\varepsilon \leftarrow \mathcal{N}_\varepsilon$ *as decision diagram*;
    foreach $s_k[i] \neq \mathbf{0} \wedge \mathcal{R}_\varepsilon[i][i'] \neq \mathbf{0}$ **do**
      $t_k[i'] \leftarrow t_k[i'] \cup RelProd(t_k[i], \mathcal{R}_\varepsilon[i][i'])$;
      if $i' \notin \mathcal{S}_k$ **then**
        $Confirm(k, i')$
      $Build(k)$;
until t_k *unchanged*;
$t_k \leftarrow PutInUniqueTable(t_k)$;
Put inputs and results in cache;
return t_k;

Algorithm 2. RelProd

input : $s_k, \mathcal{R}$: node
// s_k: node to be saturated,
// $\mathcal{R}$: next-state representation node
output : node
if $\mathcal{R} = \mathbf{1}$ **then**
  return s_k;
Return result from cache if possible;
$k \leftarrow Level(s_k)$; // retrieve the actual level of the MDD
$t_k \leftarrow new\ Node_k$;
foreach $s_k[i] \neq \mathbf{0} \wedge \mathcal{R}[i][i'] \neq \mathbf{0}$ **do**
  $t_k[i'] \leftarrow t_k[i'] \cup RelProd(s_k[i], \mathcal{R}[i][i'])$;
  if $i' \notin \mathcal{S}_k$ **then**
    $Confirm(k, i')$
$t_k \leftarrow PutInUniqueTable(Saturate(t_k))$;
Put inputs and results in cache;
return t_k;

Algorithm 3. Confirm

input : l : MDD level;
i : localstate
// l: level of the new state
// i: new local state to be confirmed
foreach $\varepsilon \in \mathcal{E} : l \in supp(\varepsilon)$ **do**
  if $ModelEnable_{\varepsilon,l}(i)$ **then**
    $\mathcal{R}_{\varepsilon,l}^{enable} \leftarrow \mathcal{R}_{\varepsilon,l}^{enable} \cup i$;
  $I' \leftarrow ModelUpd_{\varepsilon,l}(i)$;
  $\mathcal{R}_{\varepsilon,l}^{update} \leftarrow \mathcal{R}_{\varepsilon,l}^{update} \cup \{i\} \times I'$;
$\mathcal{S}_l \leftarrow \mathcal{S}_l \cup i$;

Algorithm 4. Build

input : l : MDD level
// l: actual level of MDD
foreach $\varepsilon \in \mathcal{E} : l = Top(\varepsilon)$ **do**
  $\mathcal{R}_\varepsilon^{enable} \leftarrow \bigwedge_{en \in supp(\varepsilon)} \mathcal{R}_{\varepsilon,en}^{enable}$;
  $\mathcal{R}_\varepsilon^{update} \leftarrow \bigwedge_{up \in supp(\varepsilon)} \mathcal{R}_{\varepsilon,up}^{update}$;
  $\mathcal{R}_\varepsilon \leftarrow \mathcal{R}_\varepsilon^{enable} \wedge \mathcal{R}_\varepsilon^{update}$;

Function $PutInUniqueTable$ places the newly computed nodes in the so-called *unique table* which serves as a storage and it also ensures that no redundant nodes can appear in the decision diagram.

2.4 Model Checking

Model checking is an automatic technique for verifying finite-state systems. Given a model, model checking decides whether the model fulfils the specification. Formally: let M be a Kripke structure (i. e., state transition graph). Let f be a formula of a given temporal logic (i. e., the specification). The goal of model checking according to [CGP99] is to find all states s of M such that $M, s \vDash f$. *Structural model checking* [CGP99] computes the results by exploring first the reachable states and

the state changes i. e., transition and traverses the possible behaviours to find those that satisfy the property. The properties are computed as fixed-points and according to the definitions below.

CTL (Computation Tree Logic) [CGP99] is a frequently used language for specifying requirements. CTL expresses atomic propositions and their temporal relations. It has an expressive syntax, and there are efficient algorithms for its analysis. Operators occur in pairs in CTL: the path quantifier, either A (on all paths) or E (there exists a path), is followed by the tense operator, one of X (next), F (future or finally), G (globally), and U (until). However, we only need to implement EX, EU, EG of the eight possible pairings due to duality [CGP99].
The semantics of the three required CTL operators are as follows (where p and q are predicates):

- EX: $M, s^0 \vDash \mathsf{EX}\ p$ iff for model M, $\exists s^1 \in \mathcal{N}(s^0)$ state such that $s^1 \vDash p$. This means that EX corresponds to the inverse $\mathcal{N}$ function, applying one step backward through the next-state relation.
- EG: $M, s^0 \vDash \mathsf{EG}\ p$ iff for model M, $\exists I = (s^0, s^1, s^2, \ldots)$ infinite path such that $\forall j \geq 0 : s^{j+1} \in \mathcal{N}(s^j)$ and $s^j \vDash p$, so there is a strongly connected component containing states satisfying p.
- EU: $M, s^0 \vDash \mathsf{E}(p\ \mathsf{U}\ q)$ iff for model M, $\exists n \geq 0, \exists I = (s^0, s^1, s^2, \ldots, s^n)$ path such that $\forall 1 \leq j < n : s^{j+1} \in \mathcal{N}(s^j), \forall 0 \leq k < n : s^k \vDash p$ and $s^n \vDash q$.

Various techniques are developed to handle the complexity yielded by recent systems. One of them is *symbolic model checking*, which uses special encoding to be able to store the huge number of reachable states of the systems. Decision diagrams provide a compact representation for the encoded state space, and advanced algorithms are used for model-checking.

2.5 CEGAR for Petri Nets

Petri nets have a simple structure, which makes it possible to use strong structural analysis techniques based on the so-called *state equation*. As the structural analysis is independent of the initial state, it can handle even infinite state problems. Unfortunately, its pertinence to practical problems, such as reachability analysis, has been limited. An algorithm [WW11] using CounterExample-Guided Abstraction Refinement (CEGAR) extended the applicability of state equation based reachability analysis. This section is based on [4],[10] and [15] and it introduces the CEGAR method and its application for the Petri net reachability problem.

2.5.1 Petri Net State Equation

The *incidence matrix* of a Petri net is a matrix $C_{|P|\times|T|}$, where $C(i,j) = w^+(p_i, t_j) - w^-(p_i, t_j)$. Let m and m' be markings of the Petri net, then the *state equation* takes the form $m + Cx = m'$. Any vector $x \in (\mathbb{Z}_0^+)^{|T|}$ fulfilling the state equation is called a *solution*. Note that for any realizable firing sequence σ leading from m to m', the Parikh image of the firing sequence fulfills the equation $m + C\wp(\sigma) = m'$. On the other hand, not all solutions of the state equation are Parikh images of a realizable firing sequence. Therefore, the existence of a solution for the state equation is a necessary but not sufficient criterion for the reachability. A solution x is called *realizable* if a realisable firing sequence σ exists with $\wp(\sigma) = x$.

T-invariants. A vector $x \in (\mathbb{Z}_0^+)^{|T|}$ is called a *T-invariant* if $Cx = \mathbf{0}$ holds. A realisable T-invariant represents the possibility of a cyclic behaviour in the modelled system since its complete occurrence does not change the marking. However, while firing the transitions of the invariant, some intermediate markings can be interesting for us.

Solution space. Each solution x of the state equation $m + Cx = m'$, can be written as the sum of a *base vector* and the linear combination of T-invariants [WW11], which can formally be written as $x = b + \sum_i n_i y_i$, where b is a base vector and n_i is the coefficient of the T-invariant y_i.

2.5.2 The CEGAR Approach

Counterexample-guided abstraction refinement (CEGAR) is a general approach for analysing systems with large or infinite state spaces. The CEGAR method works on an abstraction of the original model, which has a less detailed state space representation. During the iteration steps, the CEGAR method refines the abstraction using the information from the explored part of the state space. When applying CEGAR on the Petri net reachability problem [WW11], the initial abstraction is the state equation. Solving the state equation is an integer linear programming problem [DT97], for which the ILP solver tool can yield one solution, minimising a target function of the variables. Since the algorithm seeks the shortest firing sequences leading to the target marking, it minimizes the function $f(x) = \sum_{t \in T} x(t)$. The feasibility of the state equation is a necessary, but not sufficient criterion for reachability, so the following situations are possible:

- If the state equation is infeasible, the necessary criterion does not hold, thus the target marking is not reachable.
- If the state equation has a solution which is realisable by some firing sequence, the target marking is reachable.
- If the state equation has an unrealizable solution, it is a counterexample, and the abstraction has to be refined.

The purpose of the abstraction refinement is to exclude counterexamples from the solution space without losing any realisable solutions. For this purpose, the CEGAR approach uses linear inequalities over transitions, called *constraints*.

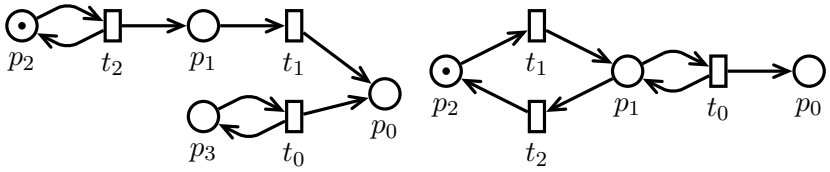
Constraints. Two types of constraints were defined by Wimmel and Wolf [WW11]:

- *Jump constraints* have the form $|t_i| < n$, where $n \in \mathbb{Z}_0^+$, $t_i \in T$ and $|t_i|$ represents the firing count of the transition t_i. Jump constraints can be used to switch between base vectors, exploiting their pairwise incomparability.
- *Increment constraints* have the form $\sum n_i |t_i| \geq n$, where $n_i \in \mathbb{Z}$, $n \in \mathbb{Z}_0^+$, and $t_i \in T$. Increment constraints can be used to reach non-base solutions.

As an example, consider the Petri net in Figure 2.2a with the reachability problem $(1, 0, 1, 0) \in R(PN, (0, 0, 1, 0))$. There are two base vectors for this problem: $(1, 0, 0)$ (firing t_0) and $(0, 1, 1)$ (firing t_1 and t_2). The ILP solver yields the solution $(1, 0, 0)$ first, which is unrealizable, but using the jump constraint $|t_0| < 1$, the ILP solver can be forced to produce the realizable solution $(0, 1, 1)$. Consider now the Petri net in Figure 2.2b with the reachability problem $(1, 0, 1) \in R(PN, (0, 0, 1))$. The only base vector for this problem is the vector $(1, 0, 0)$ (firing t_0), which is unrealizable. Using an increment constraint $|t_1| \geq 1$, the ILP solver can be forced to add the T-invariant $\{t_1, t_2\}$ to the new solution $(1, 1, 1)$, which is realizable by the firing sequence $\sigma = (t_1, t_0, t_2)$.

2.5.2.1 Partial solutions

Given a Petri net $PN = (P, T, E, W)$ and a reachability problem $m' \in R(PN, m_0)$, a *partial solution* is a tuple $ps = (\mathcal{C}, x, \sigma, r)$, where:



(a) Jump constraint example (b) Increment constraint example

Figure 2.2: Example nets for jump and increment constraints

- $\mathcal{C}$ is the set of (jump and increment) constraints, together with the state equation they define the ILP problem,
- x is the minimal solution satisfying the state equation and the constraints of $\mathcal{C}$,
- $\sigma \in T^*$ is a maximal realisable firing sequence, with $\wp(\sigma) \leq x$, i.e., each transition can fire as many times as it is included in the solution vector x and if it is enabled it must fire,
- $r = x - \wp(\sigma)$ is the remainder vector.

Generating partial solutions. Partial solutions can be produced from a solution vector x (and a constraint set $\mathcal{C}$) by firing as many transitions as possible. For this purpose, the algorithm uses a "brute force" method. The algorithm builds a tree with markings as nodes and occurrences of transitions as edges. The root of the tree is the initial marking m_0, and there is an edge labeled by t between nodes m_1 and m_2 if $m_1[t\rangle m_2$ holds. On each path leading from the root of the tree to a leaf, each transition t_i can occur at most $x(t_i)$ times. Each path to a leaf represents a maximal firing sequence, thus a new partial solution. Even though the tree can be traversed only storing one path in the memory at a time using depth-first search, the size of the tree can grow exponentially. Some optimisations to reduce the size of the tree are presented later in this section.

A partial solution is called a *full solution* if $r = 0$ holds, thus $\wp(\sigma) = x$, which means that σ realizes the solution vector x. For each realizable solution x of the state equation there exists a full solution [WW11]. This full solution can be reached by continuously expanding the minimal solution of the state equation with constraints.

Consider now a partial solution $ps = (\mathcal{C}, x, \sigma, r)$, which is not a full solution, i.e., $r \neq 0$. This means that some transitions could not fire enough times. There are three possible situations in this case:

1. x may be realizable by another firing sequence σ', thus a full solution $ps' = (\mathcal{C}, x, \sigma', 0)$ exists.
2. By adding jump constraints, greater, but pairwise incomparable solutions can be obtained.
3. For transitions $t \in T$ with $r(t) > 0$ increment constraints can be added to increase the token count in the input places of t, while the final marking m' must be unchanged. This can be achieved by adding new T-invariants to the solution. These T-invariants can "borrow" tokens for transitions in the remainder vector.

2.5.2.2 Generating constraints

Jump constraints. Each base vector of the solution space can be reached by continuously adding jump constraints to the minimal solution [WW11]. In order to reach non-base solutions, increment constraints are needed, but they might conflict with previous jump constraints. Jump constraints are only needed to obtain a different base solution vector. However, after the computation of the base solution, jump constraints can be transformed into equivalent increment constraints [WW11].

Increment constraints. Let $ps = (\mathcal{C}, x, \sigma, r)$ be a partial solution with $r > 0$. This means that some transitions (in r) could not fire enough times. The algorithm uses a heuristic to find the places and number of tokens needed to enable these transitions. If a set of places actually needs n ($n > 0$) tokens, the heuristic estimates a number from 1 to n. If the estimate is too low, this method can be applied again, converging to the actual number of required tokens. The heuristic consists of the following three steps:

1. First, the algorithm builds a dependency graph [VH10] to collect the transitions and places that are of interest. These are transitions that could not fire, and places that disable these transitions. Each source SCC[1] of the dependency graph has to be investigated because it cannot get tokens from other components. Therefore, an increment constraint is needed.
2. The second step is to calculate the minimal number of missing tokens for each source SCC. There are two sets of transitions, $T_i \subseteq T$ and $X_i \subseteq T$. If one transition in T_i becomes fireable, it may enable all the other transitions of the SCC, while transitions in X_i cannot activate each other. Therefore their token shortage must be fulfilled at once.
3. The third step is to construct an increment constraint c for each source SCC from the information about the places and their token requirements. These constraints will force transitions (with $r(t) = 0$) to produce tokens in the given places. Since the final marking is left unchanged, a T-invariant is added to the solution vector.

When applying the new constraint c, three situations are possible depending on the T-invariants in the Petri net:

- If the state equation and the set of constraints become infeasible, this partial solution cannot be extended to a full solution, therefore it can be skipped.
- If the ILP solver can produce a solution $x + y$ (with y being a T-invariant), new partial solutions can be found. If none of them helps to get closer to a full solution, the algorithm can get into an infinite loop, but no full solution is lost. A method to avoid this non-termination phenomenon will be discussed later in this section.
- If there is a new partial solution ps' where some transitions in the remainder vector could fire, this method can be repeated.

Reachability of solutions [WW11]. If the reachability problem has a solution, a realisable solution of the state equation can be reached by continuously adding constraints, transforming jumps before increments.

2.5.2.3 Optimisations

Wimmel and Wolf [WW11] also presented some methods for optimization. The following are important for this work:

- **Stubborn set:** The stubborn set method [VH10] investigates conflicts, concurrency and dependencies between transitions, and reduces the search space by filtering the transitions. The stubborn set method usually leads to a search tree with a lower degree.
- **Subtree omission:** When a transition has to fire more than once ($x(t) > 1$), the stubborn set method may not provide an efficient reduction. The same marking is often reached by firing sequences that are only different in the order of transitions. During the abstraction refinement, only the final marking of the firing sequence is important. If a marking m' is reached by firing

[1] Source strongly connected component, i.e., one without incoming edges from other components.

the same transitions as in a previous path, but in a different order, the subtree after m' was already processed. Therefore, it is no longer of interest.

- **Filtering T-invariants:** After adding a T-invariant y to the partial solution $ps = (\mathcal{C}, x, \sigma, r)$, all the transitions of y may fire without enabling any transition in r, yielding a partial solution $ps' = (\mathcal{C}', x + y, \sigma', r)$. The final marking and remainder vector of ps' is the same as in ps, therefore the same T-invariant y is added to the solution vector again, which can prevent the algorithm from terminating. However, while firing the transitions of y, the algorithm could get closer to enabling a transition in r. These intermediate markings should be detected, and be used as new partial solutions.

Chapter 3

Model Checking of High Level Models

This section introduces the new algorithms developed for the verification of high-level models. The need for the introduced techniques originated in a project where we aimed to verify a critical function (so-called PRISE function) of a control system of a nuclear power plant. Coloured Petri nets (CPN) was proposed to be used in [Ném+09] as a convenient modelling formalism to describe the PRISE logic.

In this chapter, I propose a saturation-based algorithm to verify complex CPN models, and I further extend the algorithm to reduce runtime and memory requirements. The proposed solution is the first algorithm being able to solve the verification problem of the system of the PRISE industrial case study.

In the first part of the section, I draw the motivation and the used high-level modelling formalism. In the next part, I introduce the new saturation-based verification algorithm using fine-grained decomposition and its improvement. Finally, the PRISE use-case is outlined, and the verification results are presented.

Publications related to this chapter. The results of this thesis were published in [5], [16] and [22] and this chapter is based on that papers.

Implementation and contributors. All the algorithms presented in this chapter were implemented and made available in the PetriDotNet framework. The implementation of the presented algorithms is the result of the whole PetriDotNet team. The algorithms of Section 3.4 were implemented by my students, Dániel Darvas and Attila Jámbor. The algorithm presented in Section 3.5 was implemented by Attila Jámbor under my supervision.

3.1 Motivation

The motivation of this section is a case study, the PRISE safety function which has a huge state space ($> 10^{12}$ states) and many different behaviours and functionalities, therefore efficient automatic methods are indispensable to prove its correctness. The first successful verification attempt was reported in [NB09], where the authors used coloured Petri nets and the Design/CPN modelling tool. Design/CPN has a simple explicit state model checker without built-in reduction methods. Thus it was not able to explore the complete state space of the model, only a small part (approx. $4 \cdot 10^5$ states) could fit into the memory. The authors used state space reduction techniques then partitioned the state space and separately analysed different subspaces. Finally, they have managed to obtain reduced subspaces with a manageable size and could complete the formal verification.

Later, a student of our research group created a formal model of the PRISE safety function in the Uppaal tool [Tót09]. The modelling formalism of Uppaal uses networks of timed automata extended with data structures and a data manipulation language. It has symbolic state space representation, built-in state space reduction methods, and a (partial) Computation Tree Logic (CTL) model checker. Unfortunately, Uppaal has also failed to explore the complete state space due to memory overflow. Nevertheless, by reducing the model, we have at least succeeded proving some of the requirements with Uppaal.

In [Tót09], she also tried other symbolic approaches. The first choice was the Symbolic Analysis Laboratory (SAL) model checker. Sadly, this attempt to verify the PRISE safety function has failed as well, even though SAL uses a Binary Decision Diagram based efficient state space representation. Without being able to trace the low-level operation of SAL, the assumption is that the next-state relation grew too large: the state space explosion turned into decision diagram explosion in this case.

Our research group tried using other existing advanced Petri net verification methods [18]. These approaches, however, operate on simple, uncoloured Petri nets. Therefore we have developed an automated systematic conversion procedure to convert the coloured Petri net model of PRISE to a simple Petri net first. In order to verify the Petri net model of the PRISE system, we have tried various algorithms:

- We have built an unfolding [ERV02] based verification tool. As unfolding is efficient for asynchronous models, we expected that it could overcome the state space explosion problem. Unfortunately, this approach still ran out of memory due to the long distinctive trajectories.
- In [CMS03] the authors showed the saturation-based efficient symbolic state space generation and model checking method for asynchronous systems, especially for Petri nets. We have implemented and ran the algorithm with different settings on the converted simple net, but the algorithm ran out of memory. Unfortunately, the size of the converted model was too large, which caused both the state space representation and the next-state relation to exceed our resources.

The common weakness of the listed past approaches is that they could only reach partial success, as none of them was able to explore the full state space of the PRISE safety function.

Former investigations of the problem stated that Coloured Petri nets provide a suitable formalism to efficiently design the formal model of the PRISE systems. This motivated our work to provide an efficient model checking algorithm which is able to handle the complexity of real-life industrial systems.

In this section, the saturation algorithm is extended to support the verification of high-level models. At first, the used coloured Petri net formalism is introduced according to [16]. A new decomposition algorithm [16] is introduced to handle the complex transitions of high level coloured Petri net models, and a lazy transition relation construction algorithm [5] is developed to improve the efficiency of the verification further. An industrial case study [16][22] is used to illustrate the applicability of the approach.

3.2 High-level Models: Coloured Petri Nets

Coloured Petri nets (CPN) provide a high level language to develop formal models, and they yield a compact representation for complex systems. The coloured Petri net formalism enriches ordinary Petri nets with complex data structures [JK09]. There are many types of coloured Petri nets, in this paper, a variant of well-formed coloured Petri nets from [1] and [16] is used that is also supported by the PetriDotNet tool. Well-formed coloured Petri nets have the same expressive power as ordinary

Petri nets, but they yield a more compact representation of systems. In the following, I will introduce and use the definition from [JK09], which was slightly modified in [1].
Formally, a coloured Petri net is a tuple $CPN = (P, T, A, \Sigma, V, C, G, E, M_0)$, where

- P is the finite set of *places*,
- T is the finite set of *transitions*, with $P \cap T = \emptyset$,
- $A \subseteq (P \times T) \cup (T \times P)$ is the finite set of *arcs*,
- Σ is the finite set of non-empty *colour sets*, i. e., types,
- V is the finite set of typed *variables* such that $\forall v \in V : Type[v] \in \Sigma$,
- $C\colon P \mapsto \Sigma$ is the *colour set function* assigning a colour set to each place,
- $G\colon T \mapsto EXPR$ is the *guard function* assigning a guard expression to each transition $t \in T$, with $Type[G(t)] = Bool$,
- $E\colon A \mapsto EXPR$ is the *arc expression function* assigning an expression to each arc a, with $Type[E(a)] = C(p)_{MS}$ if a is connected to place p and $C(p)_{MS}$ is the multiset over $C(p)$,
- M_0 is the *initial marking*.

A *marking* M is a function mapping each place p into a multiset of values $M(p)$ over the colour set $C(p)$. Individual elements of $M(p)$ are called *tokens*. A *multiset* m over a set S is a function $m\colon S \mapsto \mathbb{Z}^+$, where $m(s)$ is the number of occurrences of the element $s \in S$ in m.

Firing rules of transitions in Petri nets only depend on the marking. However, in coloured Petri nets the arc and guard expressions have to be considered as well. Let $Var[t]$ denote the variables of a transition $t \in T$, which includes the free variables appearing in the guard of t and in expressions on arcs connected to t. A *binding* b of a transition t assigns each variable $v \in Var[t]$ a value $b(v) \in Type[v]$. The set of all bindings for a transition t is denoted by $B(t)$. A *binding element* is a pair (t, b) of a transition t and a binding $b \in B(t)$. Given a binding element (t, b), let $G(t)\langle b\rangle$ denote the result of evaluating a guard in the binding b. Similarly, let $E(p, t)\langle b\rangle$ and $E(t, p)\langle b\rangle$ denote the result of evaluating arc expressions. If no such arcs exist, $E(p, t)\langle b\rangle = \emptyset$ and $E(t, p)\langle b\rangle = \emptyset$.

A binding element (t, b) is *enabled* in a marking M if (1) $G(t)\langle b\rangle$ evaluates to true and (2) $E(p, t)\langle b\rangle \leq M(p)$ for each $p \in P$. An enabled binding element (t, b) may occur leading to the marking M' defined by $M'(p) = M(p) - E(p, t)\langle b\rangle + E(t, p)\langle b\rangle$ for each $p \in P$, i. e., input arcs remove tokens, while output arcs produce tokens as specified by the arc expressions. Firing sequences and reachability is defined the same way as for P/T nets.

3.3 Saturation for CPN Models

As it was discussed before, existing low-level models and inefficient model checking algorithms prevented us from reaching our goal: to verify the PRISE safety function fully. Therefore, according to the findings of [NB09], I selected coloured Petri nets as the modelling formalism, and saturation as the basis of state space exploration and model checking. However, saturation did not support CPNs at that time, thus I had to adopt the saturation algorithm to handle CPN models. In this section, I overview how the state space of coloured Petri nets can be explored with the help of saturation.

Well-formed coloured Petri nets can model complex systems in a compact form by utilising the data content of tokens instead of pure structural constructs. However, this compactness takes its price during traversal: local state spaces and transition relations of the submodels in a decomposed CPN are typically much larger and more complex than in simple Petri nets. Previous research [CY05]

proved that the smaller the partitions are, the more efficient saturation becomes, since the creation and maintenance of the smaller parts require significantly less resources. In the following, I extend the saturation-based state space traversal to handle CPN models. Then I extend this approach in Section 3.4 to be able to handle the complex transition relation of CPN models.

3.3.1 Iteration Strategy for CPN

The basic prerequisites for saturation are decomposition and encoding. Decomposing CPN models can be done similarly to the decomposition of ordinary Petri nets. The approach of conjunctive-disjunctive next-state function decomposition lets us choose arbitrary decomposition of the CPN model. Encoding consists of two subtasks: encoding the states and encoding the next-state relations. The used data structures i. e., the colour types of the places increase the size of the local state spaces. However, as long as the local state spaces do not grow prohibitively large, the iteration strategy will provide an efficient solution for the state space exploration.

The iteration strategy of saturation exploits locality. CPN models can concisely represent systems by using the high-level constructs: this way the introduced dependencies will ruin locality for some cases. Applying saturation for CPNs extends the set of models verified by saturation, but the new algorithm cannot replace former saturation algorithms for all the problems. In general, when saturation-based analysis of the CPN model is not feasible, one can still unfold the CPN to an ordinary PN and try to use traditional saturation.

3.3.2 Encoding Next-state Relations

The biggest challenge in adapting saturation to CPNs is the construction of the next-state relations. As it was discussed in Section 2.3.2, the transition relation can be decomposed. According to [CY05], the relation $\mathcal{R}_\varepsilon = \mathcal{R}_\varepsilon^{enable} \wedge \mathcal{R}_\varepsilon^{update}$ has to be created for each transition (event ε). Further conjunctive decomposition $\mathcal{R}_\varepsilon^{enable} = \bigwedge_k \mathcal{R}_{\varepsilon,k}^{enable}$ and $\mathcal{R}_\varepsilon^{update} = \bigwedge_k \mathcal{R}_{\varepsilon,k}^{update}$ can significantly reduce the computational cost. The smaller the conjuncts are, the easier and cheaper it is to construct them.

Kronecker consistent models can yield fine granularity by decomposing the conjuncts to refer to only single state variables: this turned out to be very efficient in practice. However, this granularity cannot be achieved for arbitrary models. CPN models can represent complex logic in a compact form, and the dependencies in the transition relation might become intricate, so the decomposition rules applied to Petri nets do not lead to a Kronecker consistent decomposition of CPN models. This implies that each $\mathcal{R}_{\varepsilon,k}^{enable}$ and $\mathcal{R}_{\varepsilon,k}^{update}$ will refer to a group of variables. When each group is smaller than the set of all variables affected by the event, the decomposition still increases the efficiency. Unfortunately, the transition relation of CPNs usually cannot be decomposed using the conjunctive decomposition rules because of the arc and guard expressions: they often express intricate dependencies among the state variables. It is very seldom the case that the transition relation is conjunctively decomposable. In addition, separating the enable and update parts of the transition relation often does not yield any advantage for CPNs, as there is no construct in the language to express enabledness (contrary to [CY05]): a transition is enabled if the input arcs can take enough tokens from input places (so the variables representing the input places have certain values). However, input arcs also consume tokens, so they constitute also the update relations. The result is that the relation encoded in $\mathcal{R}_\varepsilon^{enable}$ is fully encoded also in $\mathcal{R}_\varepsilon^{update}$.

These reasons lead to the situation that when the algorithm builds the relations, it has to traverse all possible local state changes for all the places connected to the transition. This turned out to be very expensive in practice and prevented the algorithms to verify our industrial case study, as we could

only solve smaller models by using the traditional decomposition algorithm. In the next section I show a different decomposition method which proved to be efficient for CPNs.

3.4 Disjunctive-Conjunctive Decomposition for CPN Models

This section is based on the following papers: [16] and [22].

Disjunctive-conjunctive decomposition proved its efficiency for many classes of Petri nets. Despite the fact that these Petri net classes are enriched with various constructs to increase expressiveness, they do not support data types i. e., colour types. The introduced decomposition was tailored to the characteristics of variants of the ordinary Petri net formalism ([CY05]). Coloured Petri nets brought not only various data types, but also arc and guard variables into consideration. In addition, complex data types can have a huge number of different values depending on the size of the domain, further increasing the complexity of the analysis by increasing the number of local states to be traversed. In this section, I show how we can decompose the representation of the transition relations according to the structure of a coloured Petri net. We will exploit the locality of Coloured Petri nets and decompose the transition relations into manageable pieces. The basic building blocks of the decomposition are the places of the Petri net so that the new algorithm will use the places as a guide for the decomposition. However, in case of considering only places as state variables in the decision diagram representation, then complex guard functions will not enable the fine-grained decomposition, as the effect of the transition is global with regard to a transition.

3.4.1 Overview of the Approach

As it was discussed before, the iteration strategy of saturation provides efficiency, but we have to extend the algorithm to handle the complex transition relation of CPN models. In order to explore the possible next-state relations and build the representation on-the-fly during the traversal, the algorithm has to solve the following tasks:

1. Compute the representation according to the structure of the net.
2. Build the initial next-state relation.
3. Update the next-state relation when a new local state is discovered.

In the following sections, I will show a new efficient representation of the next-state relations of CPN models. I also introduce algorithms for the efficient construction and on-the-fly update of the next-state relation.

3.4.2 Decomposition Algorithm for CPN

I will introduce the new representation and a fine-grained decomposition method for the next-state relation of CPN models. The new algorithm exploits the formerly introduced disjunctive decomposition: at first, the system-level next-state relation is decomposed according to the events as $\mathcal{R} = \bigvee_{\varepsilon \in \mathcal{E}} \mathcal{R}_\varepsilon$, so the decomposition of the whole next-state relation into smaller relations is applied according to the literature. However, the conjunctive decomposition and construction of the individual next-state relations ($\mathcal{R}_\varepsilon$) are improved in this section. In the following, we assume that the encoding mapped each place to a state variable, for the sake of simplicity. However, the algorithm is not restricted to this encoding scheme.

Next-state storage. The next-state representation of CPN models is constructed from the constraint of the guard expression, and the local effects of the variable bindings of the arc expressions. In order to be able to represent the guard expression in the symbolic next-state representation, new variables are needed: the symbolic representation of the next-state relation contains the "from" state variables and the corresponding "to" state variables ($\mathbf{x}$ and $\mathbf{x}'$) and the next-state representation also contains new, auxiliary variables such as for each $v \in V$ a corresponding state variable x_v is included into the symbolic next-state representation.

Next-state representation and decomposition. The new fine-grained decomposition of the transition relation of complex CPN models exploits the following facts. The constraint yielded by the guard and arc expressions does not change during the state space exploration, and these constraints can be computed without knowing the exact markings of the system. The main idea is to decompose the state dependent and independent parts of the transition relation and then use the following rule for computing the transition relation: $\mathcal{R}_{\varepsilon}^{\gamma} \wedge \bigwedge_k \mathcal{R}_{\varepsilon,k}^{update}$. In the decomposition, relation $\mathcal{R}_{\varepsilon}^{\gamma}$ represents the constraint of the guard and arc expressions and each $\mathcal{R}_{\varepsilon,k}^{update}$ represents the changes caused by the bindings of the variables to the corresponding state variables.

Next-state construction. To construct the relation $\mathcal{R}_{\varepsilon}^{\gamma}$, we have to introduce new variables into the encoding. The new variables will represent variables of the set $Var[\varepsilon]$ i. e., the variables of the input and output arcs and the guard expression of event ε. For each $v \in Var[\varepsilon]$ the transition relation encoding is extended with a corresponding variable x_v and $\mathcal{R}_{\varepsilon}^{\gamma}$ will only refer to these new variables. Formally, $\mathcal{R}_{\varepsilon}^{\gamma}$ will represent the set of valid bindings for all the variables $v \in Var[\varepsilon]$ such that $\forall b \in B(t)$, $b \in \mathcal{R}_{\varepsilon}^{\gamma}$ if $G(t)\langle b \rangle$ evaluates to true and the arc expressions $E(p,t)\langle b \rangle$ and $E(t,p)\langle b \rangle$ are also satisfied.

The relation $\mathcal{R}_{\varepsilon}^{\gamma}$ has similar role as the enable relation in the former approach. The main extensions of $\mathcal{R}_{\varepsilon}^{\gamma}$ compared to the formerly introduced enable relation are summarized in the following:

- The set of state variables has to be extended with auxiliary variables to be able to encode the relation $\mathcal{R}_{\varepsilon}^{\gamma}$.
- The relation $\mathcal{R}_{\varepsilon}^{\gamma}$ represents not only the variable bindings which are enabled but also those bindings which will change the state variables as the transition fires.

However, the introduced auxiliary variables are not present in the final $\mathcal{R}_{\varepsilon}^{update}$ relation as they are only needed to build the relation efficiently, but we do not need them when using the relation.

During the state traversal, the goal is to update the next-state relation locally. This locality is provided by the introduced auxiliary variables as they support the handling of the state variables independently from each other, so we can decompose the relation and build an individual update relation $\mathcal{R}_{\varepsilon,x}^{update}$ corresponding to each state variable x. For each state variable x, the local update relation $\mathcal{R}_{\varepsilon}^{update}$ is constructed by using auxiliary variables from the set $V = Var[E(p,t)] \bigcup Var[E(t,p)]$, where state variable x corresponds to the place p. The representation of the local state changes in case of transition firing is constructed, the effects of the variable assignments to the marking of the place are encoded by building a vector $\mathbf{v}$ of variables $v \in V$ and constructing the relation $\mathcal{R}_{\varepsilon}^{update}(\mathbf{v}, x, x')$. These relations are continuously built and updated during the state space traversal and it will encode the effects of $E(p,t)\langle b \rangle$ and $E(t,p)\langle b \rangle$ to place p in the corresponding markings when the firing of transition t happens (event ε).

Projection to the state variables. During state space traversal the algorithm does not need information regarding the assignments of the variables in the arc and guard expressions. This fact reduces the size of the next-state relations as the information contained in these auxiliary variables can be omitted. Putting things together, the next-state relation for the event ε is constructed according to the following rule: $\mathcal{R}_\varepsilon = \{(\mathbf{x}, \mathbf{x}') \mid \exists \mathbf{v}\ \mathcal{R}_\varepsilon^\gamma(\mathbf{v}) \wedge \bigwedge_k \mathcal{R}_{\varepsilon,k}^{update}(\mathbf{v}_k, x_k, x'_k)\}$. The guard relation is computed from the guard and arc expressions, the next-state relations indexed by k are constructed during state space traversal, and they correspond to the local state changes caused by the variable bindings of the transition firing. Existential quantification is used to map the relation to the state variables i. e., to omit the auxiliary variables from the relation.

3.4.3 Event Handling Algorithm

Coloured Petri nets can model complex systems in a very compact form by utilising the data content of tokens instead of pure structural constructs. However, this compactness takes its toll during state traversal: the local state spaces of the sub-models in a decomposed CPN are typically much larger and more complex than in simple Petri nets. Moreover, in CPNs fewer variables are used to encode the same set of states into decision diagrams, thus there is less redundancy in the state space representation, resulting in a less efficient form of storage. Previous researches proved that the smaller the partitions are, the more efficient saturation becomes, since the creation and maintenance of the smaller parts require significantly less resources. The aim of the *conjunctive refinement* of the partitioning, as described in Section 3.4.2, is to further decompose the state transitions into smaller parts and to treat these parts separately and efficiently. The steps of this *event handling* process are shown in Figure 3.1.

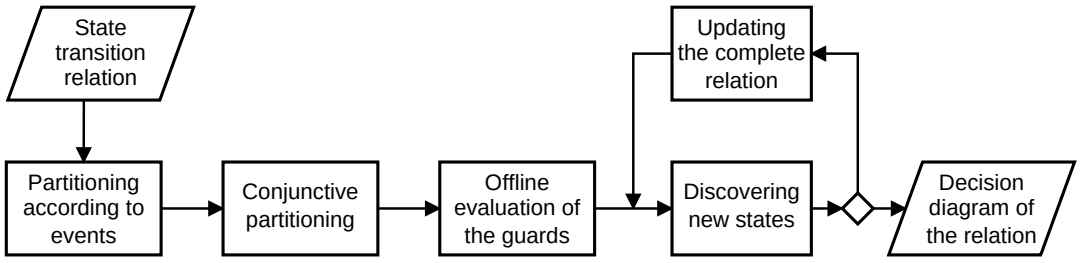


Figure 3.1: Workflow of the event handling

1. *Partition the next-state relation according to events (i. e., transitions).* The global state transition relation is partitioned disjunctively driven by the events. The partitioning is done according to the following observation: $\mathcal{N} = \bigcup_{\forall \varepsilon} \mathcal{N}_\varepsilon$ and the relation $\mathcal{R}_\varepsilon$ for each event $\varepsilon \in \mathcal{E}$ is stored in separate decision diagram. The original state transition relation can be calculated as $\mathcal{R} = \bigvee_{\forall \varepsilon \in \mathcal{E}} \mathcal{R}_\varepsilon$.
2. *Conjunctive decomposition by introducing auxiliary variables.* The above partitioning is further refined by splitting the $\mathcal{R}_\varepsilon$ state transition relation of each ε event into smaller parts according to the formerly introduced rule: $\mathcal{R}_\varepsilon = \mathcal{R}_\varepsilon^\gamma(\mathbf{v}) \wedge \bigwedge_k \mathcal{R}_{\varepsilon,k}^{update}(\mathbf{v}_k, x_k, x'_k)\}$. The fireable bindings of the variables are stored in the relations $\mathcal{R}_{\varepsilon,k}^{update}(\mathbf{v}_k, x_k, x'_k)$ encoded as MDDs. Furthermore, another MDD is created to represent the constraints imposed by the guards associated with the events. This MDD, denoted as $\mathcal{R}_\varepsilon^\gamma(\mathbf{v})$, stores those bindings of the variables in the input and output arc expressions of the transition for which the guard evaluates to true.

3. *Off-line evaluation of guards.* The symbolic representation for the guard constraint associated to the event ε is created before the iteration of the saturation: a decision diagram is built for $\mathcal{R}_{\varepsilon}^{\gamma}$. The decision diagram is built by traversing the possible bindings, which evaluate the guard expression to true. The variable bindings stored in this MDD need not be updated later during saturation.

4. *Discovering new states.* As soon as a new state is discovered during the iteration of the saturation, the fireable state transitions from this new state must be instantiated immediately. When a new state transition of event ε is found affecting variable x_k, then the MDD representing $\mathcal{R}_{\varepsilon,k}^{update}(\mathbf{v}_k, x_k, x'_k)$ is expanded with the new local state transitions. Additionally, the variable bindings that make the state transitions fireable are also stored. This update operation is realized in the function *CPNConfirm* function in Algorithm 5: the saturation algorithm uses this function instead of function *Confirm*.

5. *Updating the complete relation.* Since the original iteration order of the saturation is preserved in our algorithm, the complete state transition relation must be recreated by intersecting the MDDs of the partial relations and projecting the relation to the state variables.This functionality is implemented in the function *CPNBuild* of Algorithm 6: function *CPNBuild* will be called instead of *Build* in the saturation algorithm.

Encode self-loops The decomposition algorithm has to pay special attention to self-loops. As it was formerly mentioned, the conjuncts will encode the effects of $E(p,t)\langle b\rangle$ and $E(t,p)\langle b\rangle$ to place p (variable x), in case of self-loops, the algorithm compiles the input and output arcs into a single $\mathcal{R}_{\varepsilon}^{update}(\mathbf{v}, x, x')$ relation and ensures the correct construction of the transition relation.

3.4.4 Off-Line Evaluation of Guards

In order to exploit the fine-grained decomposition and the in-built caching mechanisms of MDDs, as much of the transition relation as possible should be built off-line. The decomposition helps us to compute off-line the relation representing the guard and arc expressions $\mathcal{R}_{\varepsilon}^{\gamma}(\mathbf{v})$. In addition, we can compute the local next-state relations from the initial state: $\mathcal{R}_{\varepsilon,k}^{update}(\mathbf{v}_k, x_k, x'_k)$.

The relation $\mathcal{R}_{\varepsilon}^{\gamma}(\mathbf{v})$ for each event ε is constructed off-line in four steps:

1. The variables included in the expressions corresponding to the transition are collected: $V^{\gamma} = Var[E(p,t)] \bigcup Var[E(t,p)] \bigcup Var[G(t)]$.

2. For each variable $v \in V^{\gamma}$ a corresponding variable (level) x_v is created in the MDD. These levels are inserted above the levels of the state variables.

3. The variables are bound for every combination of values permitted by their colour sets in an exhaustive manner.

4. Each possible variable binding of the guard expression is evaluated and every binding that evaluates to *true* is stored in the decision diagram, since with this binding the guard permits the firing of the transition. The colour sets are encoded: each colour in the colour set of a variable in a newly created level is associated with an integer.

An MDD representing $\mathcal{R}_{\varepsilon}^{\gamma}(\mathbf{v})$ is initialised with the above steps, and it contains the possible bindings that make the guard enable the firing of the transition. Since the guard expression does not change during the execution of the model, it is not necessary to update the conjunct represented by the MDD during saturation.

Algorithm 5. CPNConfirm

```
   input  : l : MDD level;
            i : localstate
1  // l:  level of the new state
2  // i:  new local state to be confirmed
3  foreach ε ∈ E : l ∈ supp(ε) do
4      foreach b(v) ∈ B(ε) do
5          I' ← ModelUpd_{ε,l}(i, b(v));
6          R_{ε,l}^{update} ← R_{ε,l}^{update} ∪ {b(v)} × {i} × I';
7  S_l ← S_l ∪ i;
```

Algorithm 6. CPNBuild

```
   input  : l : MDD level
1  // l:  actual level of MDD
2  foreach ε ∈ E : l = Top(ε) do
3      R_ε ← R_ε^γ(v) ∧ ⋀_k R_{ε,k}(v_k, x_k, x'_k);
4      R_ε ← proj_{x,x'}(R_ε(v, x, x'));
```

3.4.5 Correctness of the Algorithm

An efficient decomposition method and encoding were shown in the former section. However, beside the efficiency of the algorithms, we have to overview correctness issues too.

The iteration order of saturation is not modified, so the construction of the next-state relation has to be considered here. We have to investigate if the semantics of CPNs is mapped correctly to the next-state representation. At first, the semantics of CPN is formally defined, and then I show how the implementation reflects the possible behaviour. The algorithm constructs the next-state relation for the event ε of transition t according to the following rule: $\mathcal{R}_\varepsilon = \{(\mathbf{x}, \mathbf{x}') \mid \exists \mathbf{v}\ \mathcal{R}_\varepsilon^\gamma(\mathbf{v}) \wedge \bigwedge_k \mathcal{R}_{\varepsilon,k}^{update}(\mathbf{v}_k, x_k, x'_k)\}$. In the following, the correctness of the mapping is investigated.

Formally, a transition t is enabled with respect to the binding b if the following holds:

- The guard expression is satisfied if $G(t)\langle b\rangle$ evaluates to *true*, which is fulfilled if $\exists \mathbf{v}\ \mathcal{R}_\varepsilon^\gamma(\mathbf{v})$. Satisfying variable assignment of variables $\mathbf{v}$ represents the bindings of the CPN variables by binding b.
- $G(t)\langle b\rangle$ for all possible b is represented by $\mathcal{R}_\varepsilon^\gamma$. As the expression language of the CPN formalism introduced in Section 3.2 is built on top of predicate logic it can be easily transformed to a constraint satisfaction problem ([HVH10]) and solved with the decision diagram representation.
- $E(p,t)\langle b\rangle \leq M(p)$ has to be satisfied for each $p \in P : (p,t) \in A$ and this will be represented by $\mathcal{R}_{\varepsilon,k}^{update}(\mathbf{v}_k, x_k, x'_k)$ for each state variable x encoding the state p.

A transition t changes the state of the CPN with the binding b as follows. Firing from marking M through binding b leading to the marking M' is computed as $M'(p) = M(p) - E(p,t)\langle b\rangle + E(t,p)\langle b\rangle$ for each $p \in P$. The problem is divided into smaller pieces and represented as follows:

- For each arc from a connected place p to the transition t such as $(p,t) \in A$, the next-state is computed $M'(p) = M(p) - E(p,t)\langle b\rangle$, which is encoded as an individual relation $\mathcal{R}_{\varepsilon,k}^{update}(\mathbf{v}_k, x_k, x'_k)$ where x is the state variable corresponding to p and $\mathbf{v}_k$ represents the set of variables V_k such as $V_k = Var[E(p,t)]$.
- For each arc from the transition t to a connected place p such as $(t,p) \in A$, the next-state is computed $M'(p) = M(p) + E(t,p)\langle b\rangle$, which is encoded as an individual relation $\mathcal{R}_{\varepsilon,k}^{update}(\mathbf{v}_k, x_k, x'_k)$ where x is the state variable corresponding to p and $\mathbf{v}_k$ represents the set of variables V_k such as $V_k = Var[E(t,p)]$.
- The construction of each conjunct $\mathcal{R}_{\varepsilon,k}^{update}(\mathbf{v}_k, x_k, x'_k)$ represents the effects of the set of all possible bindings i. e., $B(t)$ to the corresponding places i. e., state variables

The semantics of the CPN transition ε is therefore represented by $\mathcal{R}_\varepsilon$. We have to overview also the implementation of the behaviour. The introduced algorithms use MDDs for representing the relations. As it was discussed in Section 3.4.2, auxiliary variables are introduced in the encoding, for the variables in the arc expressions and guards. During the traversal, Algorithm 5 updates the relations according to the newly discovered local states. The algorithms exhaustively compute the possible local state changes according to the set of all possible bindings. New local states are computed by the function $ModelUpd_{\varepsilon,l}(i, b(v))$, which function computes the new markings being reachable from the local state i by using binding b.

The procedure of Algorithm 6 updates the next-state relation $\mathcal{R}_\varepsilon$. Computing the conjunction of the individual conjuncts ensures that the firing can only happen if all the places are marked properly for the firing i. e., the binding which makes the guard enabled can be satisfied by the marking of the individual places. Finally, the algorithm projects the relation to the state variables (to eliminate the bindings from the representation): $proj_{\mathbf{x},\mathbf{x}'}(\mathcal{R}_\varepsilon)$ computes the existential quantification $\exists \mathbf{v}\ \mathcal{R}_\varepsilon(\mathbf{v}, \mathbf{x}, \mathbf{x}')$.

3.5 Lazy Saturation Algorithm

The algorithm which was introduced for the fine-grained decomposition of the transition relation for CPNs turned out to be efficient (as it will be discussed in Section 3.6). However, the exploration of the local state spaces and local next-state relations is still computationally expensive. In this section, a new saturation algorithm is introduced [5], which uses a more resource-efficient strategy to compose the next-state relations during the state space traversal. The aim is to be able to filter out the unnecessary state changes by delaying the construction of the next state relation. In the following, this new algorithm is called lazy coloured saturation, or *lazy saturation* for short. This section is mainly based on [5].

3.5.1 Performance Issues of Disjunctive-Conjunctive Decomposition for CPN

Despite the fact that the new disjunctive-conjunctive decomposition algorithm for CPNs proved to be efficient, there are some more challenges which came up when the algorithm was applied to our industrial case study and also for synthetic benchmark models.

The coloured saturation algorithm is designed for a general class of CPNs, without restrictions. As a consequence, the algorithm does not have a priori knowledge about the state space, neither about local states, next-states and local next-states. This information will only be revealed during state space exploration. Therefore, the introduced saturation algorithm builds the local state spaces and transition relations on-the-fly, without having additional information that could be used to optimise the traversal and the construction of the next-state relations.

Thus, when a new local state is discovered, both the local state space and the next-state relations need to be updated with regard to the new information. Since these updates are frequent (as all local states and next-state relations must be explored), they impose a big overhead on the algorithm. Moreover, incidental to the greedy transition relation building nature of symbolic methods, the algorithm builds many transition relations that will never be fired, imposed by the restrictions of the state space. The local state space of CPNs might become huge even in the case of decomposing the model into small pieces. Complex data structures and colour types of big domains, various combinations of the coloured tokens will all increase the complexity and overhead of computing the symbolic next-state representation.

Imagine the following example: there is a place p_1 with the colour type of domain size 10. When the algorithm places a token on it through transition t, it will immediately also discover 10 new state transitions representing the situation, that one more additional token arrives. For the 10 different tokens, this will sum up to 100 new transitions. However, if the model will never take 2 tokens on that place, these new transitions will never fire. Note that if transition t removes a token from another place p_0, which can also be marked by the 10 different values of the colour type, the possible number of state changes represented by transition t will be $100 \times 100 = 10000$ more than what saturation will use during the traversal (guard expression on transition t can reduce this number).

In the following sections, a method is introduced to decrease the computation overhead of building the next-state representation of Coloured Petri net models and reducing the size of the next-state representation for the price of introducing a new, smaller relation and a modified iteration order for saturation.

3.5.2 Overview of the Approach

Symbolic algorithms encode the possible states and state changes in decision diagrams. The data structures are continuously updated as new states are discovered and unexplored transitions appear. The data manipulation based on decision diagrams can be expensive, so various techniques are utilised to decrease the computational costs. The disjunctive-conjunctive partitioning algorithm decomposes the next-state relation, and saturation benefits from the efficient manipulation of the smaller parts. During the iteration these subrelations are updated according to the recently discovered substates: every time a new local state is discovered, all possible local state transitions are computed and added to the corresponding next-state relation. This greedy strategy constructs the next-state relations in one complex step, where each possible local and global state change is explored no matter if it is reachable in the state space, or not. However, this can be wasteful as the growing number of local states can easily lead to many unreachable combinations of them. This is especially true for CPNs, as they provide a compact representation of even complex models. In such models, there can be many state transitions that are reachable locally, but the algorithm will never reach a state where they become enabled on the global, Petri net level. Since these infeasible local state transitions have been added to the local next-state relations, the decomposed symbolic representation becomes bigger than necessary.

In this section, I introduce a new approach to decrease the size of the transition relation representation of complex CPN models. The main idea of the approach is to build the transition relation lazily in two phases. The first phase is the discovery phase, where the algorithm only registers the potentially reachable global states, but the algorithm constructs the next-state relation from a given state only if the given state becomes globally reachable. This way the algorithm tries to avoid the construction of the next-state relation for globally unreachable states. This temporal decomposition of the construction of the next-state relation can be advantageous for CPN models.

The new algorithm aims to filter out as many infeasible transition relations as possible. For this purpose, a new relation $\mathcal{ER}$ is introduced that only stores the states from which state transitions are enabled. In other words, this relation contains only "from" states ($\mathbf{x}$) and the "to" states ($\mathbf{x}'$) are not stored – contrary to the next-state relation. This lets the building of the next-state relations be delayed until the algorithm can exactly decide which relation should be updated with the new information. First, only the $\mathcal{ER}$ relation is built, and the state transition is stored in the next-state relation only when the relation becomes globally enabled. This way the next-state relation will contain less globally infeasible state transitions: its size will be reduced and also the state space traversal will be more efficient. The motivation of our work is based on the observation that the size of the $\mathcal{ER}$ relation is

always smaller than the size of the $\mathcal{R}$ relations: using this smaller $\mathcal{ER}$ relation to postpone or to skip the updating of the $\mathcal{R}$ relations is a good pay-off regarding the performance of the algorithm.

3.5.3 Iteration of Lazy Saturation

Saturation builds the next-state relation in an eager manner which has to be modified to work lazily. The main functionalities of the saturation are not changed as it is depicted on Algorithm 7. The cache manipulation and decision diagram specific operations are omitted for brevity, but the interested reader can find them in [CMS03]. The changes compared to former approaches are marked with an asterisk ($*$), the rest of the algorithms are based on the formerly introduced saturation of Algorithm 3.4.2 and [CMS03]. The lazy saturation algorithm starts the state space exploration from the initial states represented by a decision diagram and initialises $\mathcal{ER}$ and $\mathcal{R}$ representations. Function Algorithm 7 saturates the nodes of the state space representation and Algorithm 8 computes the steps of the traversal. These functions build the MDD of the state space by firing all enabled events in a recursive, exhaustive manner.

The main difference compared to the traditional saturation iteration strategy comes at this point. As the next-state relations are built lazily, the algorithm has to check before the symbolic next-state computation if a step has become enabled which is not in the next-state relation. So, before each step in the state space, the new lazy saturation algorithm checks the next-state relations and the $\mathcal{ER}$ relations: function $UpdateRelation$ of Algorithm 13 extracts the necessary information from the state space representation and from the relation $\mathcal{ER}$ to update the conjuncts of the next-state relation, if needed. The function $\mathcal{ER}Build$ updates the next-state function of event ε by reconstructing the complete relation from the conjuncts.

If a new state is discovered, the function $\mathcal{ER}Confirm$ of Algorithm 9 updates the relation $\mathcal{ER}$. This function is explained in detail in Section 3.5.4. The next-state relation $\mathcal{R}$ is updated by the function $UpdateRelation$ at this point of saturation. Its operation is described in Section 3.5.5. In the later sections, the details of the steps are discussed.

3.5.4 Computing and Using $\mathcal{ER}$

The goal of using an additional relation is to decrease the size and the computational cost of the next-state relation. For this purpose, the new saturation algorithm constructs a simple set to store those state configurations which can contribute to event firings. Constructing this representation delays the construction of the next-state representation: the new saturation algorithm will only add a step into the next-state representation if the state space traversal needs to fire it.

The construction of relation $\mathcal{ER}$ is similar to the construction of the next-state relations and also similar to the construction of relation $\mathcal{R}^{enable}$ of Section 2.3.2. The lazy saturation algorithm exploits the locality also for the construction of $\mathcal{ER}$: the disjunctive-conjunctive decomposition is also applied to the $\mathcal{ER}$ relation. The algorithm creates a separate $\mathcal{ER}_\varepsilon$ relation for each event ε. In order to efficiently manipulate the relation, the algorithm partitions each $\mathcal{ER}_\varepsilon$ relation into smaller parts, and stores them separately according to the following rule: $\mathcal{ER} = \bigvee_{\varepsilon\in\mathcal{E}} \mathcal{ER}_\varepsilon$ and $\mathcal{ER}_\varepsilon = \bigwedge_k \mathcal{ER}_{\varepsilon,k}$. This way the new lazy algorithm can exploit event-locality and the other advantages of disjunctive-conjunctive decomposition.

Contrary to the eager construction of the next-state representation of the traditional algorithm, the new algorithm builds the $\mathcal{ER}$ relation during the iteration primarily. The algorithm discovers the new states from which an event can be fired. The target states of the event firing are not traversed in this phase of the iteration. So the role of the relation $\mathcal{ER}$ is simplified compared to the next-

Algorithm 7. LazySaturate

```
   input  : s_k : node
1  // s_k:  node to be saturated,
   output : node
2  if s_k = 1 then
3      return 1;
4  Return result from cache if possible;
5  k ← Level(s_k);        // retrieve the actual
     level of the MDD
6  t_k ← new Node_k;
7  foreach i ∈ S_k : s_k[i] ≠ 0 do
8      t_k[i] ← Saturate(s_k[i]);
9  repeat
10     foreach ε ∈ E : k ∈ Top(ε) do
*11        UpdateRelation(ε, s_k, ER_ε, R_ε);
*12        LazyCPNBuild(ε);
13         R_ε ← N_ε as decision diagram;
14         foreach s_k[i] ≠ 0 ∧ R_ε[i][i'] ≠ 0 do
15             t_k[i'] ← t_k[i'] ∪ LazyRelProd(t_k[i], R_ε[i][i']);
16             if i' ∉ S_k then
17                 ERConfirm(k, i')
18         ERBuild(k);
19 until t_k unchanged;
20 t_k ← PutInUniqueTable(t_k);
21 Put inputs and results in cache;
22 return t_k;
```

Algorithm 8. LazyRelProd

```
   input  : s_k, R : node
1  // s_k:  node to be saturated,
2  // R:  next-state representation node
   output : node
3  if R = 1 then
4      return s_k;
5  Return result from cache if possible;
6  k ← Level(s_k);      // retrieve the actual
     level of the MDD
7  t_k ← new Node_k;
8  foreach s_k[i] ≠ 0 ∧ R[i][i'] ≠ 0 do
9      t_k[i'] ← t_k[i'] ∪ LazyRelProd(s_k[i], R[i][i']);
10     if i' ∉ S_k then
11         ERConfirm(k, i')
12 t_k ← PutInUniqueTable(Saturate(t_k));
13 Put inputs and results in cache;
14 return t_k;
```

Algorithm 9. $\mathcal{ER}$Confirm

```
   input  : l : MDD level;
            i : localstate
1  // l:  level of the new state
2  // i:  new local state to be confirmed
3  foreach ε ∈ E : l ∈ supp(ε) do
4      foreach b(v) ∈ B(ε) do
5          if ModelUpd_{ε,l}(i, b(v)) ≠ ∅ then
6              ER_{ε,l} ← ER_{ε,l} ∪ {b(v)} × {i};
7  S_l ← S_l ∪ i;
```

Algorithm 10. $\mathcal{ER}$Build

```
   input  : l : MDD level
1  // l:  actual level of MDD
2  R_l ← ∅;
3  foreach ε ∈ E : l = Top(ε) do
4      ER_ε ← R_ε^γ(v) ∧ ⋀_{k∈supp(ε)} ER_{ε,k}(v_k, x_k);
5      ER_ε ← proj_x(ER_ε(v_k, x_k));
```

Algorithm 11. LazyCPNConfirm

```
   input  : ε : event; l : MDD level;
            i : localstate
1  // l:  level of the new state
2  // i:  new local state to be confirmed
3  foreach ε ∈ E : l ∈ supp(ε) do
4      foreach b(v) ∈ B(ε) do
5          if ModelUpd_{ε,l}(i, b(v)) ≠ ∅ then
6              E R_{ε,l} ← E R_{ε,l} ∪ {b(v)} × {i};
7  S_l ← S_l ∪ i;
```

Algorithm 12. LazyCPNBuild

```
   input  : ε : event
1  // ε:  actual event to be updated
2  R_ε ← R_ε^γ(v) ∧ ⋀_k R_{ε,k}(v_k, x_k, x'_k);
3  R_ε ← proj_{x,x'}(R_ε(v, x, x'));
```

Algorithm 13. UpdateRelation

```
   input  : ε : event; s_k, ER, R : node
1  // ε:  fired event,
2  // s_k:  traversed node,
3  // R:  next-state representation node
4  // ER:  ER representation node
5  k ← Level(s_k);      // retrieve the actual
     level of the MDD
6  Return result from cache if possible;
7  if k = 0 then
8      return;
9  foreach i ∈ S_k : s_k[i] ≠ 0 ∧ ER[i] ≠ 0 do
10     if R[i] is unknown then
11         LazyCPNConfirm(ε, k, i)
12     else
13         foreach j ∈ S_k : s_k[i] ≠ 0 ∧ R[i][j] ≠ 0 do
14             UpdateRelation(ε, s_k[i], ER[i], R[i][j]);
15 Put inputs and results in cache;
```

state relations: this fact appears in the computational complexity and space requirements of the new relation. Constructing and storing relation $\mathcal{ER}$ is much cheaper than the next-state relations.

Set of states represented by $\mathcal{ER}$. The relation $\mathcal{ER}$ aims to represent and encode those states from which the firing of an event (in the context of CPN, a transition) is enabled, formally: $\mathbf{x} \in \mathcal{ER}$ iff $\exists \varepsilon \in \mathcal{E} \ : \ \mathcal{N}_\varepsilon(\mathbf{x}) \neq \emptyset$. According to this rule, disjunctive partitioning of $\mathcal{ER} = \bigvee_{\varepsilon \in \mathcal{E}} \mathcal{ER}_\varepsilon$ can be used to decomposed both the construction of the relation and also the usage during the state space traversal.

Connection to $\mathcal{R}$. The relation $\mathcal{ER}$ can be derived from the next-state representation as follows: $\mathbf{x} \in \mathcal{ER}_\varepsilon$ iff $\exists \mathbf{x}' \ : \ \mathcal{R}_\varepsilon(\mathbf{x}, \mathbf{x}')$. Effectively, this can be computed by the formerly presented projection function: $\mathcal{ER}_\varepsilon = proj_{\mathbf{x}}(\mathcal{R}_\varepsilon)$. However, it is not practical to compute relation $\mathcal{R}_\varepsilon$ first and derive $\mathcal{ER}_\varepsilon$ from it. Instead, I show a method to compute $\mathcal{ER}$ directly during the state space traversal and use it to reduce the size of $\mathcal{R}$.

The relation $\mathcal{ER}_\varepsilon$ is the "simplified" version of the relation $\mathcal{R}_\varepsilon$ used by the traditional saturation algorithm.

Efficient computation of $\mathcal{ER}$. The efficient computation of relation $\mathcal{ER}$ is accomplished by functions $\mathcal{ER}Confirm$ and $\mathcal{ER}Build$ of Algorithm 9 and Algorithm 10.

The pseudocode of the function $\mathcal{ER}Confirm$ updates the individual conjuncts of the $\mathcal{ER}$ relation. The input parameters of the function represent the new (recently discovered) local state (i) at the level l of the decision diagram. $\mathcal{ER}Confirm$ function examines if there is a binding of the events which can be satisfied by state i and updates the conjuncts with this information.

After a conjunct of $\mathcal{ER}_\varepsilon$ was updated, the whole $\mathcal{ER}_\varepsilon$ relation has to be rebuilt by computing symbolically: $\mathcal{ER}_\varepsilon = \mathcal{ER}_\varepsilon^\gamma \wedge \bigwedge_k \mathcal{ER}_{\varepsilon,k}$. This is carried out by the function $\mathcal{ER}Build$. As the enabling relation $\mathcal{ER}_\varepsilon^\gamma$ is not much smaller than $\mathcal{R}_\varepsilon^\gamma$, the later is used in the algorithm to reduce the computational cost and avoid redundant computations. The algorithm also projects the relation to the state

variables at line 5 as the relation has to represent if a state configuration is enabled, but the exact bindings are not used.

Optimizations. The relation $\mathcal{ER}$ contains unnecessary information to decide if an event is enabled or not in a certain state: places (and their marking) connected to the outgoing edges do not influence the enabledness of the transition. This means that the state variables corresponding to the places of the outgoing edges can be left out from the relation without losing information and this leads to a decreased number of conjuncts and reduced computational costs. Formally, the relation $\mathcal{ER}$ for the event of transition t will be constructed from conjuncts $\mathcal{ER}_k(\mathbf{v}_k, x_{in})$ where x_{in} encodes the state of p_{in} and $E(p_{in}, t) \in A$. Let $\mathbf{x}_{in}$ represent the state variables corresponding to the input places of transition t and $\mathbf{x}_{out}$ represent the state variables corresponding to the output places of transition t, then the simplified $\mathcal{ER}_\varepsilon$ relation can be expressed as $\exists \mathbf{x}_{out}\ \mathcal{ER}_\varepsilon(\mathbf{x}_{in}, \mathbf{x}_{out})$. This is computed by the projection operator as $proj_{\mathbf{x}_{out}}(\mathcal{ER}_\varepsilon)$. This optimisation is omitted from the pseudo code for the sake of simplicity.

3.5.5 Updating the Next-State Relation

The lazy saturation algorithm updates at first the relation $\mathcal{ER}$, and updates relation $\mathcal{R}$ only if necessary. Now, I will explain how the algorithm detects the situation when relation $\mathcal{R}$ has to be updated. $\mathcal{R}$ is updated if a state is reached, which is included only in relation $\mathcal{ER}$. This means that a transition firing is enabled in that state, but this transition firing is not yet included in $\mathcal{R}$, so the algorithm has to update $\mathcal{R}$ with the newly enabled possible firings. The traversal collects the possible states from which the events are enabled, and when the firing of a new state transition is triggered, the next-state relation is updated with the new information.

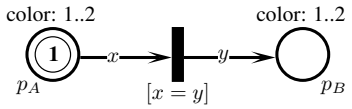
The next-state relations are updated by the function $UpdateRelation$ of Algorithm 13. This function is called from the function $LazySaturate$ at line 11. $LazySaturate$ and $LazyRelProd$ functions update first the relation $\mathcal{ER}$ during the traversal. During the state space traversal, before firing a new event, function $UpdateRelation$ is called. $UpdateRelation$ traverses the state space representation together with the relation $\mathcal{ER}$ and $\mathcal{R}$, and checks if there is any reachable state which is represented in $\mathcal{ER}$ but not represented in relation $\mathcal{R}$. Those states which are only included in $\mathcal{ER}$ are the starting state of a next-state relation to be included also in $\mathcal{R}$.

The function $UpdateRelation$ recursively computes if the given event is enabled, and updates the next-state relation if needed. It traverses all event firings recursively from the $\mathcal{ER}_\varepsilon$ and $\mathcal{R}_\varepsilon$ relations, and the MDD denoted by s_k that encodes the state space. During this traversal, the algorithm decides whether a state transition is enabled or not. If the algorithm finds at (line 10) a state from which a step through event ε is enabled, but it is not included in relation $\mathcal{R}_\varepsilon$ (i. e., the next-state is unknown), calling function $LazyCPNConfirm$ will update the corresponding conjuncts of the next-state relation. After updating the conjuncts of event ε, function $LazyCPNBuild$ will rebuild the corresponding next-state relation $\mathcal{R}_\varepsilon$. This function updates the $\mathcal{R}_\varepsilon$ relation by calculating it from the updated conjuncts as follows $(\mathbf{x}, \mathbf{x}') \in \mathcal{R}_\varepsilon$ iff and only iff $\mathbf{x} \in \mathcal{S}_{reach}$, and all the other rules remain the same as introduced in Section 3.4.

3.5.6 Operation of Lazy Saturation

The working of the lazy saturation algorithm is illustrated now with an example. The example Coloured Petri net model is shown in Figure 3.2a: the model consists of two places and a transition. Both places have the same colour type with two values: 1 and 2. Initially the place p_A is marked

with a token valued 1, and the place p_B is empty. The decomposition creates two submodels, one for place p_A and one for place p_B. The encoding of the local states is shown in Figure 3.2b. Based on the table, the initial (local) state of place p_A is 1 and the initial state of place p_B is 0. The decomposition of the relation $\mathcal{R}$ (and of the relation $\mathcal{ER}$) conforms to the decomposition of the state space, i. e., there are two update conjuncts, $\mathcal{R}_A^{update}$ and $\mathcal{R}_B^{update}$.



(a) Example CPN model

Local state	Meaning
0	The place is empty.
1	The place is marked with a token valued 1.
2	The place is marked with a token valued 2.
11	The place is marked with two tokens, both valued 1.
12	The place is marked with two tokens, the value of the first is 1, the value of the second is 2.
etc.	

(b) Encoding of the local states

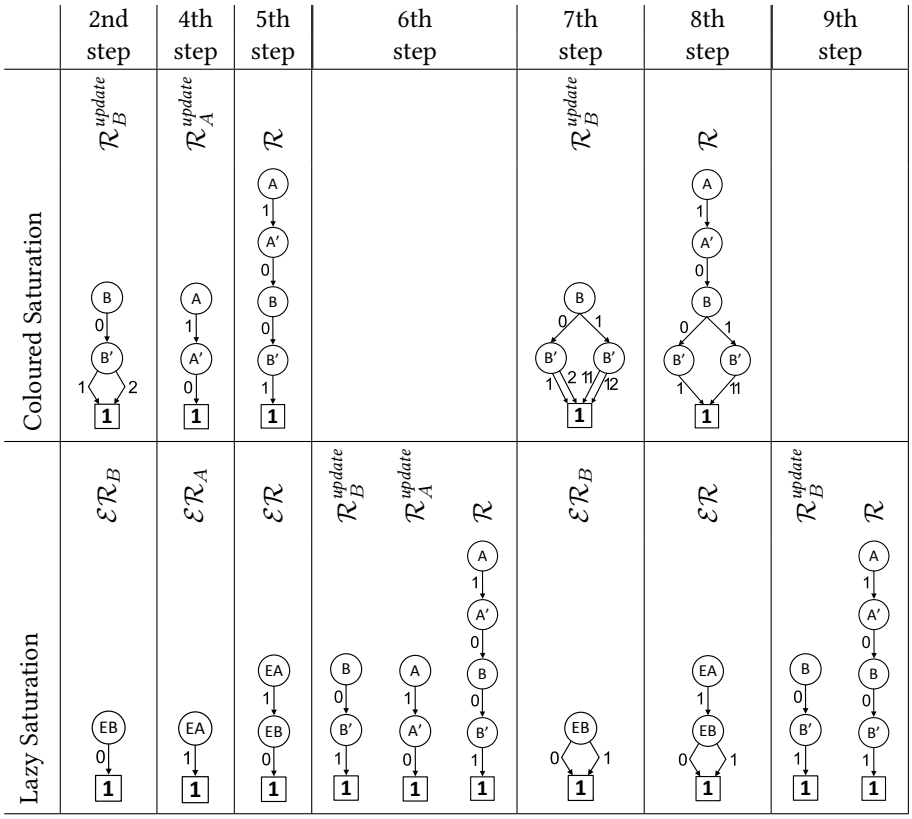
Figure 3.2: Example to illustrate the operation of lazy saturation

The MDDs created during the event handling of saturation are shown in Table 3.1. The content of the first row belongs to the coloured saturation algorithm of Section 3.4, while the second row belongs to the new lazy algorithm of Section 3.5. The decision diagram levels (variables) corresponding to the variables of the guard and arc expressions are omitted from the MDDs for brevity (both approaches build the same guard relation and use the same auxiliary variables).

The execution steps of the coloured saturation algorithm and the lazy algorithm for the example are the following:

1. State space generation is started: relation $\mathcal{R}^\gamma$ is initialized and its content is calculated off-line. (This relation is not shown in Table 3.1.)
2. Coloured saturation explores and collects all possible state changes into the $\mathcal{R}_B^{update}$ relation by calling $CPNConfirm(1, 0)$. Locally there are two new reachable states depending on the assignment of variable y. Lazy saturation examines only whether the transition is fireable from state 0, and procedure $\mathcal{ER}Confirm$ collects this enabled state into the relation $\mathcal{ER}_B$.
3. Saturation cannot make any steps, transition firings are not enabled at this level of the decision diagram as the Top value of the event is two.
4. Saturation continues the iteration and jumps to the second level of the decision diagram. CPN saturation calls $CPNConfirm(2, 1)$. Coloured saturation creates updates conjunct $\mathcal{R}_A^{update}$ while lazy saturation creates the relation $\mathcal{ER}_A$ as depicted on the figure.
5. Function $Build$ called by saturation builds up the relation $\mathcal{R}$ from the conjuncts. At the same step, lazy saturation calculates only the $\mathcal{ER}$ relation by calling the $\mathcal{ER}Build$ function. The (not represented) conjunct $\mathcal{R}^\gamma$ prevents the next-state relation from storing the $(A, A', B, B') = (1, 0, 0, 2)$ global state change (i. e., the $(1, 0) \rightarrow (0, 2)$ state transition, which is evidently impossible).
6. Saturation continues the iteration. Coloured saturation fires the global state change $(1, 0, 0, 1)$ i. e., the $(1, 0) \rightarrow (0, 1)$ state transition and updates the state space representation MDD. This is the point where the lazy saturation algorithm calls the procedure $UpdateRelation$ and realises that relation $\mathcal{R}$ has to be updated (because it is still empty). After updating the next-state relation, it makes the same steps as coloured saturation.
7. The newly reached local states must be confirmed. $Confirm(2, 0)$ (i. e., confirming the local state 0 at the level of place p_A) does nothing because the transition cannot fire when place p_A

Table 3.1: Data structures (MDDs) of coloured saturation and lazy saturation



is empty. However, the transition is enabled locally, if place p_B contains a token, so the relation $\mathcal{R}_B^{update}$ is updated by coloured saturation, and relation $\mathcal{ER}_B$ is also updated by lazy saturation.

8. The algorithm updates the relations $\mathcal{R}$ and $\mathcal{ER}$ by using the updated conjuncts, respectively.
9. Similarly to the 5th step of lazy saturation, $\mathcal{R}$ has to be extended with the enabled state changes, before saturation takes a step in the state space. However, there is no newly enabled state changes, so lazy saturation does not extend the relation with the change represented by $(A, A', B, B') = (1, 0, 1, 11)$, as the transition $(1, 1) \rightarrow (0, 11)$ is not possible with the given initial marking.
10. There is no newly enabled relation for neither the lazy nor the coloured saturation algorithm, so the procedure is finished. The next-state relation of the lazy saturation algorithm contains less next-states.

The example shows the main differences between coloured saturation of Section 3.4 and lazy saturation of Section 3.5. Lazy saturation delays the building of the next state relation resulting smaller next-state relations. Building relation $\mathcal{ER}$ is cheaper compared to the construction of $\mathcal{R}$ which makes the verification of CPN models with complex guard relations more efficient.

3.5.7 Correctness of Lazy Saturation

The correctness of the disjunctive-conjunctive decomposition was proven in Section 3.4.5. Now I show that lazy saturation will do the same steps during the state space exploration. The iteration order of saturation is slightly modified in lazy saturation: next-state computation only updates relation $\mathcal{ER}$. However, before each next-state computation, the algorithm updates the relation $\mathcal{R}$ at line 11 of

Algorithm 7 (*UpdateRelation* of Algorithm 13). As far as function *UpdateRelation* updates relation $\mathcal{R}$ to contain all states which will be used by function *LazyRelProd* (note that *UpdateRelation* iterates through the states similarly as the next-state computation), it is ensured that the algorithm will not miss any states.

There are two more conditions, which have to be ensured:

- $\mathcal{ER}$ contains all the enabled states. This is ensured as it represents a projection of the next-state function at each step.
- $\mathcal{R}$ contains all the states, which can be fired. This is ensured because function *UpdateRelation* uses the same traversal as the next-state computation (recursive traversal of all possible combinations of $\mathcal{ER}$ and the state space), so function *UpdateRelation* will traverse all possible next-states for the given state space and $\mathcal{ER}$.

Sometimes, due to the projection, $\mathcal{ER}$ will lose information and might not constrain the next-states as much as $\mathcal{R}$ would do. However, this is not a problem as relation $\mathcal{ER}$ can be permissive as the next-state computations are done according to relation $\mathcal{R}$.

3.6 Industrial Case Study

In this section, I introduce the industrial case study in which I evaluated the developed algorithms. The industrial case study was initially introduced in [16].

3.6.1 The Modelled Industrial System

The subject of our research is a safety function, designed to initiate an emergency prevention activities in the occurrence of the so-called *PRISE event*. This safety function is used in the Paks Nuclear Power Plant (Paks NPP) located in Hungary. The Paks NPP operates four VVER-440/213 type pressurised water reactor (PWR) units with a total nominal (electrical) power of approx. 2 GW. Nuclear power plants are highly safety-critical and complex systems, where the correct operation of the safety procedures is of great importance. The plant protection systems must satisfy high safety requirements and minimise spurious forced outages. Therefore, formal modelling and verification methods need to be applied to prove the correctness and completeness of the PRISE safety function.

The *PRImary-to-SEcondary leaking* (PRISE) event is one of the major faults in a reactor unit, resulting due to a non-compensable leaking of parts in the primary circuit. The PRISE event occurs when there is a rupture or other leakage within the steam generator (SG) vessel primary tubing, affecting either a few (3–10) tubes or their collector that contain the high-pressure activated liquid of the primary circuit.

The PRISE event is the VVER-440/213 analogue of the well-investigated Steam Generator Tube Rupture (SGTR) event (see e. g., [IS94]) in other types of pressurized water reactors.

In the unlikely case of a PRISE event, the safety procedures first initiate the emergency shutdown (scram, trip) of the reactor, and then isolate the faulty steam generator. However, there would still be a possibility to release some of the contaminated water to the environment if the event would not be handled properly. In order to prevent this and to increase the safety of the plant, a safety valve for draining the contaminated water into the containment has been added to each steam generator, and a new safety function, the *PRISE safety function* has been developed to control its operation.

3.6.2 The PRISE Safety Function

The technological and I&C system experts of the Paks NPP have designed a timed logical scheme, the basis of the PRISE *safety function*, in a heuristic way. The logical scheme was specified as a Functional Block Diagram (FBD) representation (a formalism similar to the one defined in the IEC 61131-3 standard). The PRISE safety function FBD is shown in Figure 3.3. The description of the inputs and outputs of the PRISE safety function are included in Table 3.2.

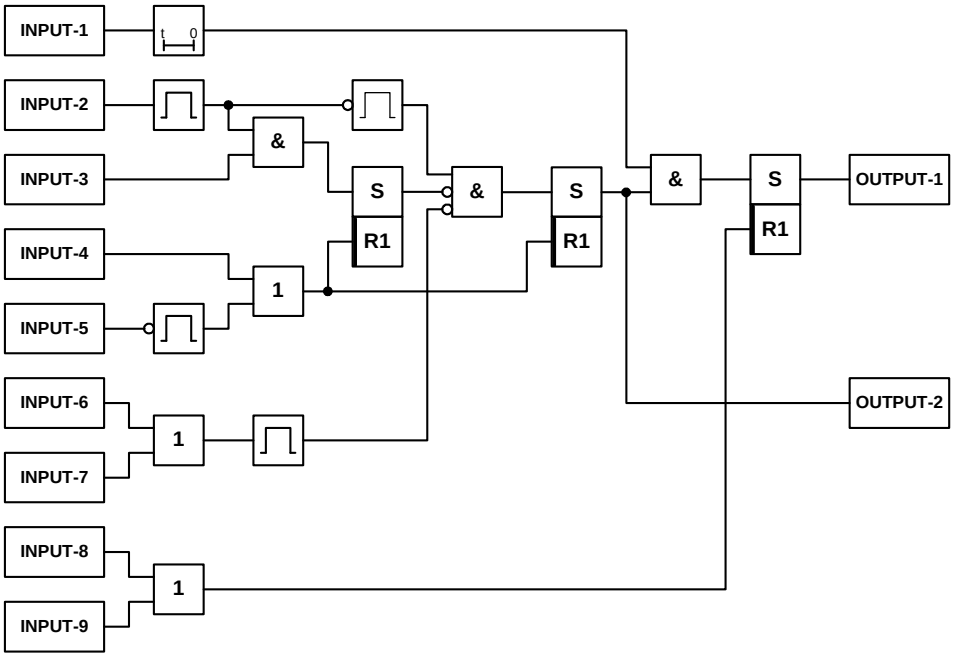


Figure 3.3: Functional block diagram of the PRISE safety function [Ném+09]

The purpose of the PRISE safety function is to initiate the draining of the steam generator *if and only if a PRISE event occurs*. This implies preventing the activation of the safety valve, when a non-PRISE fault event (causing similar symptoms but without a classified PRISE event) occurs, i. e., the PRISE safety function must be *selective*. Moreover, when the reactor unit is either being started up or shut down, thus it is not in the normal operating regime, the PRISE safety function is designed not to be active. In these circumstances the operators can activate the draining valve manually, should a need arise.

The designed safety procedure initiates the draining (OUTPUT-1) when a critical decrease in the primary pressure (INPUT-2) is followed (after a specified time delay) by the increase of the steam generator level (INPUT-1) that lasts for a certain time interval. However, the draining is initiated only if the containment pressure keeps its nominal value (INPUT-3), i. e., it is not increasing due to another, non-PRISE fault causing an inflow of the primary water into the containment. The minimum time interval constraint for INPUT-1 to hold its value prevents the incorrect initiation of draining by an unreliable water level sensor measurement showing temporarily a spuriously high value (caused by the solid scale content of the secondary water).

The INPUT-4 and INPUT-9 input conditions inhibit the operation in a startup or shutdown situation. INPUT-5 resets the operation of the PRISE safety procedure in case the reactor is being shut down. INPUT-6 and INPUT-7 prevent the erroneous draining of the containment after the isolation of a steam generator caused by a non-PRISE fault. INPUT-8 indicates the situation when the steam

Table 3.2: PRISE safety procedure I/O description

Name	Description	Function
INPUT-1	SG level high	Steam generator water level is increasing (due to closure of the turbine)
INPUT-2	Primary pressure decreasing	The pressure of the primary water is decreasing (due to PRISE or other leakage)
INPUT-3	Containment pressure is normal	The pressure of the containment is *not* increasing (no primary water inflow caused by a non-PRISE fault)
INPUT-4	Primary temperature below nominal	Technical condition signifying that the reactor is in startup/ shutdown regime
INPUT-5	Control rods fully down	Technical condition used to reset the operation of the PRISE safety procedure
INPUT-6	SG deltaP	Technical conditions used to avoid the erroneous draining of
INPUT-7	SG RAP 1/2	the secondary water after isolation of the steam generator
INPUT-8	SG inhibition	Technical condition used to indicate the SG inhibited state
INPUT-9	Primary pressure low	Technical condition signifying that the reactor is in startup/ shutdown regime
OUTPUT-1	SG is inhermetical	Primary output, activates the secondary water drain
OUTPUT-2	ACTIVE	Auxiliary output used in control operations

generator was manually isolated due to a failure indication. The primary OUTPUT-1 of the procedure signals the presence of a PRISE event. Note that the auxiliary OUTPUT-2 signal indicates the presence of all but one of the symptoms of the PRISE situation.

3.6.3 Coloured Petri Net Model of the PRISE Safety Function

I have created a hierarchical Coloured Petri net model of the PRISE safety function. Figure 3.4 shows the high-level main net of our CPN model. The grey circles are the inputs and outputs of the PRISE logic. The larger labelled rectangles are substitution transitions that denote subnets of the corresponding function blocks. The smaller net elements are simple places and transitions that are only needed for connecting the subnets. This main net integrates and connects the separately developed and validated lower-level CPN subnets of the different functional blocks. The transformation of the Functional Block Diagram (see Figure 3.3) was straightforward and simple to validate since the structure of the FBD graph and the corresponding CPN graph are isomorphic.

The run-time environment is a safety-critical, highly dependable digital distributed control system (DCS), which runs at an explicit 50-millisecond long *scan cycle*. During each scan cycle, the controller first samples its inputs, then evaluates all of its functional diagram pages starting from the blocks connected to the inputs and following the flow of data until they reach the outputs, computes its new internal state, sets the outputs, and in the remaining time performs self-tests. This behaviour is reflected by the CPN model the following way: the propagation of the tokens in the net represents the flow of data in the functional diagram. The CPN model has a feedback loop that puts a single coloured token simultaneously into each input place at the beginning of a scan cycle. The colour of the input tokens carries the input data value. These tokens initiate the execution of the subnets modelling the function blocks. When every subnet has been executed, a single coloured token is generated into each output place. The feedback loop takes away every generated token from the outputs, and the scan cycle ends.

An example CPN subnet —modelling the operation of a functional block, namely the *Delay mod-*

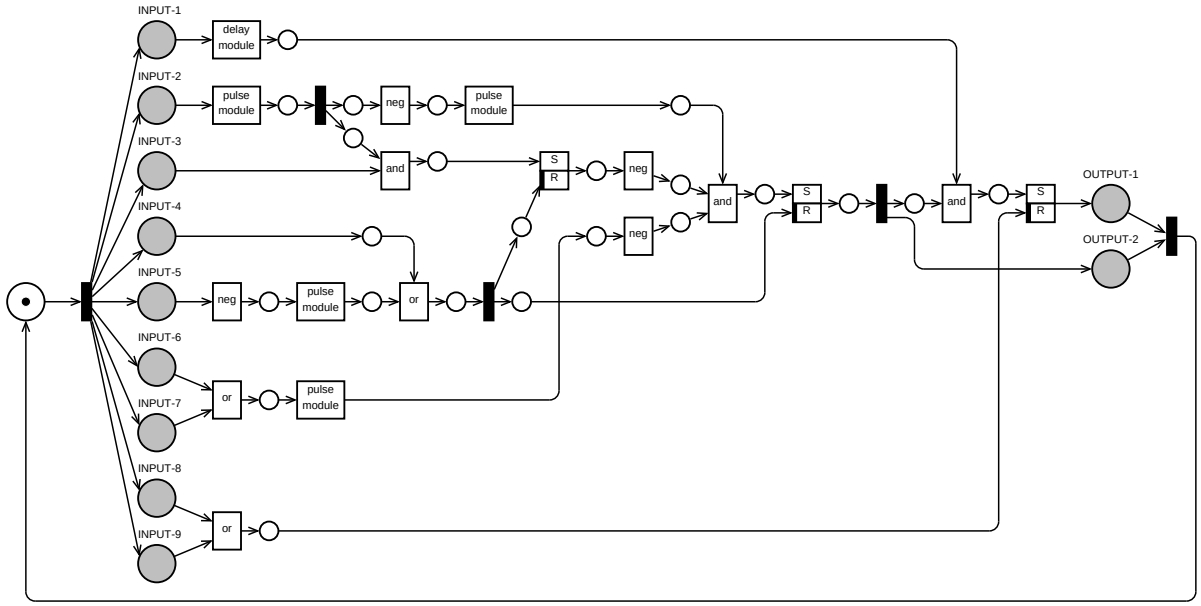
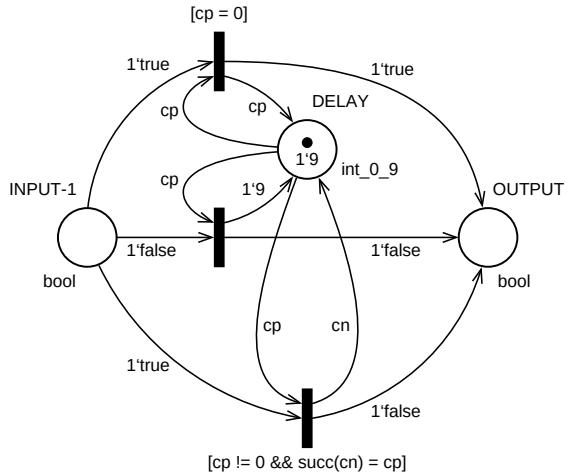
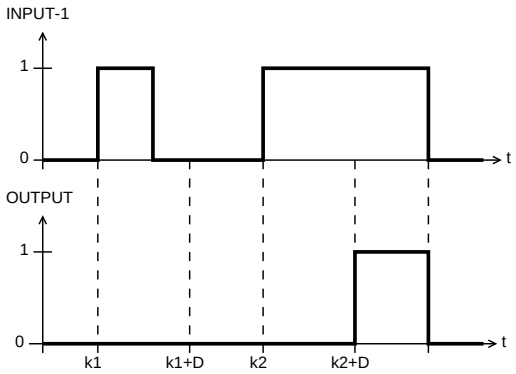


Figure 3.4: The Coloured Petri net model of the PRISE safety procedure [16]

ule— is shown in Figure 3.5a. The functionality of the Delay module is given by a time diagram in Figure 3.5b. The purpose of the module (as its name implies) is to delay a rising edge pulse for a predefined D number of cycles. When the module detects a rising edge, it starts a counter. If the pulse is active (the input remains 1) for at least D number of cycles, the Delay module will "let the pulse pass", that is it sets its output to 1 (the `true` Boolean value). The output will remain 1 as long as the input is active. When a falling edge is detected, the module resets itself to its default inactive state.



(a) The CPN subnet model of the Delay module



(b) Time diagram of the operation

Figure 3.5: Delay module: model and operation

The operation of the CPN subnet model of the Delay module (see Figure 3.5a) is easy to follow. The model has two *port places* (the INPUT-1 port, and the OUTPUT port) that represent the connections of the Delay module. The DELAY place stores the value of the delay counter. Its colour set is

`int_0_9`, a subset of integers: $\{0, \ldots, 9\}$. Its initial marking has one token whose colour equals the required delay time, given as D cycles (in the example $D = 9$). The three black rectangles are the transitions that realise the three main phases of the operation. The expressions written in brackets next to the transitions are their *guards*. A guard is a boolean expression that prohibits the firing of the corresponding transition unless it evaluates to `true`. The three main phases of the operation and their transitions are as follows:

1. The *lower transition* detects the rising edge (a `true` value is in the input and the delay counter has not yet reached zero), starts the delay counter, and continues counting down in the subsequent cycles. The guard prescribes the previous value of the counter to be the successor of the next value, thus implementing the counting down process. The output remains inactive in this phase.
2. The *upper transition* will fire whenever the delay counter has run out (reached zero), and the input is active. The transition puts a token with value `true` in the output port place, therefore the output will remain active as long as the input is active.
3. The *middle transition* detects the falling edge of the pulse, resets the value of the delay counter, and deactivates the output.

3.6.4 Verification of the PRISE Safety Function

We aimed to prove that the PRISE safety function initiates the draining *always if a PRISE event occurs* in every normal operation regime coupled with a *fault in the SG level sensor* that is highly unreliable; and *never if a PRISE event does not occur* even if severe faults causing similar symptoms happen. In addition, it is also important to prove that the PRISE detection logic is free from deadlocks as they represent dangerous situations. The required selective detection of the PRISE event and the heuristic design process of the safety logic made it necessary to perform a rigorous formal verification of the PRISE safety procedure.

3.6.4.1 Formalization of the Requirements

I could translate the above requirements into the following verification goals:

- *Liveness requirement:* the secondary water draining activity is always activated when a real PRISE accident has occurred (no actuation masking).
- *Safety requirement:* the draining activity is not activated if not a real PRISE accident has occurred (no erroneous actuation).
- *Deadlock freedom:* No deadlock situation can arise for any combination and sequence of input signals.

I used branching-time temporal logic based model checking to prove the requirements. For complexity reasons, we chose CTL temporal logic, as it provides an expressive formalism with efficient decision procedures.

- First, deadlock freedom of the system is checked. Informally this means that in every state there exists at least one reachable successor state. The equivalent CTL temporal logic formula is the following: AG(EX(*true*)).
- I also checked if the model is *reversible*, that is from every state the initial state can be reached. I expressed this property with the following CTL formula: AG(EF([*init*])). This property ensures that the safety function can be made ready to fulfil its goal in all circumstances.

- The following formula expresses the safety requirement: $\neg \mathsf{E}(\neg\,[\mathit{PRISE\text{-}event}]\ \mathsf{U}\ [\mathit{actuation}])$. I used an indirect proof to prove the safety requirement and ran the model checker with the inverse CTL formula: $\mathsf{E}(\neg\,[\mathit{PRISE\text{-}event}]\ \mathsf{U}\ [\mathit{actuation}])$. This formula is satisfied only if the draining activity is activated without a PRISE event.

- The liveness requirement was also easier to prove by using indirection. I formalised the inverse requirement as the following CTL formula: $\mathsf{EF}([\mathit{PRISE\text{-}event}] \wedge \mathsf{EG}(\neg\,[\mathit{actuation}] \wedge \neg\,[\mathit{reset\text{-}event}]))$. Informally, we are searching for strongly connected components in the state space that contain no *actuation* and *reset-event*, but contain a *PRISE-event*.

3.6.4.2 Evaluation of the Temporal Properties

The next step of the verification was to explore and store the state space of the CPN model of the safety function, using the new disjunctive-conjunctive decomposition algorithm and the lazy saturation algorithm. After obtaining the complete state space we could evaluate the four CTL expressions introduced in the previous section. For state space traversal and temporal logic-based model checking we developed our own experimental implementation of our algorithms written in the C# programming language. We used the following configuration for our measurements: Intel L5420 2.5 GHz processor, 8 GB memory, Windows Server 2008 R2 (x64) operation system, .NET 4.0 runtime. The measurement results are listed in Table 3.3.

Table 3.3: Characteristics of the state space traversal

Parameter	Coloured saturation	Lazy saturation
Run time	367 s	242 s
Number of global states	$2.701 \cdot 10^{12}$	
State space representation (nodes)	1 587	
Number of local state changes	10 084 401	1 864
Sum of nodes in next-state relations	164 711	66 741
Sum of nodes in $\mathcal{ER}$ relations	0	2 419
Total number of nodes	$2.131 \cdot 10^7$	$1.338 \cdot 10^7$

Run time represents the time needed to explore the state space. The state space generation required 367 s for the CPN model of the safety function using our former coloured saturation algorithm, and only 242 s with the new lazy saturation algorithm. This is a 35% improvement considering the runtime. Note, that former, non saturation-based approaches [NB09] could not discover the full state space of the model. The evaluation of the temporal expressions took considerably less time: deadlock freedom and reversibility checking temporal expressions took 6 s each to evaluate on the existing state space representation. The liveness and safety requirements were evaluated in 2 s and 3 s, respectively.

Beside the run time, the memory requirement is also the subject of interest. Measuring the memory consumption of programs executed in managed environment is problematic, because the garbage collector does not free up all the unused memory necessarily [GZF12]. However, as most of the memory is used by the nodes and edges of the decision diagrams, the number of these elements can be used as a representative of the memory consumption.

3.7 Thesis 1: Model Checking of High-Level Models

I used the Coloured Petri net formalism to develop formal models of complex systems. I have examined various verification approaches being able to analyse systems designed in high-level modelling languages, especially Coloured Petri nets. I investigated an industrial case study used as a motivation example, which revealed the shortcomings of explicit state model checking techniques: due to state space explosion, they can rarely handle the state space of real-life problems. Symbolic model checking algorithms provide a solution, and from the available approaches, I chose saturation as an extremely powerful method for the verification of Petri nets. However, systematically reviewing the literature I realised that saturation was not extended to handle Coloured Petri nets. I elaborated an approach to support the verification of high level Coloured Petri net based models. The existing algorithms can not handle complex guard expressions of Colured Petri nets, so I developed a new encoding of the next-state relation and I introduced efficient algorithms for the construction of the symbolic representation. The result of the research was integrated into the PetriDotNet model checking framework and proved its efficiency in an industrial setting.

Thesis 1 *I developed new verification algorithms for Coloured Petri nets. I devised an advanced disjunctive-conjunctive decomposition algorithm for the efficient representation of complex next-state relations of coloured Petri net models. The introduced new decomposition algorithm combined with the efficiency of saturation made the verification of even industrial problems possible. In addition, I developed an algorithm for the temporal decomposition of the construction of the complex next-state relations. This new algorithm further decreased the space requirements and runtime of the verification of models with complex guard expressions. I proved the correctness of the presented algorithms.*

The results of my first thesis decreased the space requirements of handling complex next-state relations by constructing smaller next-state representations for Coloured Petri nets. As a consequence, the time requirements of the verification process also decreased, and a new set of problems could be verified: the result of the thesis made possible to solve even real-life industrial examples. The new algorithms are evaluated on a model of an industrial safety-critical system (PRISE): it was the first time when the verification of the correctness properties could be verified on the entire Coloured Petri net model of the safety-logic. Successful verification proved the correctness of the system with regard to deadlock freedom, safety and liveness properties.

Publications: My new results introduced in this thesis were published in the journal paper [5] and in the following conference papers: [22] and [16]. The results contributed to the conference paper [7] and journal paper [1].

Chapter 4

Parallel Saturation-based State Space Exploration

Verification requires significant computational resources to succeed. In order to extend the limits of verification, even advanced techniques such as saturation or other symbolic techniques need further improvements to be able to solve complex problems or existing problems more efficiently. Various approaches are known to improve the performance of the algorithms, one of them is parallelisation. Recent advances in computer engineering and the increasing number of computational units in modern computers make this direction more and more attractive.

The saturation algorithm introduced an efficient traversal strategy which kept the state space representation small and could explore huge state spaces fast. However, this algorithm is inherently sequential, as both decision diagram manipulations and also the iteration strategy of the algorithm follows a well-defined, strict order of steps. Decision diagram manipulation is traditionally difficult to parallelise, and the literature also states the same for saturation [CZJ09]. Given the doubly recursive dependencies of saturation, and the top-down dependencies in the decision diagram manipulations, cumbersome synchronisation and locking mechanisms are required for the parallel implementation of saturation.

In this section I will investigate an existing parallel saturation algorithm from [ELS06] and I will introduce algorithmic improvements to increase the efficiency of parallel saturation-based state space traversal.

Publications related to this chapter. The results of this thesis were published in [17], and this chapter is based on that paper.

Implementation and contributors. The parallel saturation algorithm introduced in this chapter was implemented and made available in the PetriDotNet framework. While the theoretical and algorithmic contributions were mainly my results, the implementation of the presented algorithm is the result of the whole PetriDotNet team, and especially my students Tamás Szabó and Attila Jámbor.

4.1 Challenges

In the development of the parallel saturation algorithm, efficiency and correctness are the two most important issues. Ensuring correctness requires that the following properties of the saturation are preserved:

- Bottom-up order of saturating/finishing nodes.
- Local fixed-points are reached.
- Consistent caches store the final results of the computations.
- Decision diagram data structures are kept consistent.

In addition, beside the rigorous synchronisation, the parallel algorithm has to utilise the available resources. This means that the independent tasks have to be recognised and run parallel. As the iteration order of saturation highly depends on the structure of the decision diagram (and the model of the system), the structure of the system and also the decomposition, static division of the exploration into subtasks is not possible. Instead, during the traversal, the computing nodes have to wait for the various tasks produced by the other threads.

4.2 Cache Data Structures in Saturation

When discussing parallel algorithms, one has to look deeper into the data structures and implementation details. In the former section, saturation was discussed from a high-level point of view. In this section, I show the data structures used by saturation. In the following sections, I will also discuss how to use them in a parallel setting.

Unique table. Decision diagram manipulations rely on the efficiency of a caching mechanism, the so-called *unique table*, or *UT* for short. The *UT* is implemented as a hash-table, which contains $\langle key, node \rangle$ pairs: when the algorithms finish the computation of a node in the decision diagram, they look for a corresponding node in the *UT*, which represents the same function. If there is no such node, the algorithms put the new node into the *UT*. If the *UT* contains a node representing the same function, then the algorithms will use that node in the future. Putting a node to the *UT* is commonly referred as the operation so-called *check-in*.

Fire cache. The saturation algorithm uses special caching mechanisms in the procedure $Saturate$ (Algorithm 1 at line 4) and in the procedure of the relational product computation function $RelProd$ (Algorithm 2 at line 5). The cache in the procedure saturate is not an essential part of the algorithm (for some performance penalty it might be omitted), but the so-called fire cache (*FC*) is essential to avoid redundant computations (so it significantly reduces the computational complexity). The algorithm extensively uses the *FC* , so in the following, the role of this special cache is overviewed. As saturation computes the effects of the transition firings locally, the *FC* can be used to store the effects of the transition firings. This cache enables the algorithm to find if an operation has already been executed. The *FC* stores the effects of the relational product computations: the *FC* maps a pair of decision diagram nodes (in the state space representation and next-state representation) to another decision diagram node (in the state space representation).

4.3 Parallel Saturation

In this section, the algorithm from [ELS06] is introduced. This algorithm served as the basis of my improved algorithm, which is presented in Section 4.4.

The authors of [ELS06] divided the saturation into several stages, and defined the computation of each node as the elementary step in the algorithm which can be assigned to an individual thread. Node computations and operations consist of:

- node management in the MDD data structures,
- event and next state computations,
- node modifications,
- the manipulation of the MDD by recursive calls.

According to the investigations[ELS06], the tasks which can run parallel are the firings of the events i. e., the next-state computations on the decision diagram representation of the state space. These tasks are executed either by one thread or by multiple threads. When a thread finished a local computation, it calls other threads to do the remaining tasks i. e., firing transitions at a lower level or higher levels of the decision diagram. This way the iteration of saturation is cut into smaller pieces that have the proper size to be executed by a thread. The logic of deciding which tasks are outsourced by a thread to another is a critical point. These tasks should be large enough to avoid the increase in synchronisation and communication overhead, but they also should be of reasonable size to enable more threads to work parallel. The parallel saturation algorithm implements the work pool design pattern to provide flexible distribution of the tasks.

Beside efficiency, it is also important to ensure correctness by avoiding inconsistent MDD states and ensuring synchronisation otherwise the algorithm is not able to reach a fixed-point. For this purpose, various mechanisms are introduced. The sequential saturation algorithm was modified by introducing and exploiting the following means:

- work pool design pattern to ensure efficient parallel execution,
- new attributes in the decision diagram data structure for state space representation,
- data structures and mechanisms for the *FC* to ensure synchronisation and mutual exclusion,
- data structures and mechanisms for the decision diagram representations to ensure mutual exclusion and
- data structures and mechanisms for the other data structures (such as local states, next-state representation) to ensure mutual exclusion.

4.3.1 Extending the Decision Diagram Node Data Structure

The node data structure in the decision diagrams has to be extended to support parallel execution. The parallel saturation algorithm divides the saturation task into smaller subtasks that are run parallel, but their results have to be synchronised, and the work which has been done by other threads has to be registered. Hence the node data structure is extended with fields to store the synchronisation information. The following attributes are used for synchronisation purposes:

- *upward arcs* register arcs into which the result of the computations on the actual node will be inserted
- *ops* integer variable counts the remaining tasks; when this variable is set to zero, then the algorithm finishes the saturation of the node
- *saturating* is a Boolean variable to indicate if the actual node was started to be saturated
- *key* stores the key with which the node was inserted into the *FC* .

The algorithm uses these attributes at various procedures at various points in the execution. In the following, we shortly overview them. For the manipulation of upward arcs, the algorithm uses two functions:

- $SetUpArc(t_k, s_{k+1}, j)$ sets an upward arc in t_k pointing to the edge j of s_{k+1} (on an above level), and
- $GetUpArc(s_k)$ returns pairs of the form $\langle r, i \rangle$ representing an upward-arc where r is a node above and i is the index of an arc (of node r).

The integer variable ops is used to register the number of remaining tasks which should be finished on the node. The value of this variable might be increased and also decreased during the traversal. Variable ops is increased when a new thread starts working on that part of the state space representation (a function is called on the node).

Variable key stores the key of the node which is used to place and find the nodes in the *UT* . The algorithm stores this key for the following reason: as the node is continuously changed by many threads, using the key argument it is easier to decide if the *UT* has to be updated with the new key. The key is continuously updated, but the thread does not always have access to the exact state of the node (especially if other threads changed it) to compute the key; storing the key in the node means that the algorithm can excess the node in the *FC* .

Mutual exclusion has to be ensured i. e., only one thread can change the value of the arguments of the nodes at a given point in time (this part is omitted from the pseudo codes).

4.3.2 Working of the Algorithm

The algorithm [ELS06] introduced a correct synchronisation and locking mechanism for parallel saturation. In the following, the extensions to the sequential saturation iteration strategy are introduced: the application of the synchronisation and the locking mechanisms at the various phases of the computations is detailed.

Synchronization of data structures The saturation algorithm uses decision diagrams, therefore it has to take care of the consistency of their underlying hash tables. The unique table (*UT*) is used to store the nodes of the decision diagram. The goal of the algorithm was to enable as many threads to manipulate nodes simultaneously as possible. The algorithm synchronises the manipulation of the data structures at every level, this way avoiding inconsistent MDD levels. The responsibility for global MDD consistency is left to the iteration, which is preserved with locking sub-MDDs when they are manipulated.

Synchronization of MDD operations The parallel saturation algorithm uses a special locking strategy to preserve MDD consistency. As MDD serves as the underlying data structure for the iteration, preserving consistency is a critical task during parallel saturation. A classical decision diagram approach was used in [ELS06], so at every operation, the argument MDD-s are locked to prevent concurrent manipulation. This approach introduces a relatively high synchronisation overhead, but it is essential for ensuring consistent manipulation. A big advantage of saturation is that it tries to avoid operations on the whole decision diagram, instead, local operations are computed. This means that the algorithm locks only smaller parts of the decision diagram representation, so the algorithm itself is subject to smaller locking overhead. Therefore small MDD operations are a characteristic of saturation and smaller parts of the decision diagrams are locked.

Synchronization of the iteration An important task is to preserve the correct iteration order. The threads have to synchronise the operations executed on nodes: the algorithm has to avoid the

redundant computation, but it has to be ensured that all the transition firings are executed and no next-state computation is omitted.

The iteration is synchronised with the help of the special additional data structures introduced in Section 4.3.1. Every node has a counter ops to register the tasks which are under execution or are planned to be executed. This counter prevents the algorithm to finish the computation and finalise the results before all the tasks have been finished.

In order to preserve dependencies, the algorithm uses the upward arcs as depicted on Figure 4.1. These arcs represent the dependencies in the iteration order. If a node has an upward arc pointing to a node in the upper level of the state space representation that means: a thread computed the firing at the upper node and it called another thread to compute the lower levels of the MDD rooted there. Figure 4.1 depicts the step when the procedure $Saturate$ submits the relational product computation tasks to other threads.

The algorithm also avoids redundant computations by using the various cache structures of the sequential saturation algorithm. However, the *FC* is extended to not only store the values of finished computations but also serve as a synchronisation mechanism among the threads. The caching mechanism of the *FC* is extended with synchronisation constructs: when a thread starts computing a part of the reachable state space, it signs it in the cache by placing the actual node with a flag. This way, if another thread would start exploring that part of the state space, it can check in the cache that it is still being processed: redundant exploration is avoided, and the new thread just registers itself for the result. The key argument of the nodes supports that the threads will not miss the nodes in the cache.

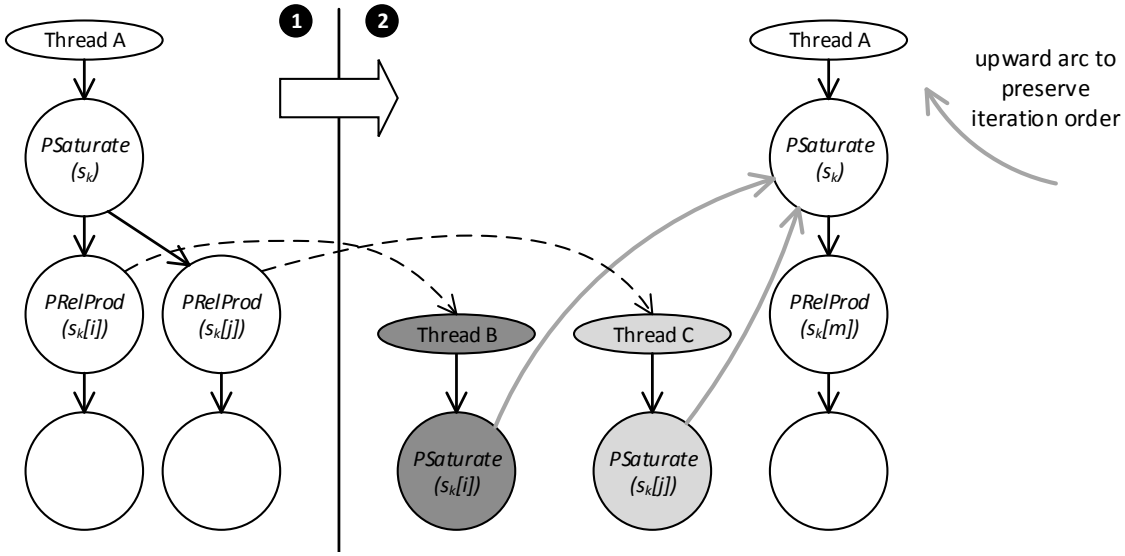


Figure 4.1: Using upward arcs in saturation

Locking mechanism. Beside synchronizing the executed tasks, it is also important to keep the consistency of the data structures: the parallel saturation algorithm introduced a locking mechanism for this purpose. The locking strategy is simple: one thread can modify a node at a single point of time which is ensured by using locks. During the next state iteration, the sub-MDD rooted in the manipulated node is also locked, so the algorithm avoids that two threads use the same sub-MDD and contained nodes during the node manipulations.

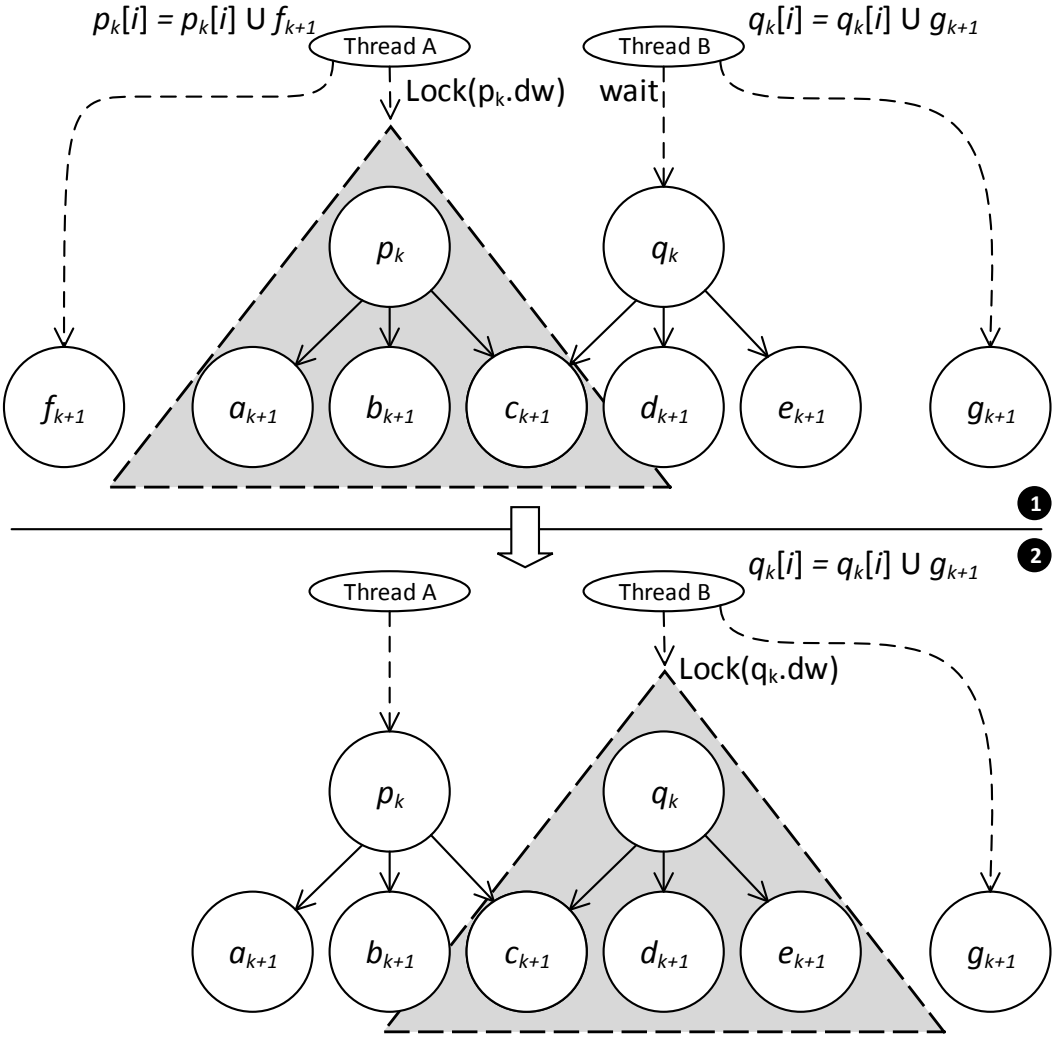


Figure 4.2: Locking in saturation

Details of the algorithm. The formerly introduced eager fixed-point computations are calculated parallel in this algorithm. Synchronization and locking mechanisms were formerly summarised, here an insight is given to the main operation of the algorithm. The pseudo-code of the algorithm is depicted on Algorithm 14, Algorithm 15, Algorithm 17 and Algorithm 16 with the modifications, so the pseudo code contains the improvements (new locking mechanism) marked with an asterisk. The original algorithm locks bigger parts of the operations as it is stated in [ELS06] and as it is depicted on the figures (discussed below). For the exact pseudo code of the original algorithm, the reader is referred to [ELS06], here we only discuss the general idea.

Local fixed-points are computed by procedure *PSaturation* of Algorithm 14 and transition firings are computed by the relational product computation function *PRelProd* of Algorithm 15.

The operation of the work distribution in the algorithm is depicted in Figure 4.1. In this figure, Thread A starts saturating a node s_k. During the computation recursive calls are needed for computing the next states. These calls are outsourced to other threads. In order to preserve the iteration order, these threads set an upward arc to the arc of the upper node (arc j of s_{k+1}) and increase the value

of variable ops of s_{k+1}. This way the algorithm ensures that the upper node could not be finished until the nodes below are finished. The signature of the parallel version of saturation and relational product computation is similar to their sequential counterpart, only function $PRelProd$ receives an additional parameter representing an arc of the callee: the set represented by this arc will be extended with the result of the procedure $PRelProd$.

Beside implementing a parallel saturation iteration and synchronising, these functions also lock the decision diagrams when the union is computed and the arcs are updated with the newly discovered states. Functions, which are responsible for synchronization: function $Lock(p_k.dw)$ locks the MDD down from the node p_k to ensure that no other thread will manipulate it; and function $Unlock(p_k.dw)$ frees the lock and lets other threads working on the MDD. These functions lock the MDD data structure downward in order to prevent concurrent manipulation: this mechanism is depicted on Figure 4.2 where two threads want to compute unions of the diagram, but they use a common argument node, so Thread B has to wait.

In addition to the above defined functions, the parallel algorithm published in [ELS06] uses function $Remove(s_k)$ of Algorithm 17 for removing dead endings from the MDD. These are created when a parallel thread starts a computation of a firing of a dead transition, which cannot fire from the given marking, and it is only detected at a later phase of the firing i. e., in lower levels of the decision diagram.

The parallel algorithm also introduced the procedure $NodeSaturated(s_k)$ of Algorithm 16 to finish the saturation when the firings of the transitions are finished. In this case, function $NodeSaturated(s_k)$ checks if the iteration reached a local fixed-point and then it persists the decision diagram node into the cache data structures.

Correctness. The locking ensures that the iteration order is preserved, and operations executed on nodes are not interfered by each other. The algorithm is proved to be correct [ELS06], as it ensures:

- correct iteration order: by removing synchronisation methods we get the sequential algorithm
- correct synchronisation of the data structures, both in the MDD operations and both in the next state representations
- since locks ensure that updating a node is atomic, exhaustively firing transitions will result in the same MDD shape for a saturated node as in the sequential algorithm

A more detailed proof can be found in [ELS06].

4.3.3 Problems

Parallel implementation of saturation involves a big synchronisation overhead, making efficient parallelisation difficult. This also emphasises the fundamental role that the proper synchronisation plays in the parallel realisation of the saturation algorithm. There are two main bottlenecks: first is that parallelisation of state space exploration is generally a hard task. In order to avoid redundant state exploration, we have to ensure that the parallel directions synchronise properly without dramatically increasing the synchronisation costs. Another reason is that saturation uses a special underlying data structure: decision diagrams. Parallelizing decision diagram operations involves a big synchronisation overhead, caused by the fact that decision diagrams are built in a bottom-up fashion, where upper levels highly depend on lower levels. As measurements showed in [ELS06], the parallel saturation algorithm runs faster on more processors than on one, but still remains slower than the sequential algorithm by 10-300%. Scalability is also an important factor in parallelisation. By scalability we mean the following two characteristics:

- The runtime of the algorithm will decrease with respect to the increasing number of resources.
- The relative speed of the parallel algorithm will increase comparing to its sequential counterpart with the growing number of tasks

It is important to examine the scalability of the parallel algorithm. The following problems were revealed by the experiments [ELS06] (which were also confirmed in our paper):

- the parallel algorithm could not exceed the speed of the sequential one, independent on how much the resources were increased;
- increasing the model size (and the amount of tasks that should be solved by the algorithm) does not yield and advantage for the parallel algorithm;
- the sequential algorithm solved the problems faster than the parallel even for large models and also for environments with more computational units.

These findings motivated my work to investigate the algorithm and find a way to improve it.

Algorithm 14. PSaturate

```
   input  : s_k : node
1  // s_k:  node to be saturated,
   output : node
2  if s_k = 1 then
3  |  return 1;
4  Return result from cache if possible;
5  k ← Level(s_k);        // retrieve the actual
     level of the MDD
6  t_k ← new Node_k;
7  t_k.saturating = true; // saturating the node
     has started
8  t_k.ops = t_k.ops + 1; // increase ops counter
9  foreach i ∈ S_k : s_k[i] ≠ 0 do
10 |  t_k[i] ← PSaturate(s_k[i]);
11 repeat
12 |  foreach ε ∈ E : k ∈ Top(ε) do
13 |  |  R_ε ← N_ε as decision diagram;
14 |  |  foreach s_k[i] ≠ 0 ∧ R_ε[i][i'] ≠ 0 do
15 |  |  |  f ← PRelProd(t_k[i], R_ε[i][i'], t_k, i');
16 |  |  |  if f ≠ 0 then
*17|  |  |  |  f ← t_k[i'] ∪ f;
*18|  |  |  |  Lock(t_k[i']);
*19|  |  |  |  t_k[i'] ← t_k[i'] ∪ f;
*20|  |  |  |  UnLock(t_k[i']);
21 |  |  |  if i' ∉ S_k then
22 |  |  |  |  Confirm(k, i')
23 |  |  |  Build(k);
24 until t_k unchanged;
25 t_k.ops = t_k.ops − 1; // decrease ops counter
26 if t_k.ops = 0 then
27 |  NodeSaturated(t_k);    // finish saturation
28 return t_k;
```

Algorithm 15. PRelProd

```
   input  : s_k, R, s_{k+1} : node;
            j : localstate
1  // s_k:  from node of the firing,
2  // s_{k+1}:  top node,
3  // R:  next-state representation node,
4  // j:  next state of the top node,
   output : node
5  if R = 1 then
6  |  return s_k;
7  t_k : Find result of (s_k, R) in cache;
8  if t_k ≠ 0 then
9  |  if t_k is not saturated then
10 |  |  SetUpArc(t_k, s_{k+1}, j);
11 |  |  t_k.ops = t_k.ops + 1;
12 |  |  return 0;
13 |  else
14 |  |  return t_k;
15 k ← Level(s_k);        // retrieve the actual
     level of the MDD
16 t_k ← new Node_k;
17 t_k.ops = t_k.ops + 1;
18 SetUpArc(t_k, s_{k+1}, j);
19 Put t_k in cache, flag set to not saturated;
20 foreach s_k[i] ≠ 0 ∧ R[i][i'] ≠ 0 do
21 |  f ← PRelProd(t_k[i], R_ε[i][i'], t_k, i');
22 |  if f ≠ 0 then
*23|  |  f ← t_k[i'] ∪ f;
*24|  |  Lock(t_k[i']);
*25|  |  t_k[i'] ← t_k[i'] ∪ f;
*26|  |  UnLock(t_k[i']);
27 |  if i' ∉ S_k then
28 |  |  Confirm(k, i')
29 t_k.ops = t_k.ops − 1;
30 if t_k.ops = 0 then
31 |  if ∃i ∈ S_k : t_k[i] ≠ 0 then
32 |  |  Put t_k into the saturation queue;
33 |  else
34 |  |  Remove(t_k);
35 return 0;
```

Algorithm 16. NodeSaturated

```
   input  : s_k : node
 1 // s_k:  node to be saturated,
   output : node
 2 Put node s_k into the UT ;
 3 Update entry of s_k in FC ;
 4 s_k.ops = s_k.ops + 1;          // increase ops
     counter
 5 foreach ⟨r, i⟩ = GetUpArc(s_k) do
*6   u ← s_k ∪ r[i];
 7   if u ≠ r[i] then
*8     Lock(r[i]);
*9     r[i] ← r[i] ∪ u;
*10    UnLock(r[i]);
 11    Confirm(k + 1, i)
 12    if r.saturating then
 13      foreach R_ε[i][i'] ≠ 0 : k + 1 ∈ Top(ε) do
 14        f ← PRelProd(r[i], R_ε[i][i'], r, i');
 15        if f ≠ 0 then
*16          f ← r[i'] ∪ f;
*17          Lock(r[i']);
*18          r[i'] ← r[i'] ∪ f;
*19          UnLock(r[i']);
 20        if i' ∉ S_k then
 21          Confirm(k, i')
 22        Build(k);
 23    r.ops = r.ops − 1; // decrease ops counter
 24    if r.ops = 0 then
 25      if r.saturating then
 26        NodeSaturated(r);   // finish saturation
 27      else
 28        Put r into the saturation queue;
```

Algorithm 17. Remove

```
   input  : s_k : node
 1 // s_k:  node to be removed,
 2 Update entry of s_k with 0 in FC ;
 3 foreach ⟨r, i⟩ = GetUpArc(s_k) do
 4   r.ops = r.ops − 1;          // decrease ops
       counter
 5   if r.ops = 0 then
 6     if r.saturating then
 7       NodeSaturated(r); // finish saturation
 8     else
 9       if ∃i ∈ S_k : r[i] ≠ 0 then
10         Put r into the saturation queue;
11       else
12         Remove(r);
```

4.4 Algorithmic Improvements

I have investigated the main characteristics of the parallel saturation algorithm. The intrinsic complexity of saturation makes the experimentation and understanding difficult. The findings were the following:

- The increasing number of computing nodes does not lead to increased performance, instead increased waiting times.
- Ensuring consistency and locking required significant resources.
- Correctness was also ensured in practice: the results of the parallel and sequential algorithms were compared.

According to the experiences, the goal was to decrease waiting times by introducing a more fine-grained locking strategy which would decrease the synchronisation costs.

I have modified the parallel saturation algorithm, and I developed a new synchronisation mechanism to improve the algorithm presented in [ELS06]. The goal of the new locking mechanism is

to localise the effect of the locks and to reduce the overhead caused by them. The improvements led to significant speed-up of the algorithm. Investigating the iteration order of the parallel algorithm revealed that the complex locking strategy could be redesigned to lock only single nodes. I introduce local synchronisation, which avoids downward locking of sub-MDDs. Beside the fact that locking downward sub-MDDs poses significant computation needs, additionally in many cases, the inefficient synchronisation makes the threads unable to run parallel.

4.4.1 New Locking and Synchronization Strategy

In this section, I introduce a new synchronisation and parallel iteration method. The goal of the new algorithm is to decrease the overhead of downward locking. The new locking strategy is shown on Algorithm 14, Algorithm 15 and Algorithm 16. The changes compared to the former parallel approach are marked with an asterisk ($*$), the rest of the algorithms are based on the formerly introduced parallel saturation algorithm.

The locking strategy has to ensure the consistency of the data structures. When the former algorithm had to compute the arc updates, the decision diagram was locked downward to ensure that no other thread will modify it during the arc manipulation. Instead of this solution, I propose to use an *arc locking strategy* to lock only the node, and especially the actual arc being processed. Function $Lock(p_k[i])$ locks the arc i of the MDD node p_k to ensure that no other thread can use it. In the former algorithm $Lock(p_k.dw)$ was used which locked the decision diagram downwards. Function $Unlock(p_k[i])$ frees the arc of the node and makes it available for other threads to work on. In the former algorithm, $Unlock(p_k.dw)$ was used to free the decision diagram downwards. I do not detail the inner locking strategy (implementation of $Lock$ and $Unlock$) as I used a programming environment built-in library for this purpose with proven correctness.

In addition, investigating the iteration strategy of saturation revealed that locking could be applied lazily which can reduce the synchronisation overhead: my proposed solution exploits the fine-grained arc locking strategy and puts the union computation outside the scope of the locking. However, letting the threads do the image computations and the update operations in parallel may lead to lose information and spoil convergence. The old algorithm could do in this case the following: updating arc i leading to node p with the newly discovered local state space represented by q and r should result that the arc i leads to a node representing $p \cup q \cup r$, but if the threads compute $p \cup q$ and $p \cup r$ and update arc i independently, then the result will be only the subset of the expected.

To ensure the correct iteration and avoid the information loss, I propose to compute the result of the union twice. The placement of locking is depicted on Algorithm 14 at line 18 and line 20, Algorithm 15 at line 24 and line 26, and Algorithm 16 at line 8 and line 10 and line 17 and line 19. At each procedure, the union of the old edge and the recently discovered state space representation is computed twice at the following points: Algorithm 14 at line 17 and line 19, Algorithm 15 at line 23 and line 25, and Algorithm 16 at line 6 and line 9 and line 16 and line 18. At first, the union is computed outside of the scope of the locking so that they can run in parallel. After the union computation, the algorithm locks the node and computes the union once more and updates the arc of the corresponding node. Even if multiple threads are working on the same node, as the algorithm locks the node when the union is computed, and the arc is updated, it is ensured that the algorithm does not lose information, so the convergence of the algorithm is ensured. However, the question naturally arises why we need to compute the union twice. The answer is that in most cases the union will be computed by the unlocked union indeed, and the locked union will only get the result from the corresponding cache (union cache). As receiving the result from the cache is a fast operation, the node will be locked only for a very short period. However, when the interference of the threads results that the value of the

arc is updated during the union computation, then the thread will lock the arc for the time when the union is computed, and the arc is updated with the results.

The new locking strategy significantly reduces the computation cost of the synchronisation.

A flag is introduced as an attribute in the node data structure. Atomic operations are ensured on the arcs with the help of this flag, without making the MDD operations such as the union or next-state computations mutually exclusive. This locking mechanism is applied in the fixed point computation at every iteration step when the set represented by the arc is augmented. As functions $PSaturate$, $PRelProd$ and $NodeSaturated$ are all augmenting the set represented by the node, they all use this new synchronisation strategy. A simplified view of the locking strategy is depicted on Figure 4.3, where the main idea of the arc locking strategy is compared to the solution of [ELS06] (referred to as old locking strategy on the figure). The node is locked for the time when the data structures are manipulated, and also the cache is locked. However, the computationally most expensive task, when saturation extends the represented set of states, can run parallel as the scope of locking is restricted to the arc manipulation.

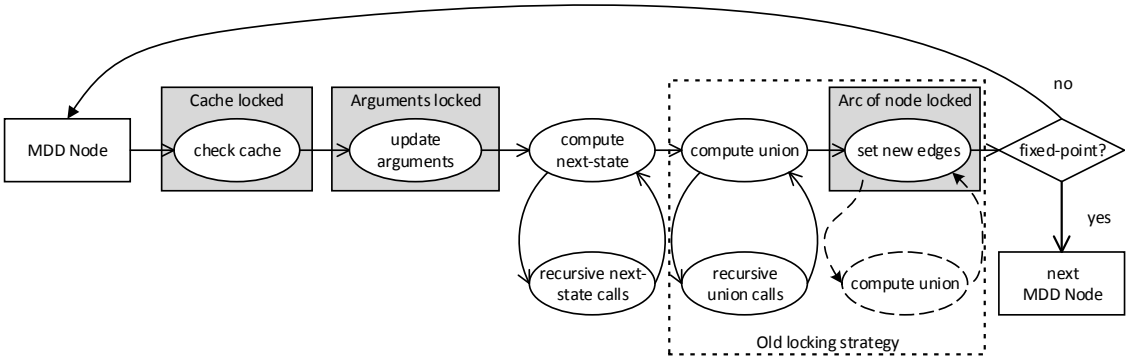


Figure 4.3: Workflow of the new synchronization strategy

4.5 Correctness of the Algorithm

The introduced locking and synchronisation strategy is proved to be the proper solution because if the scope of the locking would be reduced, we could lose information, the algorithm would be slower, or the convergence could be spoiled. On the other side, the introduced algorithms ensure correct iteration and parallel state space exploration.

4.5.1 General Issues

The correctness of saturation was proved in many papers as it was discussed in former sections. The basic parallel saturation algorithm was presented in [ELS06], where the correctness of the algorithm is also proved. The main challenge in parallel saturation is to ensure uncorrupted iteration order. Otherwise, either convergence is lost, or the final result is just the subset of the real state space. To ensure both convergence and avoid omitting states, the sequential algorithm was extended in [ELS06] with the locking and the proper work distribution mechanisms. These modifications let the algorithm run hardly parallel, which is confirmed by the measurements in [ELS06] and [17].

The modifications I presented exploit the resources of recent multiprocessor architectures more efficiently in parallel saturation-based model checking. Now I prove the correctness of the presented

approach. I discuss only modifications affecting the iteration order, as other improvements concern mainly the implementation. The modified algorithm should:

- Preserve iteration order,
- Reach saturated final state,
- Preserve consistency of data structures.

Iteration order is not affected by my modifications, so the reader is referred to [ELS06] for a complete proof.

The procedures of [ELS06] are also introduced in my solution to compute the next-state $\mathcal{N}_\varepsilon$ by $PRelProd$ on Algorithm 15 and by $Remove$ on Algorithm 17, the transitive closure $\mathcal{N}_k^*$ computed by $PSaturate$ on Algorithm 14 and $NodeSaturated$ on Algorithm 16. Consequently calling these functions preserves the iteration order as it is proved in [ELS06]. After an iteration is finished, function call $NodeSaturated$ ensures that every node encodes a saturated set, so the iteration is completed. The algorithm also ensures that function $NodeSaturated$ finalizes the nodes after saturation and puts them into the corresponding data stores in line 2 and line 3.

4.5.2 Correctness of the Iteration

The improvements to the algorithm modified the order of the union function calls. However, by using the tricky redundant union computation scheme, the convergence of the algorithm is ensured, and the final result is the same as for the traditional sequential saturation. Commutativity of the union operation is exploited to ensure that the final result contains the whole set of states computed by the threads independently. For each $thread_i$, f_i represents the result computed by the thread and the set represented by f_i is used to extend the set represented by arc $s_k[j]$. The algorithm ensures that the arcs will properly represent the results of the next-state computations, so the set represented by $s_k[j]$ will be the following: $\bigcup_{\forall i}(f_i \cup s_k[j]) = s_k[j] \cup \bigcup_{\forall i}(f_i)$.

4.5.3 Consistency

Preserving the consistency of data structures is especially important in parallel algorithms, and it is highly affected by the new locking strategy. Consistent data manipulation is required to ensure global consistency: the goal is to ensure that during the decision diagram and node manipulations, and also during the procedures of saturation, the algorithm should use and produce consistent data. As I used up many parts of the former algorithm, in the following, I will concentrate on the new parts, namely to the computations of the union operation. It is important to examine whether the new locking and iteration strategy did not spoil the consistency of the data structures, the data structures being used in the union should be in a stable state (they must not change during the operations). The new approach omits downward locking and preserves consistency without locking the argument sub decision diagram of the union operation. From the consistency point of view, it has to be assured that the operations do not change the state of the operands of the union operation.

Instead of proving the consistent manipulation of the nodes, I will prove a stronger statement which also implies the consistency of the operations. My assumption is that the algorithm performs MDD operations only on nodes, which are permanently placed in the MDD data structures: with the correct locking of the actually manipulated node, this ensures that the consistency of the data structures is preserved. What we have to prove now is that the algorithm only performs actions on finalised nodes that are placed in the *UT* i. e., the nodes used for compute the union are checked-in (Algorithm 16 at line 2). This fact also implies that the result will be consistent.

The assumption can be proved inductively. In the following, I go through the proof. The saturation algorithm with the introduced arc locking strategy is correct as the parallel saturation algorithm ensures that the arguments of the union operation are checked-in.

- At the beginning of the algorithm, all edges are set to terminal nodes, so the condition holds: we can only apply operations on nodes being checked-in.
- Each step of the algorithm either sets an arc to point to a checked-in decision diagram node which is in the *UT* , or the algorithm delays the computation and sets upward-arcs to represent the dependency.

The first condition expresses that the initial state is consistently represented. The second condition needs a more rigorous analysis, so I will detail why individual steps ensure that the union computations use only consistent results of the former steps. Each step of the parallel saturation algorithm, which extends the represented set of states, uses only a decision diagram node if it represents a finished part of the state space and it has already been put into the decision diagram data structure i. e., the *UT* . Correct operation of functions *NodeSaturated*, *PRelProd* and *PSaturate* ensure that only checked-in nodes are used during the construction of the state space representation, which is summarised in the following.

Saturating the nodes: *PSaturate* starts the saturation of a node by firing the transitions. The consistency of the computations in *PSaturate* is preserved if the results of the function *PRelProd* are checked-in to the *UT* . The function *PSaturate* does not place nodes into the decision diagram data structures but if the saturation of a node can be finished then function *NodeSaturated* is called at line 27 to finalise the results and data structures.

Firing transitions: Transition firings are executed by the function *PRelProd*, which sets upward-arcs: in Algorithm 15 at line 10 and line 18 an upward-arc is set to represent the dependency which means the upper node can not be finished before the actual node is saturated. The function *PRelProd* returns in both cases the terminal node **0**. The callee that is either function *NodeSaturated*, *PRelProd* or *PSaturate* will examine the returned value and it realizes (in the different procedures) at line 22, line 16 and line 15 that the returned value is the empty set, so no further work has to be done now. Upward-arcs will be processed in later phases by other threads.
The result of *PRelProd* on which the union operation is called in other operations is consistent as:

- *PRelProd* returns the node **0** if the computations are not finished in lower levels (this also means a new upward-arc to represent the dependency) or
- *PRelProd* detects that this step has already been executed and returns a consistent value from the *FC* (the returned decision diagram node is also in the *UT*).

Transition firings executed by the function *PRelProd* only returns a node different from **0** if the result is in the *FC* and it is a saturated decision diagram node (depicted at line 14). This node can be safely used by other functions and threads to compute consistent results. The results of *PRelProd* are used in function *PSaturate*, *PRelProd* and *NodeSaturated* at line 15, line 21 and line 14 to compute the new set of states.

Adding new nodes to the decision diagram: We have to overview how new nodes can appear in the decision diagram. The parallel saturation algorithm finishes the saturation of a node by calling function *NodeSaturated*. Function *NodeSaturated* checks-in the argument node at line 2 and places the actual node into the MDD data structure. This node will not change in the future. Function

NodeSaturated also updates the *FC* at line 3 and ensures that the cache hits will contain saturated nodes that can be used safely in the operations. Note that when the cache hit is not saturated, then function *PRelProd* will only return **0** instead of the unfinished nodes in *FC* .

Using nodes in the procedures of saturation: As it was discussed, a complex traversal strategy ensures that the decision diagram nodes are only used for the construction of the state space representation after they are checked-in. Decision diagram nodes may reach the union operation in the following ways:

- When a node is saturated, the state set is extended by calling the operation union in function *NodeSaturated*. In this case, the algorithm computes the union of a recently saturated node with the old arc of the upper node, which has already been saturated. Both nodes are checked-in, the result is consistent.
- The other case is when computation of the next-state succeeds as the node is found in the cache. The algorithm uses the cache value only in the case if it is saturated, so this argument of the operation is finalised. The other argument of the union is also saturated as it was formerly the endpoint of an arc of the node. As both nodes are saturated, which means that they were put into the data structures by function *NodeSaturated*, the result will also be consistent.

Arc locking: Besides that the operation of the union is called on saturated nodes being checked-in, mutual exclusion of the arcs has to be ensured. $Lock(t_k[i])$ prevents other threads reading this arc during the union computation and arc manipulation, so the consistency of the operations is also ensured. As well known locking mechanisms are used inside the function *Lock*, we can expect that the synchronisation of the data structure is correct. Note that as we formerly discussed, union computations are only called on nodes being checked-in to the *UT* , so only saturation and transition firings can modify the actually locked node. As parallel saturation preserves the saturation iteration strategy according to [ELS06], only function *PSaturated* and *NodeSaturated* will manipulate the actual node being locked, which means that only local arc manipulations are executed, it is enough to lock only the arcs during the traversal.

4.6 Implementation

We have implemented the formerly mentioned algorithms in the PetriDotNet framework[17]. We have developed a complex synchronisation mechanism at the data structure level of saturation to prevent data races and to ensure consistent execution. We have implemented a mutually exclusive access to the data structures of the next-state computation, such as state transition representations and globally reachable states. In addition, the implementation also pays attention to the access of the data structures used for mapping the Petri-net states to the symbolic data structures.

The access to the MDD data structures is serialised at every level, in this way we can preserve the consistency of the data structures. MDD operations manipulate the data structures and rely on the synchronisation mechanisms of the *UT* , *FC* and the other data structures. The fact that MDD operations do not modify the nodes which are checked-in to the *UT* significantly reduces the synchronisation cost. However, this approach will increase the number of produced nodes: unnecessary nodes should be cleaned from the data structures. Every node has a counter counting the references pointing to it so that we can decide at any time to clean the data structures and the algorithm can easily decide if a node is necessary. The algorithm introduced in [ELS06] exploited the *FC* data structures

and used it as a pre-cache mechanism to avoid redundant state space exploration. We have implemented this method in our approach too. Using this cache for synchronisation helps avoid redundant state-space computations. We only have to register the event and the node immediately if the event is executed on it. All other threads intending to explore the same sub-state space will realise that it is now executed, and the new threads just register themselves for the result. The synchronisation of this cache is important. Our approach does not use a global cache; instead, we assign a cache to each level. This reduces the synchronisation costs. The same strategy is used for the union operation, as the algorithm does not lock the operations, but the MDD levels and the cache data structures, for the time of modifications. This strategy enables the parallel computation of union operations even with common arguments, which was a shortcoming in former algorithms. This leads to increased parallelism and reduced overhead.

4.7 Evaluation of the Algorithm

In this section, the measurements of the new algorithm are presented, and the approach is compared to the former algorithm of [ELS06].

4.7.1 Environment

We have developed an experimental implementation in the Microsoft C# programming language. We used some of the framework's built-in services, like the so-called ThreadPool and also the built-in locking mechanisms. We examine our algorithm and compare our approach both to a sequential algorithm written in C#, and to the implementation written in C programming language [ELS06]. We used a desktop PC for the measurements: *Intel Core2 Quad CPU Q8400 2,66GHz, 4 GB memory.* For our implementation we used *Windows 7 Enterprise, .NET 4.0 x64.* To run the implementation from [ELS06] we used *Ubuntu 10.10 with gcc-4.4.*. Comparing the performance of [ELS06] and my approach implemented by our team is a little bit difficult. In order to make the comparison more realistic, our implementation was extended to handle Kronecker matrix based next-state representation. However, our implementation computes the local states dynamically. In contrast the algorithm [ELS06] needs a pre-computation step and works with a formerly computed Kronecker representation, so they are two different variants of saturation. Former measures[CMS05] showed that with the use of precomputed Kronecker representation 50-60% speedup can be gained. However in most cases, the user has to adapt the model to some special requirements[CMS05], so it is more difficult to use. The models we used for the evaluation are widely known in the model checking community. Flexible Manufacturing System (FMS) and Kanban system are models of production systems [7]. The parameter N refers to the complexity of the model, and it influences the number of the tokens in it. Slotted Ring (SR) and Round Robin (RR) are models of communication protocols [CMS05], where N is the number of participants in the communication. The state spaces of the models range from 10^{15} up to 10^{150}.

4.7.2 Objectives of the Measurements

Measurements were conducted to evaluate both runtime and memory requirements. PetriDotNet was developed in the Microsoft C# programming language, so the program runs in a managed environment, memory measurements are very difficult. The implementation (work of Attila Jámbor, Tamás Szabó and Dániel Darvas) follows best-practices from the community and some special heuristics were also added (such as reusing freed node data structures). However, memory consumption was still not optimal and varied from run to run. The reason for this phenomena is that parallel threads

SR(N)	30	60	90	120	150
sequential	0.66s	4.5s	14.8s	34.7s	70.7s
parallel	0.64s	4.5s	14.4s	33.8s	65.2s
speed-up	1.03	1.0	1.027	1.027	1.084
Kanban(N)	50	100	200	300	400
sequential	0.5s	5.1s	63.2s	295s	890s
parallel	0.4s	2.6s	20.5s	80.6s	228s
speed-up	1.25	1.96	3.08	3.66	3.90
FMS(N)	50	100	150	200	250
sequential	1.7s	14s	61s	180s	444s
parallel	1.2s	7.9s	27.1s	67s	143s
speed-up	1.41	1.77	2.25	2.68	3.10

Table 4.1: Runtime results of our algorithm

SR(N)	30	60	90	120	150
sequential	0.2s	1.4s	4.4s	10.2s	19.7s
parallel	0.4s	2.3s	7.5s	17.1s	34.4s
speed-up	0.5	0.61	0.59	0.6	0.57

Table 4.2: Runtime results of [ELS06]

consumed the memory in an irregular manner and depending on the garbage collection strategy, the runs had different runtime characteristics. As optimizing memory consumption in a parallel environment is a challenging task, we did not aim to provide proper memory measurements. However, as an observation 20-100% memory consumption increase was caused by the parallel algorithm. A more rigorous analysis is left to the future.

4.7.3 Runtime and speed-up results

Slotted Ring: The regular characteristic of the model suggests that it cannot be parallelised well. Our measurements show that the parallel algorithm has the same performance as the sequential one. Also, as the size of the model grows, the parallel algorithm outperforms the sequential one up to 8.4%. Comparing this result with the runtime of the former implementation (in Table 4.2), the version written in C is faster as the programming environment yields less overhead. In addition, the solution in Table 4.2 exploits precomputed next-state representations, whose computation time is not included in these measurements.

However, we can also take another viewpoint and examine the relative speed-up of the algorithms compared to their sequential counterpart. When we examined the relative speed of the algorithms, our approach reached 8% runtime gain compared to its sequential counterpart, while the old one from [ELS06] just about 40% runtime penalty. This suggest that the new locking strategy and iteration lets the algorithm exploit more efficiently the additional computational power of multi-core computers.

Kanban: The introduced algorithm turned to be very efficient for the Kanban model. The state space exploration of the Kanban system was 25% faster with the parallel algorithm for still small models. However, for bigger models, the performance gain of the parallel algorithm increased. The

model	Dining Philosophers	Round Robin
size	1000	1000
sequential	0.91s	17.9s
parallel	1.35s	34.6s

Table 4.3: Runtime of not parallelizable models

measurement showed that the parallel algorithm is nearly 4 times as fast as the sequential (Table 4.1). Comparing my algorithm with [ELS06] shows that the parallel algorithm from [ELS06] is about 50% slower than its sequential counterpart. Direct comparison is omitted here as we can not reproduce the measurements of [ELS06], for us, the tool of [ELS06] did not produce the results of that paper (indeed, it produced much worse).

Flexible Manufacturing System: The FMS model has a huge state space and complex interactions, which means that saturation faces problems during the state space traversal. However, this irregular structure of the model and its state space yields more work for the threads of the parallel algorithm: the parallel algorithm runs at least 41% faster than the sequential one. For large models, the sequential algorithm needs 3 times as much time as the parallel one. We could not compare this result with [ELS06] due to a segmentation error.

Models where parallel execution is not efficient: The efficiency of symbolic methods is highly model-dependent. This is especially true for saturation and parallel saturation. Those models that can not be verified by saturation due to the high memory consumption, these models could also not be verified with parallel saturation. As parallel saturation usually uses 10-50% more memory than the sequential one, the models which do not fit into memory in the sequential case will also not fit in the parallel case. On the other side, for highly regular models, where sequential saturation turned out to be extremely efficient, parallelisation leads to 30-50-100% runtime overhead. These models usually have only a few nodes at each level, so they provide less work for parallel threads. Moreover, saturation usually finishes state space generation within a second, so the overhead of creating threads also makes worse the performance of the parallel algorithm. In the following table (Table 4.3) we show the runtime results for two extremely big, but extremely regular models of [CMS05]. We present here the measures for our algorithm, but the former approach [ELS06] produced similar runtime results in general.

4.7.4 Scalability

In this section, the scalability of the new parallel algorithm is investigated: the scaling of the runtimes with the number of used processors is measured. The goal of the new algorithm was to reduce the synchronisation overhead and increase parallelism in the execution. In this section, I show the results of the FMS model with parameter $N = 200$. The execution time of the parallel algorithm is compared to the runtime of the sequential algorithm, and the number of available processors is increased to 1-2-3-4 CPU-s. Figure 4.4 shows that the algorithm scales well with the growing number of processors. Also, the algorithm is faster than the sequential one still on one CPU, as it can efficiently exploit hyper-threading technology and that we can run multiple threads on one CPU.

We also examined the scaling of the runtimes with the growing size of the models. Figure 4.5 depicts the runtimes of the parallel and sequential state space generator algorithms presented in this

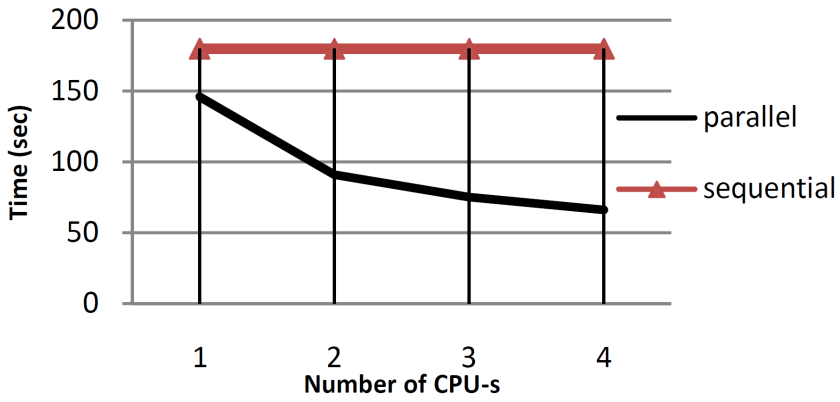


Figure 4.4: Runtimes of our implementations, FMS model

paper for the FMS model. The advantage of the parallel algorithm grows with the growing number of tasks meant by the bigger models (N is the size of the model).

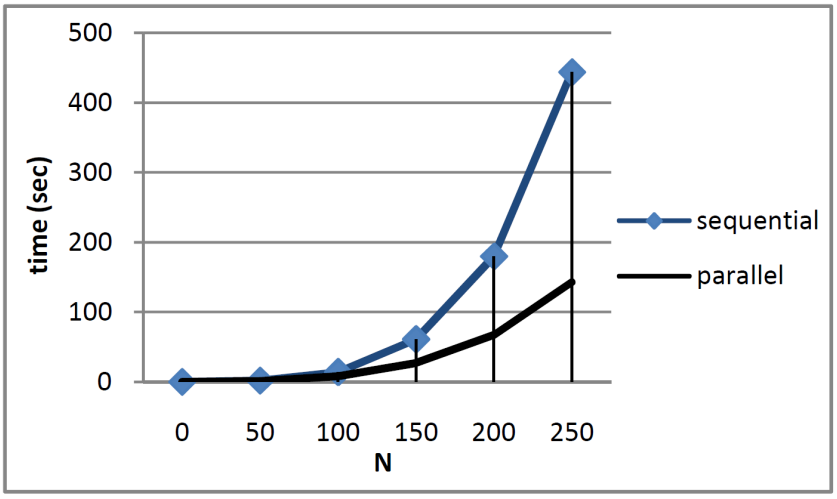


Figure 4.5: Scaling of the parallel algorithm

4.7.5 Summary

The introduced parallel algorithm is more efficient than its sequential counterpart regarding the runtime performance. However, from the memory consumption point of view, the situation is different: as parallel threads start computing more "dead endings" (directions where no solution can be found), memory consumption is usually 10-50% more than for the sequential algorithm. Comparison of the introduced approach and the former one of[ELS06] is quite difficult: as they use neither the same kind of saturation algorithm nor the same programming environment, runtime results are not easily comparable. However, the speedup factor compared to the sequential counterparts of each algorithm suggests that the introduced new locking and synchronisation strategy leads to the more efficient parallelisation of the computations.

4.8 Thesis 2: Parallel State Space Exploration Techniques

I investigated various techniques to speed-up the model checking algorithms. Extending existing algorithms to exploit the computational power of modern multicore computers necessitates the construction of rigorous parallel algorithms and synchronisation mechanisms. Extending symbolic algorithms to run parallel is especially challenging due to the complex data structures and intricate symbolic computations. Saturation uses a special iteration strategy for traversing and building the symbolic representation of both the state space and the next-state relation in an incremental manner, which means that the steps heavily rely on the results of the former computations. This makes the parallel implementation a challenging task. I investigated the existing parallel saturation algorithm, and I identified some points where it could be improved. I introduced a new synchronisation mechanism which reduces the synchronisation overhead and it could significantly speed up the model checking algorithm. The developed parallel algorithm could exploit the computational power of modern multi-core processor computers in saturation-based state space exploration.

Thesis 2 *I developed a parallel saturation based state space traversal algorithm using a novel synchronisation method and locking strategy. The new locking strategy applies a fine-grained locking mechanism, which only synchronises the manipulation of the state space representation. The algorithm prevents the occurrence of inconsistent states and ensures the correct execution of the saturation iteration order. The new synchronisation algorithm decreased the synchronisation overhead and led to increased parallelism. The new parallel algorithm can exploit the computational power of modern multicore computers by decreasing the synchronisation overhead – for certain benchmark models – significantly. I proved the correctness of the new parallel algorithm.*

Various measurements showed the competitiveness of the new algorithm on benchmark models. The new algorithm scales with the growing number of computation units better than the former approaches. The new synchronisation algorithm significantly reduced the synchronisation overhead, and it could lead to significant performance gain compared to former approaches.

Publications: My new results introduced in this thesis were published in the conference paper [17].

Chapter 5

Synchronous Product Generation for LTL Model Checking

This chapter introduces my results in the field of automata theoretic model checking of regular properties. As I showed in the former chapters, CTL-based structural model checking was investigated by many researchers and used for the model checking of ordinary Petri nets. However, there was a need for a larger set of specification languages. LTL model checking has different expressive power and it is considered more convenient for engineering applications [Var01]. My work in this chapter is a step towards providing language theoretic LTL model checking by using regular properties as the input of the model checking procedure. Later, this research was continued by my colleagues, and it was extended to provide saturation-based verification for the full set of LTL properties.

Publications related to this chapter. The initial results of this chapter were published in [24] and than it was continued in [20], this chapter is based on the later paper. Based on these results, the synchronous product generation approach was further improved later by my colleague, who combined it with efficient on-the-fly LTL model checking algorithms in [11] and [2]

Implementation and contributors. All the algorithms presented in this chapter were implemented and made available in the PetriDotNet framework as a joint effort of the development team. The algorithms of this section were implemented by my student, Vince Molnár.

5.1 Introduction

Model checking requires that the property should be expressed either declaratively in a temporal logic language or imperatively by using, for example, an automaton formalism.

Many kinds of properties might be of interest from which safety properties constitute a significant part of the verification problems [BKL08].

The verification of safety properties is often reduced to analysing finite traces. Regular languages can express a safety subset of LTL properties, and finite automata accept regular languages describing safety properties. In addition, the finite automaton formalism is used to express safety properties: finite automaton is a simple formalism, can be used conveniently and can naturally express finite error traces. For these reasons, it is widely used by the verification community, but it is also widely accepted by software engineers to specify requirements (for example, in the form of protocol state machines). Automata can be used to specify the correct or the incorrect runs of the system. However,

when applying model checking, traditional automata-theoretic model checking approaches require that the automata used by the algorithm represent the incorrect runs of the system. This can also be achieved in the case when the given automaton describes the correct traces: the complement language will define the incorrect behaviour, and an accepting automaton can be constructed for this purpose by switching the accepting and non-accepting states of the property automaton.

When the automata describing the violations of the property is given, automata-theoretic model checking can be applied, which involves two main challenges:

1. compute the synchronous product of the system automaton with the property automaton describing the possible property violations, and
2. check if the product language is empty.

Constructing the synchronous product is a computationally expensive task as it can easily blow up the size of the state space representation. This causes not only a storage complexity, but it also makes language containment analysis difficult. Various techniques exist to decrease resource consumption and make the verification more efficient: one of them is on-the-fly model checking. Doing the verification on-the-fly during the state space traversal has two advantages:

- Synchronous product computation may filter out parts of the state space which are irrelevant for the verification.
- The verification can stop immediately when an error is found, and the remaining parts of the state space will not be explored.

On-the-fly model checking is widely used with explicit state space traversal techniques. However, when extending symbolic state encoding techniques with on-the-fly model checking capabilities, synchronous product computation becomes a complex problem. Symbolic synchronous product computation involves two main problems to solve:

- encoding both the property automaton and also the state space symbolically, and
- synchronising the steps taken through the symbolic next-state representations.

These tasks are far from trivial. The big advantage of symbolic methods is that they can handle a huge set of states (and state transitions) together. However, this makes the synchronisation problem difficult: Synchronous product computation relies on stepping the transitions of the state space and the property automaton together. As symbolic methods handle sets together, doing element-wise synchronisation is challenging.

Some attempts [CGH97; STV05] target to combine traditional BDD-based symbolic approaches with automata-theoretic model checking. These approaches encoded the synchronous product and the transition relation of software and hardware models. However, these approaches do not provide a solution for asynchronous systems.

My goal was to devise an algorithm which can exploit the efficiency of saturation in automata-theoretic model checking.

In the following I will overview the general scheme of the automata-theoretic model checking approach and then I introduce my contributions:

- I have identified a special form of finite automata which has the necessary properties to serve as an input for the synchronous product computation.
- I investigated the properties of this special class of finite automata.
- I devised an algorithm based on saturation to compute the synchronous product of the automaton and the state space on-the-fly during the state space traversal.

5.2 Preliminaries

In this section, the preliminaries of my work is introduces. I will shortly overview the target specification language and also the model checking approach, which serves as the algorithmic framework for my contribution.

5.2.1 Property Specification

In this thesis, I aim to provide an efficient verification technique for *regular safety properties*, which can be represented by a finite automaton. This class of properties consists of invariants, reachability and even more complex properties.

Regular expressions and LTL can be both used to define safety properties. However, due to the semantic difference, they used to be interpreted differently.

In the following, we will consider the runs of the system as paths/trajectories, where each state is consistent with a set of atomic propositions from the set of all atomic propositions AP.

The properties under consideration are defined below (according to [BKL08]).

Definition 1 (Safety Property) A property P is called safety property, if for each violation σ of the property, there exists a finite prefix $\hat{\sigma}$ of the violation such as $\sigma = \hat{\sigma} \cdot \sigma_{rem}$ and σ_{rem} is the arbitrary remaining part of the trace, then $\hat{\sigma}$ can not be a prefix of any trajectory satisfying the property, so for any $\sigma_P \models P$, $\sigma_P \cap \hat{\sigma} = \emptyset$. □

Definition 2 (Regular Safety Property) Safety property over AP is called *regular* if its set of bad prefixes constitutes a regular language over 2^{AP}. □

For example, invariants are regular safety properties: let invariant property P_{inv} prescribe state formula ϕ to be satisfied by all the states, then the violations of P_{inv} are characterized by the regular expression: $\phi^* (\neg\phi)\ true^*$. $true^*$ captures the fact that for regular safety properties, after finding an accepting path, the remaining behaviour of the system is not relevant. When dealing with LTL, it is assumed that the remaining behaviour of the system contains at least one infinite run. However, the model checking approach I suggest will terminate immediately when reaching an accepting state.

In the following the semantics of regular expression will be interpreted as follows: if a prefix of a trajectory leads to an accepting state of the automaton, then the trajectory is accepted by the automaton. So the expression $true^*$ is implicitly assumed to be at the end of each regular expression (for example, $\phi^* (\neg\phi)\ true^*$ will be equal to $\phi^* (\neg\phi)$).

5.2.2 Automata Theoretic Model Checking of Regular Properties

Deciding whether a system M satisfies a regular property P over a set of atomic propositions AP is reduced to checking language emptiness of the system constructed as the parallel (synchronous) composition of the model $\mathcal{M}$ and the automaton $\mathcal{A}$ for (the violations of) property P (referred to as a *property automaton*).

The language $\mathcal{L}(M)$ describes the runs of system M and the language of the possible violations of property P accepted by automaton $\mathcal{A}$ is denoted by $\mathcal{L}(A)$. The model checking procedure then reduces to check if $\mathcal{L}(M) \cap \mathcal{L}(A) = \emptyset$. If the intersection of the languages is not the empty set that means the existence of a feasible error trace. Figure 5.1 depicts the general workflow, which either produces an error trace (an element of the language accepted by the product automaton) or it proves the correctness of the system.

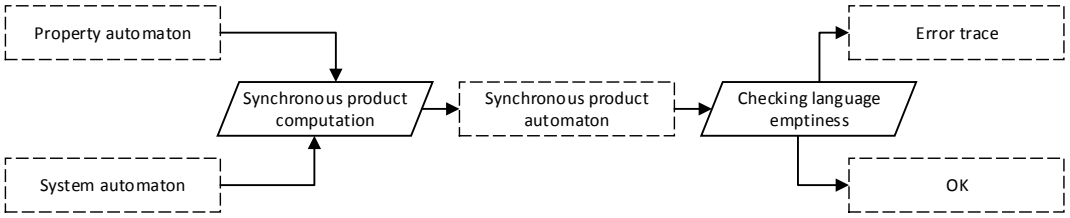


Figure 5.1: Automata theoretic model checking

Formally an automaton $\mathcal{A}$ is a tuple $\langle \Sigma, Q, \Delta, Q_0, F \rangle$, where Σ is a finite *alphabet*, Q is a finite set of *states*, $\Delta \subseteq Q \times \Sigma \times Q$ is a *transition relation*, $Q_0 \subseteq Q$ is a set of *initial states* and $F \subseteq Q$ is a set of *accepting states*. A run of an automaton over an input word is a sequence of states starting with an initial state, where the transition relation holds between the consecutive states. A run is *accepting* if it passes an accepting state in F. In case of finite automata modelling regular safety properties, the alphabet is the set of possible valuations of atomic propositions in the property: $\Sigma = 2^{AP}$. In practice, the model checking procedure consists of constructing the synchronous product of the automaton and the transition system, and then finding accepting runs. The product inherits the accepting states of the property automaton and search procedure can be defined algorithmically as searching the product for accepting states.

Constructing a finite automaton of the system description necessitates the traversal of the state space (resulting $\mathcal{S}$) and the possible state transitions $\mathcal{N}$. The state space representation is considered as a finite automaton in the automata theoretical sense with all states as accepting. The reason behind representing the state space as accepting is that the synchronous product will inherit the acceptance condition of the property automaton.

Checking language emptiness is an important part of the automata-theoretic model checking algorithm. In this thesis, I reduce language emptiness checking to checking if an accepting state is reachable in the synchronous product. This can be achieved on-the-fly during the traversal, as I will show later.

LTL is a more convenient and wide-spread formalism to express properties so I will also use LTL requirements during this thesis. As in this thesis we will consider only reactive models, which do not contain deadlock, we can assume that regular properties are a subset of LTL properties. This also demonstrates that even though the introduced model checking algorithm necessitates the automata formalism as an input, the approach is neither restricted strictly to one property specification language nor to one specific automata formalism.

In the following, I show how properties can be specified to support synchronous product computation based on saturation. A new form of finite automata is introduced which supports the efficient encoding of the synchronous product and the synchronisation of the steps of the state space traversal. On top of the new automata formalism, I build a saturation-based on-the-fly model checking algorithm.

5.2.3 Synchronous Product

The synchronous product of the model $\mathcal{M}$ is defined by interpreting the transition system generated by $\mathcal{M}$ as a (finite) automaton. This automaton represents the possible states, i. e., the state space $\mathcal{S}$. A labeling function over $\mathcal{S}$ is defined as $L : \mathcal{S} \to 2^{AP}$ assigning a valuation of the atomic propositions of P to each state of the state space $\mathcal{S}$ (of $\mathcal{M}$). The alphabet of the automaton corresponding to $\mathcal{M}$ is the same as that of the property automaton: 2^{AP}. $\mathcal{N}$ is extended to support composition Inputs of its

transitions are the valuations assigned to the target state by L. Synchronous composition with this automaton forces the property automaton $\mathcal{A}$ to read the valuations that appear on a state sequence of $\mathcal{M}$. Regarding the structures defined so far, the synchronous product can be defined as follows: $\mathcal{M} \times \mathcal{A} = \langle \Sigma, S \times Q, \Delta^{\times}, S_{init} \times Q_0, F \rangle$, where $\Delta^{\times} = \{\langle\langle \mathbf{s}, q\rangle, \alpha, \langle \mathbf{s}', q'\rangle\rangle | \langle \mathbf{s}, \mathbf{s}'\rangle \in \mathcal{N}, \langle q, \alpha, q'\rangle \in \Delta, \alpha = L(\mathbf{s}')\}$.

5.3 Special Encoding Based On Constrained Saturation

In this section, I will characterise a special form of finite automata on which the symbolic encoding relies. After that, I propose a new encoding that can be used as an input for the constrained saturation algorithm to compute the product state space. This efficient encoding serves as the background of the new model checking algorithm of Section 5.4.

5.3.1 Tableau Automata

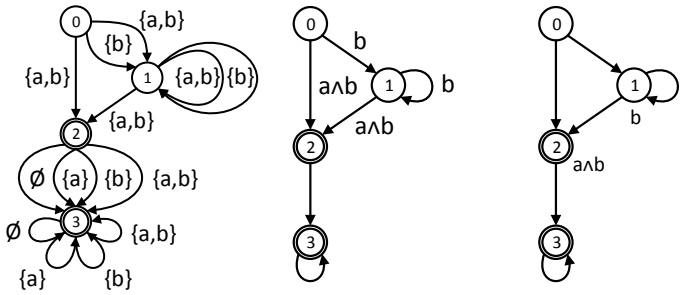
The first step in automata theoretic model checking is the translation of the property into a finite automaton. Observing the output automaton of widely used finite automata conversion algorithms such as [Brü93] and also tableau-based conversion algorithms (such as [Ger+95; SB00; Kes+93]) for LTL, I identified a common structural property that can be exploited to encode and compute the product efficiently. I refer to these kinds of automata as *tableau automaton*.

Definition 3 (Tableau automaton) A tableau automaton is a tuple $\langle AP, \Sigma, Q, \Delta, Q_0, F, L^+, L^- \rangle$, where

- AP is a set of atomic propositions,
- $\Sigma = 2^{AP}$ is the alphabet of the automaton,
- Q is the set of states,
- $\Delta \subseteq Q \times Q$ is the transition relation,
- $Q_0 \subseteq Q$ is the set of initial states,
- $F \subseteq Q$ is the set of accepting states, and
- $L^+ \colon Q \to 2^{AP}$ and $L^- \colon Q \to 2^{AP}$ are labeling functions, L^+ assigning propositions that must hold in the given state, while L^- assigning those that must not. □

A run of a tableau automaton over an input word is also a finite sequence of states $q_0 q_1 \ldots$ starting with an initial state $q_0 \in Q_0$, but unlike simple automata, there is an additional requirement beyond satisfying the transition relation. For every i, the input letter $\alpha_i \in 2^{AP}$ of the word representing a valuation of the atomic propositions must contain every proposition assigned to q_{i+1} by L^+ and must not contain any assigned by L^-, formally: $L^+(q_{i+1}) \subseteq \alpha_i$ and $L^-(q_{i+1}) \cap \alpha_i = \emptyset$. Accepting runs are defined the same way as for finite automata.

At this point, it is important to emphasise that tableau automata are only a special form of finite automata, with the same expressive power. An equivalent finite automaton has the same states (including initial and final states), the same alphabet, and a transition relation in the form of $\bigcup_{\langle q,q'\rangle \in \Delta} \{\langle q, \alpha, q'\rangle | L^+(q') \subseteq \alpha, L^-(q') \cap \alpha = \emptyset\}$. Because of this equivalence, we will often refer to a finite automaton directly corresponding to a tableau automaton to be in *tableau form*. In Section 5.3.3 I will show that every finite automaton can be transformed into this form.



(a) By definition (b) Compact representation (c) Tableau automaton

Figure 5.2: Three forms of a finite automaton corresponding to the regular property "$b^* \, (a \wedge b)$" (LTL property "$a \, \mathsf{R} \, b$").

In the following example, we will use a safety property which can be expressed either with the help of regular expressions or by using the LTL language.

Figure 5.2 shows different representations of a safety property expressed as a regular expression $b^* \, (a \wedge b)$ and an LTL expression $a \mathcal{R} b$. On Figure 5.2a, the corresponding automaton is shown exactly as it is described by the definition: each transition in the transition relation gets an own arc. Labels of the arcs are the sets representing valuations of the atomic propositions a and b. Figure 5.2b shows the same automaton in a more compact form, merging arcs and characterising their labels with a conjunctive expression. This automaton is in tableau form since all of the arcs targeting the same state are labelled by the same conjunction. Moving these labels to the state itself results in a tableau automaton shown on Figure 5.2c. Let ϕ_q denote the conjunction on the arcs targeting q. Then the labelling functions L^+ and L^- are defined such that every atomic proposition that is positive in ϕ_q is in $L^+(q)$, while those that are negative are returned by $L^-(q)$.

5.3.2 Encoding the Product Automaton

In this section, I investigate the main aspects of the symbolic encoding, and I suggest a solution which supports the efficient application of saturation.

For now, we assume that the state space and the next state relations of the system are already computed. Later in Section 5.4.2, I will show an on-the-fly construction algorithm. The specification is given as an automaton. Since the main goal is to exploit the power of saturation in model checking, we need to define the states and transitions in the way we did in Section 2.3. Formally a *product system* is built which is a tuple $\mathcal{M}^\times = \langle \mathcal{S}^\times, \mathcal{S}^\times_{init}, \mathcal{E}^\times, \mathcal{N}^\times \rangle$ collecting states in $\mathcal{S}^\times$, initial states in $\mathcal{S}^\times_{init}$ and transitions into $\mathcal{N}^\times$ preferably partitioned by events $\varepsilon \in \mathcal{E}^\times$ according to the events of the transition system.

Besides keeping the original state variables of the transition system, one or more additional variables are needed to encode the states of the automaton representing the property.[1] Note that every event of the product system must affect the encoding variables of the automaton, since their steps are synchronised. For this reason, to keep the efficiency of the saturation iteration strategy, these variables need to be situated in the lower levels of the decision diagram encoding: inserting them

[1]The property automaton is typically small enough to get encoded into a single variable, but a binary encoding (or anything in between) can also be used for a more compact representation.

immediately above the terminal level is an ideal choice. This way the encoding has no impact on the *Top* values of system events. Since the efficiency of saturation iteration strategy is *highly dependent on the Top values of system events*, not ruining the *Top* values produced by a good variable order is a sane requirement towards any algorithm.

The encoding of the transitions is a bit more challenging: as mentioned in Section 5.2.3 the steps of the system model and the property automaton need to read the same input letters, i.e. valuations of atomic propositions in the checked property. Thus it is insufficient to simply compute the Cartesian product of the next state relation of the model and the transition relation of the automaton.

In Section 5.2.3, we have already defined the product automaton of $\mathcal{M}$ and $\mathcal{A}$. To define the next state relation of the product system, we will drop the input labels of the transitions of the automaton. Saturation and model checking altogether is not interested in what input the product automaton reads during the state space traversal as long as the system model and the property automaton both read the same word. Formally, the next state relation of the product system is $\mathcal{N}^{\times} = \{\langle\langle \mathbf{s}, q\rangle, \langle \mathbf{s}', q'\rangle\rangle | \exists \alpha \in 2^{AP}, \langle\langle \mathbf{s}, q\rangle, \alpha, \langle \mathbf{s}', q'\rangle\rangle \in \Delta^{\times}\}$. While this definition is mathematically correct and can even be realised as conjunctive-disjunctive decomposition suitable for saturation [CMS05] (i.e. events are kept and the next state relation is the composition of next state relations of events), it fails to accomplish one of our main goals: preserving the *Top* values of events.

If the synchronisation on the input word is encoded into the next state relation, *Top* values of every event are inevitably raised *to the same value* that can even be K, the highest level possible. This means that saturation's strategy to apply the next state relation in a finer granularity is spoiled, every event is processed on the same level and the optimisations of saturation targeting concurrency are lost. To understand the reason for this rise in the *Top* values, we define the subject level of atomic propositions.

We assume that the truth value of an atomic proposition is only dependent on a single state variable, we call this the *subject* of the atomic proposition. Let $Sub(p)$ denote the level on which this variable is encoded in the decision diagram. Due to the synchronisation, each step of the system model results in a step of the property automaton. A step of the property automaton requires a full valuation of the atomic propositions, so every event $\varepsilon \in \mathcal{E}^{\times}$ of the product system now depends on all variables that are subjects of any $p \in AP$. By definition, this means that $supp(\varepsilon) \supseteq \{i | \exists p \in AP, Sub(p) = i\}$, i.e. the support of ε contains every level encoding variables that are subject to an atomic proposition in *AP*. It is easy to see that $Top(\varepsilon)$ is now at least $\max\{i | \exists p \in AP, Sub(p) = i\}$. For an example, imagine a property in which subjects of atomic propositions cover every state variable, so regardless of variable ordering, all *Top* values are raised to the maximum.

Since this is clearly not what we want to do, I devised a solution that preserves the *Top* values of the events by *decomposing the problem, separating the next state relation and the constraint of reading the same word.* This enables us to keep saturation's every advantage.

My proposed solution exploits the way tableau automata work. Furthermore, I employ the main idea of *constrained saturation*[ZC09]: check and fire. Instead of intersecting relations, I 1) relax the next state relation of the product to $\mathcal{N} \times \Delta$ in order to ignore the input of the participating automata and then 2) only allow state transitions reaching *legal* states. A reached state $\langle \mathbf{s}, q\rangle$ is legal, if the valuation $L(\mathbf{s})$ determined by the system's state satisfies ϕ_q, i.e. all propositions in $L^{+}(q)$ are true and those in $L^{-}(q)$ are false. I use the characteristics of tableau automata to be able to validate *states*, not *steps* – just like constrained saturation does.

I have to constrain the steps of relaxed next state relation $\mathcal{R} = \mathcal{N} \times \Delta$ to traverse only legal states. As I utilise the constrained saturation algorithm for this purpose, I have to compute the input constraint for the algorithm. We have to recall now that constrained saturation computes the set of states $\mathcal{N}(S) \cap \mathcal{C}$ in each step during the state space traversal, where $\mathcal{C}$ is the constraint characterising

possible states. Following this idea, I define the constraint as the set of legal states: $\mathcal{C}^\times = \{\langle \mathbf{s}, q\rangle | s \in \mathcal{S}, q \in Q, L^+(q) \subseteq L(s), L^-(q) \cap L(s) = \emptyset\}$ (where $\mathbf{s}$ corresponds to the vector representation of s).

Last but not least, the initial states of the product system can be obtained by pairing the appropriate initial states of $\mathcal{A}$ with initial states of M. The property automaton is typically interpreted such that the input of the first step from the initial state is the valuation implied by the initial state of M. This means that we initialise the property with the current (initial) state of the system, observing its behaviour starting from this point of time.

In Section 5.4, I will show how to provide an on-the-fly model checking algorithm based on the presented approach.

5.3.3 Investigation of Correctness and Efficiency

In order to prove the correctness of the algorithm, it is necessary to show that using the introduced next state relation and constraint, constrained saturation applies the same state transitions as saturation would use the previously defined next state relation $\mathcal{N}^\times$.

Theorem 1 *Given a set of states S, a product next state relation $\mathcal{N}^\times$, a relaxed next state relation $\mathcal{R}$ and a constraint of legal states $\mathcal{C}^\times$ the constrained saturation algorithm with $\mathcal{R}$ and $\mathcal{C}^\times$ computes the same set of states reachable with one step from S as the saturation algorithm does with $\mathcal{N}^\times$.* □

The reader shall consider the definitions of Section 5.3.1 and 5.3.2 in order to prove that the set of states $\mathcal{R}(S) \cap \mathcal{C}^\times$ constrained saturation computes are equivalent to $\mathcal{N}^\times(S)$ directly computed by saturation. The equivalence holds because $\mathcal{N}^\times$ contains only legal state transitions: while $\mathcal{R}$ is less strict, $\mathcal{C}^\times$ restricts it to legal states. Due to the introduction of tableau automata, constraining the transitions or the target states has the same effect.

I will examine the size of tableau automata in general. It is easy to see that most automata are not in tableau form, so they need to be transformed to get the corresponding automaton.

Theorem 2 *Given a (finite) automaton $\mathcal{A}$ with a state count of n over valuations of atomic propositions AP as an alphabet, a tableau automaton accepting the same language can be constructed with a state count at most $n \cdot O\left(2^{|AP|}\right)$.* □

Proof Constructive proof. Let q' be a state of $\mathcal{A}$ with input transitions $\delta = \{\langle q, \alpha, q'\rangle \in \Delta\}$. Denote the set of valuations appearing in these transitions by $\sigma = \{\alpha | \exists \langle q, \alpha, q'\rangle \in \delta\}$. Let ϕ_σ denote a logical function in minimal disjunctive normal form that is true for exactly the valuations in σ (such an expression always exists). Finally, denote the conjunctive parts of ϕ_σ by $c(\phi_\sigma) = \{\phi_\sigma^i\}$.

If $|c(\phi_\sigma)| = 1$, replace the parallel transitions with a single one of the tableau automaton, then label q' according to ϕ_σ. Otherwise, an automaton accepting the same language is obtained by splitting q into $|c(\phi_\sigma)|$ states $\{q^i\}$, each of them with the same outgoing transitions and input transitions defined by ϕ_σ^i. Once every (potentially new) state is processed, the result is a tableau automaton.

To examine the number of resulting states, consider that every state will only be split once. The number of states it is split into depends on the disjunction ϕ_σ, since every conjunction in it will yield a new state. It is well known that the upper limit on the number of conjunctions in a minimal disjunctive normal form of any Boolean formula over b binary variables is $O\left(2^b\right)$. As a result, every state will be split into at most $O\left(2^{|AP|}\right)$ new states. ■

Although this may sound very disappointing, I emphasize that widely-used tableau-based automata construction algorithms produce automata that are *always* in tableau form, so no more trans-

formation is needed. However, there may be a rightful need to further simplify these automata. Fortunately, many methods aiming to reduce the size of the property automaton keep the tableau form, such as those introduced in [EH00] excluding the reduction based on bisimulation, which can be modified to preserve the tableau form in exchange for some loss of compacting power. Note however, that for widely used regular safety and even temporal logic properties, the tableau form yields only a constant factor growth in the size: the efficiency of saturation used to overcome this difficulty as my measurements will also show.

5.4 Saturation-based On-the-fly LTL Model Checking

In the former section, I introduced an efficient encoding and computation scheme of the synchronous product. Now, I extend this framework to provide efficient on-the-fly model checking. The novelty of my approach is that it provides fully symbolic on-the-fly model checking, where both the state space and the automaton is encoded. The proposed algorithm computes the synchronous product on the fly during the traversal and searches for accepting states at each step.

5.4.1 Abstracting the Constraint

The presented algorithm of Section 5.3.2 introduced an efficient encoding of the product system by decomposing the transition relation into an over-approximating next state relation and a constraint of legal states. I utilised constrained saturation to build the state space of the product system. I defined the legal state constraint as set $\mathcal{C}^{\times} = \{\langle s, q\rangle | s \in S_{rch}, q \in Q, L^{+}(q) \subseteq L(s), L^{-}(q) \cap L(s) = \emptyset\}$. With the previously defined next state relation $\mathcal{R}$ and this constraint, the constrained saturation algorithm explores the state space and builds a symbolic representation of the synchronous product.

Now I introduce an abstraction layer providing the ability to build the symbolic product representation on the fly. This abstraction layer will "virtualize" the legal state constraint, letting the algorithm build the abstraction without precomputing the state space of the system model. The virtual constraint will encode the possible valuations and corresponding automaton states symbolically. Suppose that *AP* is a list of atomic propositions p_i ordered by $Sub(p_i)$, the level on which their subjects are encoded. Then the constraint is the set $\bigcup_{q \in Q}(p_1(q) \times \ldots \times p_n(q) \times \{q\})$ where $p_i(q)$ is a function assigning the possible valuations of p_i that satisfies the labeling of q, i.e. $p_i(q) = \{true\}$ if $p_i \in L^{+}(q)$, $p_i(q) = \{false\}$ if $p_i \in L^{-}(q)$ and $p_i(q) = \{true, false\}$ otherwise. Figure 5.3 shows the constraint of the example tableau automaton of Figure 5.2c.

In order to be able to use this abstract constraint for the on-the-fly traversal, I defined a function to map between the abstract constraint and the concrete states. This function is depicted on Algorithm 18. The function takes a constraint node c, a local state i and its level number l, and evaluates all atomic propositions whose subject is encoded on level l in a fixed order. If i fulfils p (i.e. $ap(i) = true$), then it steps downward in the decision diagram through the $true$ branch (reaching $c[true]$), otherwise through the branch labelled with $false$ (reaching $c[false]$). This way the algorithm is able to locally evaluate if a valuation of the global state satisfies the labelling of the reached state of the property automaton.

This approach has the advantage of using the constrained saturation algorithm with only a slight modification: the algorithm will only have to use the simple function described above to determine the next constraint node based on the valuations of the local states currently processed. This way the constraint only depends on the property automaton and can be built before starting the state space exploration.

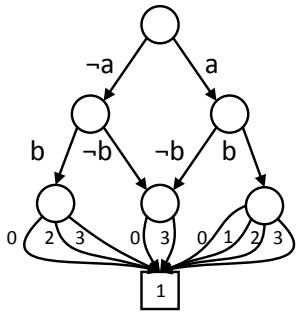


Figure 5.3: A constraint encoding the tableau automaton of Figure 5.2c.

Algorithm 18. StepConstraint

```
   input  : c : node i, l: indices
1  // c:  product constraint,
2  // i:  index of state
3  // l:  index of actual level
   output : node
4  if l = 1 then
5  |  return c[i];
6  foreach p ∈ AP, Sub(p) = l do
7  |  c ← c[p(i)] ;  // evaluate p on i
8  return c;
```

5.4.2 Units of Processing – A Framework for On-the-fly Model Checking

Based on the formerly introduced algorithms I developed an on-the-fly model checking framework. This framework uses constrained saturation with the abstract constraint for the exploration of the state space of the synchronous product. Now I extend it to be able to do on-the-fly model checking. The building blocks of the algorithm are the recursive saturate function calls computing local fixed-points. The introduced new algorithm combines the traversal with local accepting state detection, so after each local fixed-point computation I apply a search algorithm to detect accepting states in the sub-state space of the product.

The whole algorithm is similar to the constrained saturation algorithm, the extensions for being able to compute the synchronous product on the fly are signed with asterisks on Algorithm 19 and 20. $AcceptingStateDetection(t)$ is the function for searching accepting states locally in the sub-state space represented by decision diagram node t.

The introduced algorithm is easy to extend with SCC detection algorithms from [11] or other arbitrary SCC detection algorithm to provide LTL model checking: the only modification is that we have to change function $AcceptingStateDetection(t)$ searching for accepting states to a function, which looks for accepting cycles.

In order to evaluate the solution and compare it with traditional model checking algorithms, I use an SCC detection solution of [11], which combined the traditional Emerson Lei [EL86] SCC computation and an incremental approach of [Wan+01]. This way my algorithm can also provide verification support for the full set of LTL specifications and I am able to compare it with other LTL model checking algorithms.

Algorithm 19. ProdConsSaturate

```
   input  : c, s : node
 1 // c:  product constraint,
 2 // s:  node to be saturated
   output : node
 3 l ← s.level; r ← N_l^{-1};
 4 t ← new Node_l; pc ← c;
 5 foreach i ∈ S_l : s[i] ≠ 0 do
*6   pc ← StepConstraint(c, i, l);
 7   if pc ≠ 0 then
 8     t[i] ←
         ProdConsSaturate(pc, s[i]);
 9   else
10     t[i] ← s[i];          // no steps
         allowed
11 repeat
12   foreach i, i' ∈ S_l : r[i][i'] ≠ 0 do
*13     pc ← StepConstraint(c, i, l);
14     if pc ≠ 0 then
15       u ← RelProd(pc, t[i], r[i][i']);
16       t[i'] ← Union(t[i'], u);
17 until t unchanged;
18 t ← PutInUniqueTable(l, t);
19 AccetpingStateDetection(t);
20 return t;
```

Algorithm 20. RelProd

```
   input  : c, s, r : node
 1 // c:  product constraint,
 2 // s:  node to be saturated,
 3 // r:  next state function
   output : node
 4 if s = 1 ∧ r = 1 then
 5   return 1
 6 l ← s.level;
 7 t ← 0;
 8 pc ← c;
 9 foreach i, i' ∈ S_l : r[i][i'] ≠ 0 do
*10   pc ← StepConstraint(c, i, l);
11   if pc ≠ 0 then
12     u ← RelProd(pc, t[i], r[i][i']);
13     if u ≠ 0 then
14       if t = 0 then
15         t ← new Node_l;
16       t[i'] ← Union(t[i'], u);
17 t ← ProdConsSaturate(c, t);
18 t ← PutInUniqueTable(l, t);
19 return t;
```

5.5 Evaluation

We have developed an experimental implementation of the formerly introduced algorithms in the PetriDotNet framework and evaluated it by comparing runtime results on well known concurrent benchmark models, running the classic BDD-based algorithm of NuSMV, the state-of-the-art SAT-based algorithm IC3 [Bra+11] of nuXmv, the saturation-based algorithm of ITS Tools [Dur+11] and the introduced new algorithm called Tableau–ModelChecker, or shortly T–MC. The results are shown on Table 5.1.

My algorithm was extended with SCC detection (from [11] and [2]) to provide verification of the full set of LTL as it is also done by the competitors (this work was done by Vince Molnár). Note that if the comparison would only target regular safety properties, that would not be a fair comparison as all the competitors do LTL model checking by searching for strongly connected components.

The algorithms were run by using the same memory and computational resources to be fair: the computer used for the evaluation had Intel Xeon processors (4 cores, 2.2GHz) and 8 GB of RAM.

In Table 5.1, $-^{(\mathrm{M})}$ marks cases in which the tool ran out of memory, and $-^{(\mathrm{I})}$ is shown in case the input model was not constructed in the format of a tool. It can be concluded from the results that my new approach is competitive: ITS tool could significantly beat it for the Round Robin model, where the task is a simple cycle detection (which is not the target of my thesis). For all other models, my algorithm has the same runtime complexity, or it was faster than the other tools.

Figure 5.4 depicts a comparison of the scaling of the tools with the growing number of participants in the Slotted Ring protocol. I investigated a simple safety property here to focus mainly to the introduced new synchronous product generation algorithms (and not on the SCC detection problem).

Table 5.1: Measurement results of state-of-the-art model checking tools and T–MC.

N	Runtime (s)			
	NuSMV	nuXmv	ITS–LTL	T–MC
Counter–N, expression: $\neg\mathsf{G}(bit_{N-1})$				
10	**0.01**	**0.01**	—(I)	**<0.01**
15	0.21	0.02	—(I)	**<0.01**
20	15.30	**0.02**	—(I)	**0.01**
50	>1200	0.04	—(I)	**0.02**
DPhil–N, expression: $\mathsf{GF}(HasRightN) \rightarrow \mathsf{GF}(HasLeftN \wedge HasRightN)$				
50	>1200	178.87	0.22	**0.16**
100	>1200	>1200	0.64	**0.60**
200	—(M)	—(M)	**2.19**	2.45
300	—(M)	—(M)	**5.13**	5.93
Kanban–N, expression: $\neg\mathsf{G}(Pback2 = N-1 \vee Pback3 = N-1)$				
10	0.07	0.02	0.02	**<0.01**
20	9.51	0.02	0.03	**<0.01**
30	13.64	0.03	0.02	**<0.01**
SlottedRing–N, expression: $\neg\mathsf{G}(H_1 = 0 \vee G_1 = 0)$				
5	39.43	0.03	0.06	**0.02**
10	>1200	0.11	0.22	**0.07**
50	>1200	6.10	18.33	**2.69**
100	—(M)	—(M)	168.73	**16.51**
SlottedRing–N, expression: $\neg\mathsf{G}(C_N = 0)$				
5	14.45	0.04	0.03	**<0.01**
10	>1200	0.10	0.03	**<0.01**
50	>1200	5.87	0.19	**0.04**
100	—(M)	—(M)	0.58	**0.14**
RoundRobin–N, expression: $\neg\mathsf{G}(true)$				
10	>1200	0.08	0.08	**0.03**
50	>1200	4.70	**0.64**	1.37
100	—(M)	—(M)	**2.13**	9.12
FlexibleManufacturingSystem–N, expression: $\mathsf{F}(P1s = P2s = P3s = N)$				
5	>1200	0.03	0.13	**0.01**
10	>1200	**0.02**	0.53	**0.01**
20	>1200	**0.03**	3.70	**0.02**

It shows that for properties having compact tableau representation my model checking approach has competitive performance.

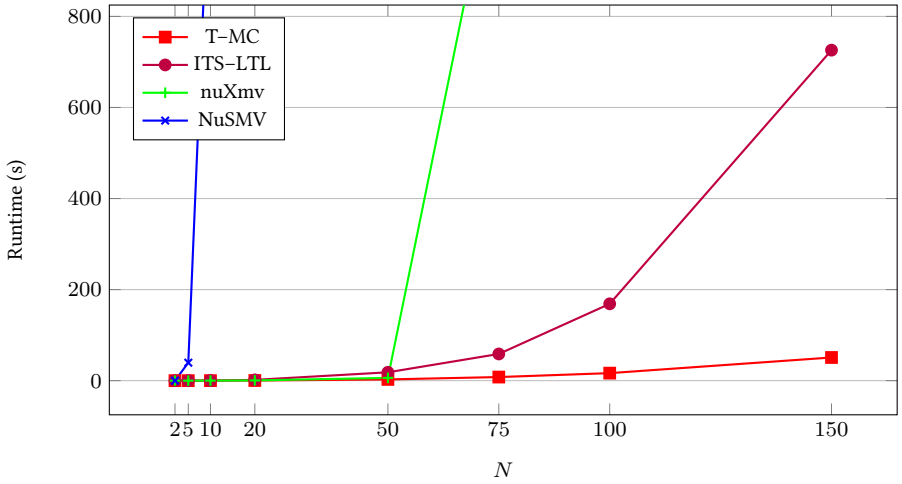


Figure 5.4: Slotted Ring, scaling (expression: $\neg\mathrm{G}(H_1 = 0 \vee G_1 = 0)$)

5.6 Thesis 3: On-the-fly Synchronous Product Generation for Model Checking Regular Safety Properties

Users have to analyse various kinds of properties, which can be expressed with the help of temporal logics. CTL and LTL are widely used temporal logics and they have different expressive power. Deadlock-freedom is expressible in CTL while fairness properties are supported by LTL. To support the engineers in verification, it is suggested to provide verification for both specification languages. I investigated the literature and efficient saturation based algorithms exist for the structural model checking of CTL properties. However, LTL model checking lacks the verification support based on efficient symbolic algorithms. In this thesis, I focus on a significant subset of the LTL language, namely regular safety properties. Model checking regular safety properties can be traced back to two main problems: synchronous product generation and detection of accepting states. Synchronous product generation is a difficult problem in a symbolic setting where one has to encode the property automaton and has to synchronise the steps with the state space. This is a difficult problem as saturation traverses the states in an irregular order which makes the synchronous product computation extremely difficult. I propose an efficient technique to compute synchronous product on the fly during the state space exploration and model checking of safety regular properties. The goal of the approach is to enable on-the-fly model checking during the state space traversal.

Thesis 3 *I developed a saturation based model checking algorithm for the safety regular subset of LTL properties. I propose a symbolic encoding of the automaton, and I introduce a new symbolic constraint to the saturation algorithm. I also introduce a new state space traversal technique to compute synchronous product on the fly during the state space traversal and do on-the-fly LTL model checking. The new algorithm served as the foundation of a new saturation-based LTL model checking procedure.*

My solution is the first algorithm which provides verification for a rich set of specification languages based on the saturation algorithm. The new algorithm extends the set of systems and requirements which could be verified by saturation. Various measurements showed the competitiveness of the new algorithm on benchmark models. In addition, the LTL model checker based on the new syn-

chronisation algorithm was the first which could verify LTL properties of the PRISE industrial case study.

Publications: My new results introduced in this thesis were published in the paper [20]. The results contributed to the conference paper [7] and journal papers [2] and [1].

Chapter 6

PetriDotNet Model Checking Framework

In this chapter, I put the pieces together and I introduce the PetriDotNet model checking framework. This chapter encapsulates the former results and also the results of many collaborators into a consistent, coherent framework. The main goal is enable users to use the algorithms in the modelling, formalization and analysis of their problems. This part of my work bridges the gap between the scientific results and the applications in the engineering domain and the introduced approach and the framework serves as a usable solution for the verification engineers.

My contribution in this chapter is the the novel combination of the various algorithms that yielded the efficient verification approach in the PetriDotNet framework. The devised approach proved its applicability in many research projects. In addition, the framework was used in various industrial case-studies to analyse the functional and even extra-functional aspects of system designs.

Beside the introduced verification workflow, I also made theoretical investigations regarding the CEGAR algorithm. The CEGAR algorithm complements the symbolic techniques, which are the topic of the former theses. However, the completeness of the Petri net CEGAR approach has never been evaluated before. In this chapter I also extend former theoretical results and I prove the incompleteness of the algorithm. This work served as a basis for other researchers to extend the CEGAR algorithm and integrate the new developments also into PetriDotNet .

Publications related to this chapter. The results of this chapter were published in [1], [4], [7], [10] and [15] and this chapter is based on that papers.

Implementation and contributors. All the algorithms presented in this chapter were implemented and made available in the PetriDotNet framework. The implementation of the presented algorithms is the result of the whole PetriDotNet team: Dániel Darvas, Vince Molnár, Attila Jámbor, Tamás Szabó, Ákos Hajdu, Zoltán Mártonka, Attila Klenik, Kristóf Marussy.

6.1 Model Checking Workflow

The goal of my work was to provide a comprehensive approach covering all phases of the verification process.

The aforementioned challenges belong to three aspects of the verification problem according to [BKL08]:

1. Modelling of complex systems.
2. Specifying formal requirements.
3. Verifying the model with regard to the requirements.

The goal of my research was to introduce a framework supporting the main tasks in verification. In this section, the proposed verification approach and the corresponding subtasks are introduced.

6.1.1 Modelling and Verification Approach

In this section I propose a modelling and verification approach. This new approach was designed according to lessons learnt from the research and industrial projects and case-studies of the research group. The introduced approach targets a certain class of problems: the modelling and verification of asynchronous, safety-critical systems with control and finite data.

Problem. Formal modelling necessitates a proper modelling formalism, which is able to capture the problem of the domain. Our experience showed that there is no comprehensive tool and approach which would support the modelling and analysis of asynchronous systems. Petri net based modelling languages provide modelling means for asynchronous systems. However, either a tool has an expressive formalism, but weak analysis such as [JKW07], or it provides efficient analysis techniques but difficult to use [Cia+03].

Formal verification requires expressive specification languages to be able to capture the intent of the designers. This is often a problem, as tools either support reachability checking, or CTL or LTL, but not all of them.

Verification tools supporting Petri net formalisms tend to use only one technique for verification. However, one technique is rarely enough to analyse all aspects of a system. There exists a framework using a wide range of techniques for the verification of synchronous hardware or software systems [Cav+14], but these techniques are not efficient for asynchronous systems [2]. LTSmin is another framework for process algebra, timed automata and extended state machines and it supports various symbolic and explicit techniques [Kan+15]. However, LTSmin does not support Petri nets as a modelling language, and it lacks those techniques from the literature that are efficient Petri net based models.

Summarizing the problems, we need a tool to support all aspects of the verification problem and an approach to support the verification of asynchronous, distributed safety-critical systems.

Goal. As it was discussed, ordinary Petri nets and Coloured Petri nets efficiently capture the behaviour of asynchronous, distributed safety-critical systems, so I propose to combine efficient model checking algorithms from the literature to provide LTL and CTL model checking for Petri net based models.

The overall goal is to provide a tooling for verification engineers. The approach targets verification engineers who aim to develop formal models and execute verification tasks. In order to cover the tasks arising during the verification of complex systems, three main functionalities have to be provided by the framework:

- Editor and persistence support for designing formal models in the Petri net and Coloured Petri net formalism,
- Specifying the formal requirements with CTL and LTL temporal logics,
- Model checking of the formal models if they satisfy the temporal logic requirements.

The goal is to provide modelling and verification support tailored to not a specific domain, but for a wide range of problems, which can be naturally captured by Petri net based models. The target problem domain of the framework is **asynchronous**, **concurrent** or **distributed systems** with data dependence.

Proposed approach. I propose a modelling and analysis approach which combines the expressive power of Petri net based models with the efficiency of saturation and abstraction based algorithms. The approach supports widely-used specification languages such as LTL and CTL.

I propose the verification workflow depicted on Figure 6.1 consisting of various methods to cover the main aspects of designing and analysing formal models. As verification is a complex task, a wide-range of algorithms is available, and the goal of the workflow is to combine the advantages of these techniques.

According to the literature, I propose to use, integrate and extend the following algorithms into a framework to provide formal modelling and verification support for the engineers:

- Saturation-based algorithms for the efficient state space exploration of PN and CPN models of asynchronous systems.
- Saturation-based LTL and CTL model checking algorithms.
- Bounded saturation and abstraction based algorithms such as CEGAR (Counterexample-Guided Abstraction Refinement) extends the verification capabilities of the framework to handle finite and infinite state models.
- Bounded saturation and abstraction based algorithms are used to generate counterexamples for safety properties.

Figure 6.1 depicts the verification workflow starting with the formal modelling step and the development of the formal specification. The goal of the framework is to provide Petri net based modelling languages such as Petri nets and Coloured Petri nets and also temporal logic-based specification languages such as CTL and LTL.

The proposed (symbolic) analysis methods need to encode the state space and also the next-state function symbolically. Kronecker matrix based representations are used for PN models, and I propose special algorithms for the handling of CPN models, these algorithms are the disjunctive-conjunctive and the lazy decomposition algorithms.

Various algorithms can be used to explore the state space: beside traditional saturation algorithm, bounded saturation can help verifying models with infinite state space and parallel saturation can exploit the computation power of recent multicore computers. In addition, if the model has infinite state space (for example when representing parametric systems), Counterexample-Guided Abstraction Refinement solves the safety verification problem efficiently. CEGAR can also be used efficiently in other cases of safety verification.

In case of intended LTL model checking, synchronous product generation is needed to compute the product representation of the state space and the property automaton.

Finally, we need temporal logic model checking algorithms in the framework to verify CTL and LTL properties.

Extending the state-of-the-art. In this section, I summarise the work I was involved in, and I will discuss how we advanced the state-of-the-art.

The proposed approach uses the state space exploration algorithm for Petri net and Coloured Petri net models, which is based on [CMS03; CY05]. An efficient structural CTL model checking approach of [CS03; ZC09] is integrated into the framework.

The idea of handling models with infinite state space continues the work of Ciardo et al. [Cia+03; YCL09]. The combination of counterexample generation with saturation follows also this line of research. However, beside the existing techniques, our research group extended the set of bounded state space exploration algorithms and introduced new bounded model checking algorithms. In addition, we utilised also a CEGAR approach of [WW11] to handle a new set of problems and also to provide efficient trace generation.

I propose to integrate saturation-based bounded model checking with structural model checking approaches.

The LTL model checking approach continues the results in this field of [Thi15; Wan+01; Dur+11; CGH97; STV05; Ger+95] and the goal is to exploit and combine their strengths, such as: on-the-fly LTL model checking, abstraction techniques specialised for LTL model checking, synchronous product generation and saturation.

However, some of these algorithms did not exist before I started to devise the approach. In addition, their integration into an efficient analysis framework was also far from trivial.

During the development, we aimed to extend the existing approaches with new algorithms to fill the research gaps. In the following the extensions are summarised, which had to be contributed to provide a flexible and configurable model checking process:

- Bounded saturation-based CTL model checking was presented in [3],[6],[14] and [18] in order to be able to efficiently combine bounded saturation-based state space traversal of [YCL09] with structural model checking [CS03; ZC09].
- SCC computation and efficient on-the-fly LTL model checking based on saturation presented in [11].
- New CEGAR algorithms in [4], [10] and [15] to extend the solvable set of reachability problems.

My contribution. Beside the complex approach that was put together by the extensive work of our research team, my contributions also significantly extended the applicability of the approach. The model checking framework was introduced in [1] and [7]. The verification process of the framework supports the verification of a wide set of problems, with the help of multiple combinations of algorithms. Beside the whole approach, Figure 6.1 highlights the steps improved by my work of this dissertation with a grey background.

My algorithmic contributions to the model checking approach are summarised as follows:

- New model checking algorithms for Coloured Petri net models presented in [5].
- New parallel saturation algorithm presented in [17].
- A new synchronous product generation algorithm, presented in [20] in order to provide saturation-based LTL model checking [2].
- Theoretical investigation of the Petri net CEGAR approach in [4] and [15].

6.1.2 State Space Exploration Techniques

Saturation is proved to be one of the most efficient techniques for the verification of asynchronous systems, which motivated our choice that saturation is used as the state space exploration engine in the framework. The introduced approach provides three different next-state representation and computation algorithms:

- Kronecker matrices,
- decision diagram based disjunctive-conjunctive decomposition, and

- decision diagram based lazy representation.

The different variants of Petri nets require different next-state representations to be efficient: Kronecker matrix representation is advised to the verification of ordinary Petri net models, disjunctive -conjunctive decomposition is used for Coloured Petri net models and the lazy approach yield efficiency gains over other approaches for Coloured Petri net models with large domains or bigger token counts. The algorithms can be combined with other algorithms presented in the framework.

6.1.3 Temporal Logic Model Checking

The framework provides support for various temporal logics. Saturation-based LTL and CTL model checking can be provided both for ordinary and also Coloured Petri nets as Kronecker matrices, disjunctive-conjunctive decomposition and also lazy saturation can be used.

CTL model checking is based on the traditional approach of [ZC09], which combines structural model checking with the so-called constrained saturation algorithm.

Automata-theoretic LTL model checking computes the synchronous product of the state space with the property automaton. Various automaton formalisms are supported with various synchronous product generation algorithms, which can be used for LTL model checking. Safety regular properties are efficiently verified by tableau automaton based model checking, more complex properties require the more complex SCC-detection algorithm combined with the general Büchi automata representation [2].

6.1.4 Bounded Model Checking

Various bounded model checking algorithms are available in the framework, they are summarised in [Dar17]. They aim to provide analysis support for models with infinite state spaces. Beside the traditional bounded algorithms of [YCL09], the provided algorithms are the *B-I-Sat* and the *compacting* saturation approaches which provide significantly better performance [Dar17]. In addition, more variants of the B-I-Sat algorithm were developed such as *restarting* and *continuing*. Bounded model checking algorithms can be run on ordinary and also Coloured Petri nets, and they can be combined with arbitrary next-state computation/representation strategy.

6.1.5 CEGAR Approach

Counterexample-Guided Abstraction Refinement (CEGAR) is an efficient state space exploration and trace generation technique which has complementary strengths as saturation. CEGAR can easily handle models with infinite state space and efficiently searches for traces by using various reduction techniques. However, when the models are correct, and no counterexample (trace) exists then it often fails to prove the absence of the error states. However, according to the intricate iteration of the Petri net CEGAR algorithm, no theoretical analysis was available aiming the completeness and correctness of the algorithm. Without such investigations, it is impossible to tell when to use CEGAR and when to use other algorithms.

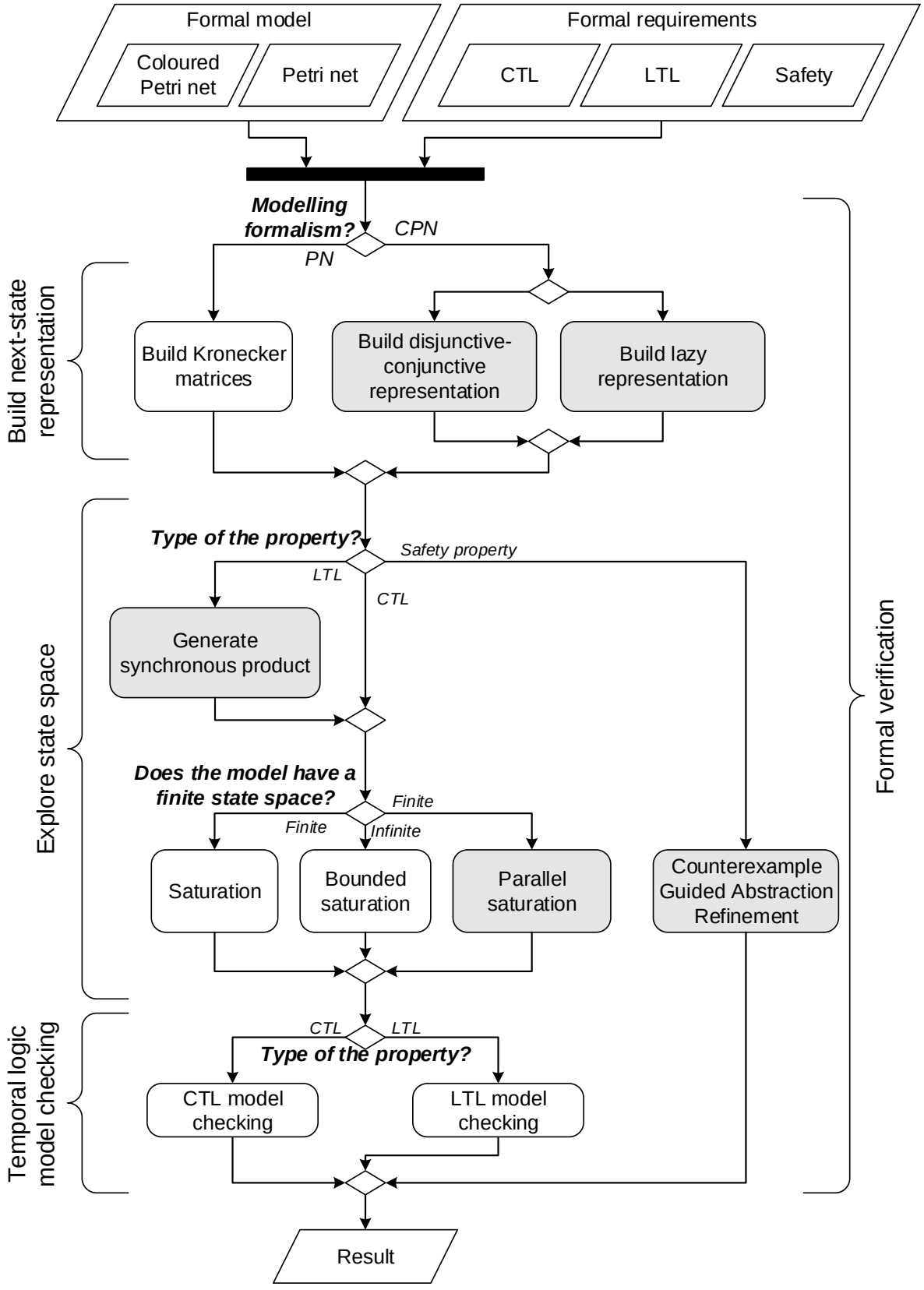


Figure 6.1: High-level view of the proposed verification approach

6.2 Advancing the State-Of-The-Art

In this section, I shortly summarise the ingredients of the novel model checking approach and also my theoretical results regarding the CEGAR step of the workflow.

6.2.1 Configurable Approach for Model Checking Petri Net Models

The approach envisaged on Figure 6.1 was worked out in the PetriDotNet framework. The main advantage of the approach is that it supports the configuration of the algorithms according to the chosen formalisms: different variants of Petri nets require different variants of the saturation algorithm, and also different specification languages need different algorithms to be efficient. The tool contains prepared chains of the algorithms providing the best performance for certain classes of problems. However, these algorithms can be further combined and modified. The provided approaches for each problem domain are the following.

Temporal logic model checking of ordinary Petri nets. Kronecker matrix based next-state representation combined with saturation provides an efficient means for the state space exploration of bounded ordinary Petri nets, while bounded saturation techniques [Dar17] or CEGAR [WW11] can efficiently explore the behaviour of ordinary Petri nets with infinite state spaces. The structural CTL model checking algorithm can be used with arbitrary next-state representations, and it works together even with bounded saturation providing bounded CTL model checking capabilities. Model checking reachability properties of ordinary Petri nets can be solved with CEGAR. Regular safety properties of ordinary and Coloured Petri nets are verified by tableau automaton based model checking algorithm (introduced in Chapter 5) and general LTL model checking (for future and also past LTL) is provided by the framework [2].

Temporal logic model checking of Coloured Petri nets. Disjunctive-conjunctive decomposition based state space exploration for Coloured Petri nets (introduced in Section 3.4) is the basic underlying algorithm: if this algorithm does not succeed then it can be changed to lazy saturation (Section 3.5) which can battle the problem caused by the possibly bigger local state spaces. The CEGAR approach works on ordinary Petri nets, so if we want to apply them on a problem represented by a Coloured Petri net, then it has to be unfolded, which is also provided by the framework.

The structural CTL model checking algorithm can be combined with the disjunctive-conjunctive decomposition algorithm and also lazy saturation. In addition, it works together even with bounded saturation providing bounded CTL model checking capabilities. Regular safety properties and general LTL model checking (for future and also past LTL) is provided as it was discussed in Section 5 and in [2].

Handling infinite state spaces. Various bounded model checking algorithms are present in the framework working on ordinary and also on Coloured Petri net representations. Bounded model checking is provided to find a counterexample or a witness for properties. However, bounded model checking is not complete unless we explored the full state space. For this purpose, I propose to use the Petri net CEGAR approach, which is able also to prove the absence of traces to a certain state. However, this approach lacked theoretical investigation regarding completeness. In the following section I will show that unfortunately, the Petri net CEGAR approach is not complete.

6.2.2 Theoretical Investigation of the Petri Net CEGAR Algorithm

The CEGAR algorithm was introduced in [WW11], where the authors combined the CEGAR approach with the Petri net state equation. However, the completeness of CEGAR highly depends on the systems it is applied to. On one hand, CEGAR provides a convergent iteration strategy for finite state systems. On the other hand, CEGAR may not find a solution when applied to an infinite state system. The Petri net CEGAR approach was not investigated from the completeness and correctness point of view. In this chapter I try to fill this gap and provide theoretical investigation of this question.

6.2.2.1 Completeness Analysis of the Petri Net CEGAR Algorithm

To my best knowledge, the completeness of the algorithm has neither been proved nor disproved yet. When I examined the iteration strategy of the abstraction loop, I found a whole subclass of nets that cannot be solved with this strategy. As an example, consider the Petri net in Figure 6.2 with the reachability problem $(1, 1, 0, 0) \in R(PN, (0, 1, 0, 0))$, i.e., we want to produce a token in p_0. We constructed the net so that the firing sequence $\sigma_s = (t_1, t_4, t_2, t_3, t_3, t_0, t_1, t_2, t_5)$ solves the problem. The main concept of this example is that we lend an extra token in p_1 indirectly using the T-invariant $\{t_4, t_5\}$.

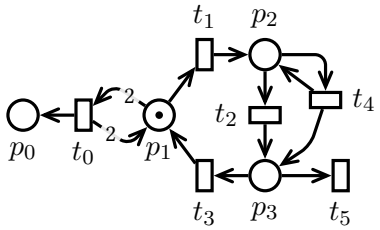


Figure 6.2: Counterexample of completeness

When applying the algorithm on this problem, the minimal solution vector is $x_0 = (1, 0, 0, 0, 0, 0)$, i.e., firing t_0. Since t_0 is not enabled, the only partial solution is $ps_0 = (\emptyset, x_0, \sigma_0 = (), r_0 = (1, 0, 0, 0, 0, 0))$. The algorithm finds that an additional token is required in p_1 and only t_3 can satisfy this need. With an increment constraint $c_1 : |t_3| \geq 1$, the T-invariant $\{t_1, t_2, t_3\}$ is added to the new solution vector $x_1 = (1, 1, 1, 1, 0, 0)$, giving us one partial solution $ps_1 = (\{c_1\}, x_1, \sigma_1 = (t_1, t_2, t_3), r_1 = r_0)$. Firing the invariant $\{t_1, t_2, t_3\}$ does not help getting closer to enabling t_0, since no extra token can be "borrowed" from the previous T-invariant. The iteration strategy of the original algorithm does not recognize the fact that an extra token could be produced in p_3 (using t_4) and then moved in p_1, therefore it cannot decide reachability.

As this example shows, the algorithm can not properly traverse the space of the possible t-invariants. Any extension of this problem will not be solved by the algorithm. In addition, when the complex combination of possible t-invariants should be used to solve the reachability problem, de algorithm will fail to detect the need for additional firings and omit the solution.

6.2.2.2 Extensions to CEGAR

In order to assess the applicability of the CEGAR approach, I have investigated the completeness in Section 6.2.2. This work was continued by my colleagues, who provided results regarding the correctness of the algorithm and major improvements.

The result of the completeness analysis was used and the CEGAR approach was extended to be able to handle a bigger subset of the reachability problems and also to handle inhibitor arcs in [15] [4] [10]. In this work I contributed as the supervisor of the students: Ákos Hajdu and Zoltán Mártonka.

6.3 Tool Support for Usable Formal Methods

In this section, I overview the PetriDotNet tool, which was developed to provide a comprehensive tool support for the introduced approach and algorithms. The tool was developed by our research group, the implementation was done mainly by Dániel Darvas, Vince Molnár, Attila Jámbor, Tamás Szabó, Ákos Hajdu, Kristóf Marussy and Attila Klenik under the supervision of Tamás Bartha and myself.

This overview was formerly published in [1] and [7].

6.3.1 Functionality

This section overviews the main functionality of PetriDotNet and the plug-ins shipped with the tool.

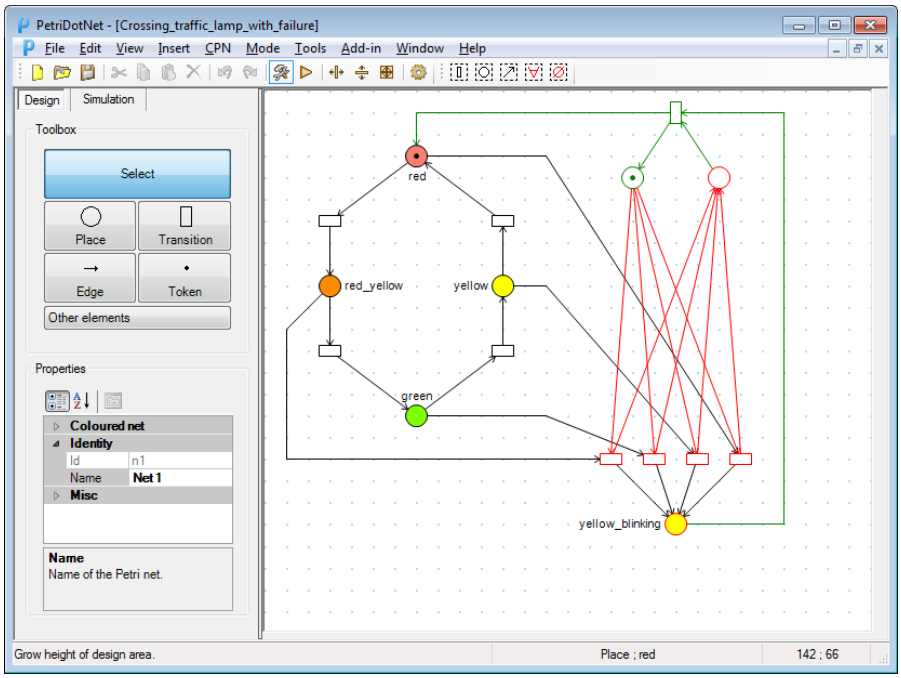


Figure 6.3: The main window of PetriDotNet [1]

6.3.1.1 Editor Features

First and foremost, PetriDotNet is an editor for Petri nets. It provides graphical editing capabilities (cf. Figure 6.3) for both ordinary and well-formed coloured Petri nets (see Section 3.2 for the definition of the supported coloured Petri net variant). The tool supports Petri nets extended with inhibitor arcs, transition priorities, and places with limited token capacity. Moreover, the construction of hierarchical Petri nets is supported by allowing coarse transitions that can be refined by a subnet.

The tool provides simulation functionality (*token game*) for Petri nets, where the simulation can be manually conducted or automatically executed. The tool is shipped with a plug-in that can perform

large-scale simulation, executing thousands or millions of non-deterministic firing, and then present the statistics.

To save and load the Petri nets, PetriDotNet supports two formalisms natively. The default format is PNML (Petri Net Markup Language) [ISO11], a standard, XML-based Petri net description format. PNML is supported by various other tools, therefore this is an interface between these tools and the PetriDotNet framework. A binary, custom file format is also supported that provides more efficient persistence for large models.

6.3.1.2 Plug-in Features

The functionality of the tool is extensible with plug-ins. Plug-ins can perform simulation tasks, provide analysis features (e. g., model checking) or export/import capabilities. Each plug-in can access the Petri net data models, use the graphical user interface, add new menu items, and call built-in PetriDotNet commands. The architecture of the tool is designed to keep the development of plug-ins simple, in order to help the users to focus on functionality instead of technology. See Section 6.3.2 for more details.

6.3.1.3 Export and Import Features

It is possible to export the constructed Petri nets into other Petri net formalisms, such as to the syntax of the GPenSIM[1] (General Purpose Petri Net Simulator) and the .pnt format of the INA[2] (Integrated Net Analyzer) tool. Also, the Petri net models can be translated into to the input format of SAL[3] (Symbolic Analysis Laboratory). Furthermore, import is also provided from the .net textual Petri net file format used by the INA/Tina[4] tools, among others. New import or export plug-ins can be developed easily as the internal model representations are simply accessible.

6.3.1.4 Formal Methods Course Plug-in

As one of the first motivations was to support the education, the framework has built-in support for the following tasks:

- Calculating invariants, and displaying the results right on the Petri net,
- Generating the reachability/coverability graph, and exporting their graphical representation into image files,
- Computing various liveness properties [Mur89].

The invariant analysis covers both P and T-invariants based on the well-known Martinez–Silva algorithm [MS82], and a different algorithm by Cayir and Ucer [CU05] that computes the bases of invariants.

6.3.1.5 Integrated Analysis Methods

In the last five years, in addition to the educational features, PetriDotNet became a Petri net analysis package providing plug-ins for a wide range of analysis methods. Among others, as detailed below, PetriDotNet supports advanced formal verification techniques based on decision diagrams and abstraction. The algorithms of the approach are discussed in earlier sections with some more details.

[1] `http://www.davidrajuh.net/gpensim/`
[2] `http://www2.informatik.hu-berlin.de/lehrstuehle/automaten/ina/`
[3] `http://sal.csl.sri.com/`
[4] `http://projects.laas.fr/tina/`

Saturation-Based Model Checking Algorithms. In PetriDotNet , various algorithms provide model checking based on the saturation algorithm [CMS03; CMS06; CY05; CZJ12]. The CTL model checking approaches are based on the work of Ciardo [ZC09] and the bounded model checking approach is the extension of [YCL09]. The LTL model checking algorithms are built on top of the ideas of [HRS13; Dur+11]. Our research resulted in significant extensions and improvements, this way PetriDotNet currently supports novel analysis algorithms as follows:

- CTL model checking of ordinary and coloured Petri nets based on traditional and extended versions of saturation [5],[16],
- Bounded CTL model checking based on a novel saturation-based algorithm, with various search strategies [6],[3],
- LTL model checking based on a novel synchronous product computation algorithm [2] and incremental SCC detection [11].

CEGAR-Based Reachability Algorithms. PetriDotNet includes reachability analysis algorithms based on Counterexample-Guided Abstraction Refinement (CEGAR) [Cla+00] for ordinary Petri nets. Petri net CEGAR-based algorithms over-approximate the set of reachable states using the state equation, which is a necessary criterion for reachability. The CEGAR algorithm for Petri nets introduced in [WW11] was the base of our work. Our implementation includes various search strategies, adapted to the characteristics of the different models [4],[10].

Stochastic Analysis Algorithms. Recently the tool was extended to support the modelling and analysis of stochastic Petri net models. The goal was to provide a configurable stochastic analysis framework where various state space exploration, matrix representation and numerical analysis algorithms can be combined [9],[7],[8]. PetriDotNet provides the following stochastic analysis for ordinary stochastic Petri net models:

- Steady-state reward and sensitivity analysis,
- Transient reward analysis,
- Calculation of the mean time to reach a state partition, that is used to calculate mean-time-to-first-failure (MTFF) in dependability models.

6.3.2 Architecture

General Architectural Overview. The tool is written in C#, based on the Microsoft .NET framework. The architecture of PetriDotNet is kept as simple as reasonably possible. It is a modular tool: it provides some basic functionalities and can be extended by various plug-ins.

The tool uses a base library defining the Petri net data structures, developed for PetriDotNet . This library contains object models for ordinary and coloured Petri nets. The PetriDotNet core contains the graphical user interface and the plug-in interface. The architecture of the tool is summarized in Figure 6.4.

Plug-in Interface. To follow the previously presented educational goals, it is simple to extend PetriDotNet with a new plug-in. This allows a steep learning curve and low entry barrier, therefore the plug-in developers can focus on their algorithms, instead of the applied technologies. From the tool's point of view, a plug-in is just a .dll file in the `add-in` folder, in which at least one class implements the

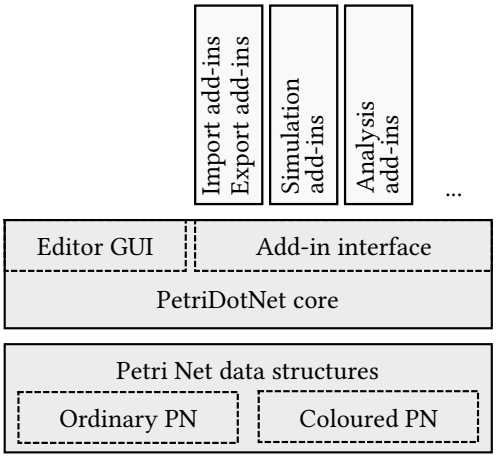


Figure 6.4: High-level overview of the PetriDotNet architecture[1]

`IPDNPlugin` interface (see Figure 6.5). Metadata about the plug-in (e. g., name, author, required PetriDotNet version) can be provided using annotations of this class (e. g., `[AddinAuthor("X. Y.")]`). When PetriDotNet starts, it loads all plug-ins and calls their `Initialize` method. In this method, the plug-ins can make their menu contributions and store the application descriptor. This latter allows the plug-ins to call commands (e. g., save, load) and to access the currently active Petri net.

Being a .NET-based tool, PetriDotNet requires that the plug-ins be also implemented in one of the .NET languages. While having a graphical editor for Petri nets developed in .NET is a reasonable choice, implementing e. g., model checking algorithms seems to be uncommon, as managed languages are considered to have some overhead. However, *(i)* according to our experience the runtime of the .NET-based implementations of various model checking algorithms proved to be competitive compared to their native version, and *(ii)* the development in .NET is easier and less error-prone than e. g., in C or C++ for computer engineering students, allowing them to make correct implementations in a shorter time. Thus the choice of .NET can be regarded as sacrificing some runtime performance in favour of development time, which is similarly important in our educational setting. If the performance needs cannot be satisfied using .NET, the plug-in can wrap or depend on a native implementation (.dll).

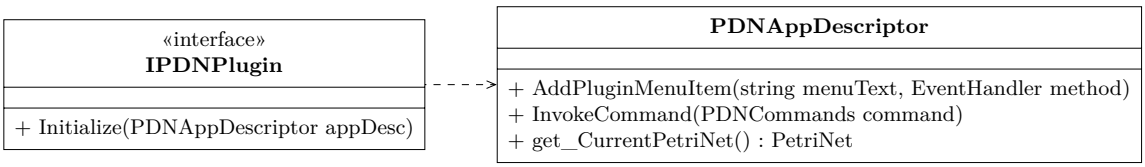


Figure 6.5: PetriDotNet plug-ins[7]

6.3.3 Use Cases

This section overviews the use cases where we applied PetriDotNet as an editor or an analysis tool. According to the original goals we start the overview with educational use cases, then we move on to industrial case studies.

Application in Education. PetriDotNet is used as an educational tool and a tool for the homework assignments in the Formal Methods course of the Budapest University of Technology and Economics since 2011. During this time, approximately 900 M.Sc. students attended the course. The stochastic analysis module of the tool is used for demonstration purposes in the Software and Systems Verification course to teach reliability modelling for the students.

Student Projects. To this day 23 B.Sc. and M.Sc. theses were written that applied or extended PetriDotNet and various student projects used it to get an insight to formal methods. Besides, the various new formal verification algorithms resulted in 20 scientific papers presented at conferences or journals[5]. Several students who started to get familiar with research by extending and implementing an algorithm in PetriDotNet are now Ph.D. students or planning to apply for post-graduate programmes. So far one Ph.D. was awarded for a topic related to PetriDotNet [Dar17].

Application in Industrial Cases. We are aware of various usages of our tool to model, simulate and analyse different real-life systems.

- We have applied PetriDotNet to model and formally verify a safety logic of the Paks Nuclear Power Plant using saturation-based CTL model checking in [5],[16] (Section 3.6). This work validated the coloured Petri net editing capabilities and proved the efficiency of our CTL model checking algorithms, as [16] presented the first successful formal verification of the complete safety logic.
- PetriDotNet was used to model and simulate sensor nets in [Mil+14] and in the FuturICT.hu project[6].
- PetriDotNet was applied to model and study railway interlocking systems [CTS14].
- The R3-COP project[7] applied PetriDotNet to generate test input sequences for testing the robustness of communicating autonomous robots [DV13].
- Initial case studies were made to apply PetriDotNet to analyse control software used at the European Organization for Nuclear Research (CERN) [DFB13].
- Stochastic analysis and MTFF computation were used in an industrial project at our department to evaluate safety (hazard rate) of an embedded control system. The mean time to reach undetected failures or shutdown was computed in a stochastic model of a two-channel architecture with separate diagnostic facility, comparison, and time-limited degraded (single-channel) functionality.

6.4 Thesis 4: PetriDotNet Model Checking Framework

The usability of the developed algorithms cannot be achieved without tool support. I have investigated many existing tools and approaches and evaluated them amongst others on the industrial problem of the PRISE safety system. According to the experiences, I chose to use Petri nets as a simple formalism and Coloured Petri nets as a convenient formalism to develop high-level models. However, the available tool support for the verification of Petri nets was weak in the sense that either good editor and tooling were available or advanced verification algorithms. However, the verification problem

[5] See the complete list of related publications at `http://petridotnet.inf.mit.bme.hu/publications/`.
[6] `http://www.futurict.szte.hu/en/home/`
[7] `http://www.r3-cop.eu/`

is difficult in general so no single algorithm or approach can be efficient on their own. Therefore I developed a model checking workflow to combine the various advantages of the different algorithms and approaches. The novel combination of the algorithms was implemented in the PetriDotNet model checking framework. Theoretical examination of the algorithms was needed in order to extend them and combine their strength in a framework. The proposed model checking framework addressed the problem of modelling safety-critical systems with high-level modelling languages, specifying the requirements with the help of temporal logics and verify finite state and infinite state models with various algorithms.

Thesis 4 *I worked out an approach for the modelling and verification of complex systems. We developed a framework to support the Coloured Petri net based modelling and verification of complex systems. The framework provides CTL and LTL model checking based on novel algorithms. In order to extend the handled classes of models, infinite state and trace generation algorithms were integrated. We extended the model checking algorithms to be able to handle infinite state systems by applying bounded model checking and a special algorithm based on Counterexample-Guided Abstraction Refinement (CEGAR). The latter algorithm provides traces as a feedback for the developers. I did theoretical investigations, and I examined the CEGAR algorithm from the completeness point of view: I proved the incompleteness of the CEGAR-based Petri net reachability algorithm.*

This thesis encapsulates the various results together into a framework supporting the engineers developing correct systems. A theoretical analysis was elaborated on a well-known algorithm, and its applicability for trace generation was examined. This opens new directions for further developments in the future[10]. I envisioned and designed the PetriDotNet model checking framework where we could successfully integrate the research results of the participants of the research group and students supervised by myself.

Publications: My new results introduced in this thesis were published in the journal papers [4] and [1] and in the following conference papers: [15] and [7].

Chapter 7

Conclusion and future work

7.1 Summary of the research results

In this section, I summarise the challenges of my work (also discussed in Section 1.3.4.1) and also my solutions for each problem.

Challenge 1: Verification of complex systems High-level modelling languages are needed to model complex systems. High-level models of complex systems require rigorous verification techniques, so the existing verification approaches and algorithms have to be extended to overcome the challenges.

Solution in Thesis 1.: *Chapter 3 presented novel techniques for the verification of high-level system models. At first, I propose to use Coloured Petri nets as an expressive, high-level modelling language being able to capture the behaviour of asynchronous, distributed safety-critical systems. I investigated the verification algorithms in the literature especially symbolic algorithms such as saturation. I identified that existing algorithms are not efficient for CPNs due to the high expressiveness of the language. This motivated my research to develop a new model checking algorithm, which can efficiently represent the symbolic next-state relations. I introduced two new symbolic decomposition and next-state computation algorithms for the verification of Coloured Petri nets: by using the disjunctive-conjunctive decomposition algorithm we could successfully verify a safety function of an industrial case study. The second algorithm could further extend the applicability of the approach by decreasing the computational complexity of the verification algorithm for CPN models with large variable domains.*

Challenge 2: Increase the efficiency of model checking algorithms New techniques are needed to increase the efficiency of model checking algorithms and decrease runtime requirements.

Solution in Thesis 2.: *Parallelization is a common approach to improve the performance of algorithms. However, saturation is inherently sequential, so it is difficult to parallelise [CZJ09]. Former attempts [ELS06] faced difficulties, and they could not reach significant performance gains. In my work in Chapter 4 I investigated the former parallel saturation approaches, and I identified the causes of the poor scalability of the algorithm. According to the insights I gained during the investigations, I propose to use a more lightweight synchronisation mechanism. I devised a new locking and synchronisation strategy to reduce the synchronisation overhead. We did extensive measurements to compare the new algorithm to its competitor regarding scalability, and the introduced new approach proved its efficiency.*

Challenge 3. Verification support for various requirement specification languages Research and industrial case studies revealed the need for a wide range of specification languages to support the various types of requirements of the use-cases.

Solution in Thesis 3.: *CTL and LTL temporal logics have different strength and weaknesses, so it is important for a model checker to support both formalisms from the usability point of view. In Chapter 6 I introduced a complex model checking approach which provides efficient verification techniques for various temporal logic-based specification languages. As the efficiency of saturation was mainly exploited in CTL model checking, in Chapter 5 I propose an approach for the computation of the synchronous product of a property automaton and the state space representation on-the-fly during the exploration. This is the essential step of automata-theoretic model checking and based on the novel synchronous product computation algorithm, I propose a model checking algorithm for the set of regular safety properties.*

Challenge 4: Formal modelling and verification framework. The wide range of industrial problems necessitates a formal modelling and verification framework with various modelling languages and verification algorithms. As no single formalism or algorithm can support the many aspects of the use-cases, a configurable framework is needed, which can be fine-tuned to handle the verification problems.

Solution in Thesis 4.: *The main goal of my work was to introduce an approach which is able to support the verification engineers in the development and analysis of verification models. I introduced a comprehensive verification approach in Chapter 6 and I put its pieces together. The approach uses saturation-based symbolic algorithms and abstraction based techniques to support all aspects of the verification problem. In addition, I investigated one key component, namely the Petri net CEGAR approach, and I found that this algorithm is not complete. In Chapter 6, I also provided a simple proof for the incompleteness of the CEGAR algorithm.*

The introduced model checking algorithm manifested in the PetriDotNet framework proved its applicability in many research and industrial projects, also discussed in this thesis.

7.2 Future work

Beside the results, my work also opened new questions and research directions to be explored. I was very fortunate to work with many students who will continue this line of research.

Incremental model checking. The results presented in Chapter 5 were the first steps towards efficient model checking of LTL properties, which was then extended later to provide efficient model checking capabilities for the general class of LTL properties. A promising direction was revealed during our research: by exploiting the locality of saturation, one could build an incremental model checking algorithm by tracking the changes of the models and using saturation to adjust the verification results according to the changes. This could lead to faster verification results and decreased response time. My colleague, (my former student) Vince Molnár is working on this direction in his PhD studies.

CTL* model checking. The introduced approach supports both CTL and LTL model checking. However, CTL^* is a more general specification language, which is supported yet neither by the PetriDotNet framework and also nor by other model checkers, due to complexity reasons. It could be a

potential research direction to combine the strengths of the CTL and LTL model checking algorithms used in the PetriDotNet framework and provide CTL^* model checking capabilities.

Improve the Petri net CEGAR approach. I proved the incompleteness of the approach, and it revealed many new research directions. The CEGAR approach could be further improved by combining its forward iteration strategy with backward search algorithms. This way I am sure that we could further extend the verification capabilities of the algorithm. This research line is continued by my colleague (my former student), Ákos Hajdu.

Exploit saturation for stochastic analysis. In the PetriDotNet framework, we now provide basic stochastic analysis capabilities. However, in the future, it could be further improved by using more information from the decision diagram encoding. Other research teams produced some initial results, and with my students, we are also working on this direction now actively.

Combining POR with saturation. Saturation is efficient for asynchronous systems. However, traditional algorithms used Partial Order Reduction (POR) techniques to combat the state space explosion problem. An interesting future work is to combine the two approaches. This will need much theoretical investigation. We did the first steps into this direction with our student in a Scientific Student Association report [ÉS15].

Bibliography

Publication list

Journal papers

[1] A. Vörös, D. Darvas, Á. Hajdu, A. Klenik, K. Marussy, V. Molnár, T. Bartha, and I. Majzik. "Industrial Applications of the PetriDotNet Modelling and Analysis Tool". In: *Science of Computer Programming* (2017). In press. ISSN: 0167-6423. DOI: 10.1016/j.scico.2017.09.003

[2] V. Molnár, A. Vörös, D. Darvas, T. Bartha, and I. Majzik. "Component-wise Incremental LTL Model Checking". In: *Formal Aspects of Computing* 28.3 (2016), pp. 345–379. ISSN: 0934-5043. DOI: 10.1007/s00165-015-0347-x

[3] D. Darvas, A. Vörös, and T. Bartha. "Improving Saturation-based Bounded Model Checking". In: *Acta Cybernetica* 22.3 (2016), pp. 573–589. ISSN: 0324-721X. DOI: 10.14232/actacyb.22.3.2016.2

[4] Á. Hajdu, A. Vörös, T. Bartha, and Z. Mártonka. "Extensions to the CEGAR Approach on Petri Nets". In: *Acta Cybernetica* 21.3 (2014), pp. 401–417. DOI: 10.14232/actacyb.21.3.2014.8

[5] A. Vörös, D. Darvas, A. Jámbor, and T. Bartha. "Advanced Saturation-based Model Checking of Well-formed Coloured Petri Nets". In: *Periodica Polytechnica, Electrical Engineering and Computer Science* 58.1 (2014), pp. 3–13. ISSN: 2064-5279. DOI: 10.3311/PPee.2080

[6] A. Vörös, D. Darvas, and T. Bartha. "Bounded saturation-based CTL model checking". In: *Proceedings of the Estonian Academy of Sciences* 62.1 (2013), pp. 59–70. ISSN: 1736-6046. DOI: 10.3176/proc.2013.1.07

International conference and workshop papers

[7] A. Vörös, D. Darvas, V. Molnár, A. Klenik, Á. Hajdu, A. Jámbor, T. Bartha, and I. Majzik. "PetriDotNet 1.5: Extensible Petri Net Editor and Analyser for Education and Research". In: *Application and Theory of Petri Nets and Concurrency*. Ed. by F. Kordon and D. Moldt. Vol. 9698. Lecture Notes in Computer Science. Springer, 2016, pp. 123–132. ISBN: 978-3-319-39086-4. DOI: 10.1007/978-3-319-39086-4_9

[8] K. Marussy, A. Klenik, V. Molnár, A. Vörös, I. Majzik, and M. Telek. "Efficient decomposition algorithm for stationary analysis of complex stochastic Petri net models". In: *Application and Theory of Petri Nets and Concurrency*. Ed. by F. Kordon and D. Moldt. Vol. 9698. Lecture Notes in Computer Science. Springer, 2016, pp. 281–300. ISBN: 978-3-319-39086-4. DOI: 10.1007/978-3-319-39086-4_17

[9] K. Marussy, A. Klenik, V. Molnár, A. Vörös, M. Telek, and I. Majzik. "Configurable Numerical Analysis for Stochastic Systems". In: *Proceedings of the 2016 Workshop on Symbolic and Nu-*

merical Methods for Reachability Analysis (SNR). ed. by E. Ábrahám and S. Bogomolov. Vienna, Austria: IEEE, 2016. ISBN: 978-1-5090-3079-8. DOI: 10.1109/SNR.2016.7479383

[10] Á. Hajdu, A. Vörös, and T. Bartha. "New search strategies for the Petri net CEGAR approach". In: *Application and Theory of Petri Nets and Concurrency*. Ed. by R. Devillers and A. Valmari. Vol. 9115. Lecture Notes in Computer Science. Springer, 2015, pp. 309–328. ISBN: 978-3-319-19488-2. DOI: 10.1007/978-3-319-19488-2_16

[11] V. Molnár, D. Darvas, A. Vörös, and T. Bartha. "Saturation-Based Incremental LTL Model Checking with Inductive Proofs". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C. Baier and C. Tinelli. Vol. 9035. Lecture Notes in Computer Science. Springer, 2015, pp. 643–657. ISBN: 978-3-662-46680-3. DOI: 10.1007/978-3-662-46681-0_58

[12] D. Darvas, B. Fernández Adiego, A. Vörös, T. Bartha, E. Blanco Viñuela, and V. M. González Suárez. "Formal verification of complex properties on PLC programs". In: *Formal Techniques for Distributed Objects, Components, and Systems*. Ed. by E. Ábrahám and C. Palamidessi. Vol. 8461. Lecture Notes in Computer Science. Springer, 2014, pp. 284–299. ISBN: 978-3-662-43612-7. DOI: 10.1007/978-3-662-43613-4_18

[13] Z. Micskei, R.-A. Konnerth, B. Horváth, O. Semeráth, A. Vörös, and D. Varró. "On Open Source Tools for Behavioral Modeling and Analysis with fUML and Alf". In: *Proceedings of the 1st Workshop on Open Source Software for Model Driven Engineering*. Ed. by F. Bordelau, J. Dingel, S. Gerard, and S. Voss. Valencia, Spain, Sept. 2014, pp. 31–41

[14] D. Darvas, A. Vörös, and T. Bartha. "Efficient Saturation-based Bounded Model Checking of Asynchronous Systems". In: *Proceedings of the 13th Symposium on Programming Languages and Software Tools, SPLST'13*. Ed. by Á. Kiss. Szeged, Hungary: University of Szeged, 2013, pp. 259–273. ISBN: 978-963-306-228-9

[15] Á. Hajdu, A. Vörös, T. Bartha, and Z. Mártonka. "Extensions to the CEGAR Approach on Petri Nets". In: *Proceedings of the 13th Symposium on Programming Languages and Software Tools, SPLST'13*. Ed. by Á. Kiss. Szeged, Hungary: University of Szeged, 2013, pp. 274–288. ISBN: 978-963-306-228-9

[16] T. Bartha, A. Vörös, A. Jámbor, and D. Darvas. "Verification of an Industrial Safety Function Using Coloured Petri Nets and Model Checking". In: *Proceedings of the 14th International Conference on Modern Information Technology in the Innovation Processes of the Industrial Enterprises (MITIP 2012)*. Ed. by E. Ilie-Zudor, Z. Kemény, and L. Monostori. Budapest, Hungary: Hungarian Academy of Sciences, Computer and Automation Research Institute, 2012, pp. 472–485. ISBN: 978-963-311-373-8

[17] A. Vörös, T. Bartha, D. Darvas, T. Szabó, A. Jámbor, and Á. Horváth. "Parallel Saturation Based Model Checking". In: *Proceedings of the 10th International Symposium on Parallel and Distributed Computing (ISPDC)*. Cluj Napoca, Romania: IEEE Computer Society, 2011, pp. 94–101. ISBN: 978-1-4577-1536-5. DOI: 10.1109/ISPDC.2011.23

[18] A. Vörös, D. Darvas, and T. Bartha. "Bounded Saturation Based CTL Model Checking". In: *Proceedings of the 12th Symposium on Programming Languages and Software Tools, SPLST'11*. Ed. by J. Penjam. Tallinn, Estonia: Tallinn University of Technology, Institute of Cybernetics, 2011, pp. 149–160. ISBN: 978-9949-23-178-2

Local conference and workshop papers

[19] Á. Hajdu, R. Német, S. Varró-Gyapay, and A. Vörös. "Petri Net Based Trajectory Optimization". In: *ASCONIKK 2014: Extended Abstracts. Future Internet Services.* Veszprém, Hungary: University of Pannonia, 2014, pp. 11–19

[20] V. Molnár and A. Vörös. "Synchronous Product Automaton Generation for Controller Optimization". In: *ASCONIKK 2014: Extended Abstracts. I. Information Technologies for Logistic Systems.* Veszprém, Hungary: University of Pannonia, 2014, pp. 22–29. ISBN: 978-963-396-046-2

[21] D. Darvas and A. Vörös. "Szaturációalapú tesztbemenet-generálás színezett Petri-hálókkal [in Hungarian]". In: *Mesterpróba 2013. Konferenciakiadvány.* Budapest, Hungary, 2013, pp. 48–51

[22] A. Vörös. "Modellellenőrzés alkalmazása egy biztonságkritikus rendszer védelmi logikájának verifikációjára [in Hungarian]". In: *XVII. Fiatal Műszakiak Tudományos Ülésszaka.* Cluj Napoca, Romania: Erdélyi Múzeum-Egyesület Műszaki Tudományok Szakosztálya, 2012, pp. 383–386

[23] A. Vörös. "Forward Saturation Based Model Checking". In: *Proceedings of the 19th PhD Minisymposium of the Department of Measurement and Information Systems.* Budapest, Hungary, 2012, pp. 38–41

[24] A. Vörös. "Optimizing Saturation Based Model Checking". In: *Proceedings of the 18th PhD Minisymposium of the Department of Measurement and Information Systems.* Budapest, Hungary, 2011, pp. 96–99

References

[Adi+15] B. F. Adiego, D. Darvas, E. B. Viñuela, J. C. Tournier, S. Bliudze, J. O. Blech, and V. M. G. Suárez. "Applying Model Checking to Industrial-Sized PLC Programs". In: *IEEE Transactions on Industrial Informatics* 11.6 (Dec. 2015), pp. 1400–1410. ISSN: 1551-3203. DOI: 10.1109/TII.2015.2489184.

[Ana+13] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. McMinn. "An orchestrated survey of methodologies for automated software test case generation". In: *Journal of Systems and Software* 86.8 (2013), pp. 1978–2001. ISSN: 0164-1212. DOI: 10.1016/j.jss.2013.02.061.

[Bey17] D. Beyer. "Software Verification with Validation of Results". In: *Tools and Algorithms for the Construction and Analysis of Systems.* Ed. by A. Legay and T. Margaria. Berlin, Heidelberg: Springer, 2017, pp. 331–349. ISBN: 978-3-662-54580-5. DOI: 10.1007/978-3-662-54580-5_20.

[BKL08] C. Baier, J.-P. Katoen, and K. G. Larsen. *Principles of model checking.* MIT press, 2008.

[BKP17] H. Bride, O. Kouchnarenko, and F. Peureux. "Reduction of Workflow Nets for Generalised Soundness Verification". In: *Verification, Model Checking, and Abstract Interpretation: 18th International Conference, VMCAI 2017, Paris, France, January 15–17, 2017, Proceedings.* Ed. by A. Bouajjani and D. Monniaux. Cham: Springer, 2017, pp. 91–111. ISBN: 978-3-319-52234-0. DOI: 10.1007/978-3-319-52234-0_6.

[BP12] D. Beyer and A. K. Petrenko. "Linux Driver Verification". In: *Leveraging Applications of Formal Methods, Verification and Validation. Applications and Case Studies: 5th International Symposium, ISoLA 2012, Heraklion, Crete, Greece, October 15-18, 2012, Proceedings, Part II.* Ed. by T. Margaria and B. Steffen. Berlin, Heidelberg: Springer, 2012, pp. 1–6. ISBN: 978-3-642-34032-1. DOI: 10.1007/978-3-642-34032-1_1.

[Bra+11] A. R. Bradley, F. Somenzi, Z. Hassan, and Y. Zhang. "An incremental approach to model checking progress properties". In: *Proc. of The Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD'11)*. FMCAD Inc., 2011, pp. 144–153. ISBN: 978-0-9835678-1-3.

[Brü93] A. Brüggemann-Klein. "Regular expressions into finite automata". In: *Theoretical Computer Science* 120.2 (1993), pp. 197–213.

[Buc+00] P. Buchholz, G. Ciardo, S. Donatelli, and P. Kemper. "Complexity of Memory-Efficient Kronecker Operations with Applications to the Solution of Markov Models". In: *INFORMS J. on Computing* 12.3 (July 2000), pp. 203–222. ISSN: 1526-5528. DOI: 10.1287/ijoc.12.3.203.12634.

[Bur+92] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. "Symbolic model checking: 10^{20} States and beyond". In: *Information and Computation* 98.2 (1992), pp. 142–170. ISSN: 0890-5401. DOI: 10.1016/0890-5401(92)90017-A.

[Cal+15] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez. "Moving Fast with Software Verification". In: *NASA Formal Methods: 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings*. Ed. by K. Havelund, G. Holzmann, and R. Joshi. Cham: Springer, 2015, pp. 3–11. ISBN: 978-3-319-17524-9. DOI: 10.1007/978-3-319-17524-9_1.

[Cav+14] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta. "The nuXmv Symbolic Model Checker". In: *Computer-Aided Verification*. Ed. by A. Biere and R. Bloem. Vol. 8559. Lecture Notes in Computer Science. Springer, 2014, pp. 334–342. ISBN: 978-3-319-08866-2.

[CES86] E. Clarke, E. A. Emerson, and A. P. Sistla. "Automatic Verification of Finite-state Concurrent Systems Using Temporal Logic Specifications". In: *ACM Trans. Program. Lang. Syst.* 8.2 (Apr. 1986), pp. 244–263. ISSN: 0164-0925. DOI: 10.1145/5397.5399.

[CGH97] E. Clarke, O. Grumberg, and K. Hamaguchi. "Another Look at LTL Model Checking". In: *Formal Methods in System Design* 10.1 (1997), pp. 47–71. ISSN: 0925-9856. DOI: 10.1023/A:1008615614281.

[CGP99] E. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[Cia+03] G. Ciardo, R. L. Jones, A. S. Miner, and R. Siminiceanu. "Logical and stochastic modeling with SMART". In: *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*. Springer. 2003, pp. 78–97.

[Cla+00] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. "Counterexample-guided abstraction refinement". In: *Computer-Aided Verification*. Ed. by E. A. Emerson and A. P. Sistla. Vol. 1855. LNCS. Springer, 2000, pp. 154–169. ISBN: 978-3-540-67770-3. DOI: 10.1007/10722167_15.

[CMS03] G. Ciardo, R. Marmorstein, and R. Siminiceanu. "Saturation Unbound". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2003, pp. 379–393.

[CMS05] G. Ciardo, R. Marmorstein, and R. Siminiceanu. "The saturation algorithm for symbolic state-space exploration". In: *International Journal on Software Tools for Technology Transfer* 8.1 (Nov. 2005), p. 4. ISSN: 1433-2787. DOI: 10.1007/s10009-005-0188-7.

[CMS06] G. Ciardo, R. Marmorstein, and R. Siminiceanu. "The saturation algorithm for symbolic state-space exploration". In: *Int. J. Softw. Tools Technol. Transf.* 8.1 (2006), pp. 4–25. ISSN: 1433-2779.

[CS03] G. Ciardo and R. Siminiceanu. "Structural symbolic CTL model checking of asynchronous systems". In: *Computer-Aided Verification*. Vol. 3. Springer. 2003, pp. 40–53.

[Cse+02] G. Csertan, G. Huszerl, I. Majzik, Z. Pap, A. Pataricza, and D. Varro. "VIATRA - visual automated transformations for formal verification and validation of UML models". In: *Proceedings 17th IEEE International Conference on Automated Software Engineering*, 2002, pp. 267–270. DOI: 10.1109/ASE.2002.1115027.

[CTS14] A. Cseh, G. Tarnai, and B. Sághi. "Petri Net Modelling of Signalling Systems [in Hungarian, original title: Biztosítóberendezések modellezése Petri-hálókkal]". In: *Vezetékek Világa* XIX.1 (2014), pp. 14–17. ISSN: 1416-1656.

[CU05] S. Cayir and M. Ucer. "An Algorithm to Compute a Basis of Petri Net Invariants". In: *4th ELECO Int. Conf. on Electrical and Electronics Engineering*. Bursa, Turkey: UCTEA, 2005.

[CY05] G. Ciardo and A. Yu. "Saturation-based symbolic reachability analysis using conjunctive and disjunctive partitioning". In: *Correct Hardware Design and Verification Methods* 3725 (2005), pp. 146–161.

[CZJ09] G. Ciardo, Y. Zhao, and X. Jin. "Parallel symbolic state-space exploration is difficult, but what is the alternative?" In: *arXiv preprint arXiv:0912.2785* (2009).

[CZJ12] G. Ciardo, Y. Zhao, and X. Jin. "Ten Years of Saturation: A Petri Net Perspective". In: *Transactions on Petri Nets and Other Models of Concurrency V*. Ed. by K. Jensen, S. Donatelli, and J. Kleijn. Vol. 6900. Lecture Notes in Computer Science. Springer, 2012, pp. 51–95. ISBN: 978-3-642-29071-8.

[Dar17] D. Darvas. "Practice-Oriented Formal Methods to Support the Software Development of Industrial Control Systems". PhD thesis. Budapest University of Technology and Economics, 2017. DOI: 10.5281/zenodo.162950.

[DFB13] D. Darvas, B. Fernández Adiego, and E. Blanco Viñuela. *Transforming PLC programs into formal models for verification purposes*. Internal Note CERN-ACC-NOTE-2013-0040. CERN, 2013.

[DKW08] V. D'Silva, D. Kroening, and G. Weissenbacher. "A Survey of Automated Techniques for Formal Software Verification". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27.7 (July 2008), pp. 1165–1178. ISSN: 0278-0070. DOI: 10.1109/TCAD.2008.923410.

[DMB16] D. Darvas, I. Majzik, and E. Blanco Viñuela. "Formal Verification of Safety PLC Based Control Software". In: *Integrated Formal Methods: 12th International Conference, IFM 2016, Reykjavik, Iceland, June 1-5, 2016, Proceedings*. Ed. by E. Ábrahám and M. Huisman. Cham: Springer, 2016, pp. 508–522. ISBN: 978-3-319-33693-0. DOI: 10.1007/978-3-319-33693-0_32.

[DT97] G. B. Dantzig and M. N. Thapa. *Linear programming 1: introduction*. Secaucus, NJ, USA: Springer, 1997. ISBN: 0-387-94833-3.

[Dur+11] A. Duret-Lutz, K. Klai, D. Poitrenaud, and Y. Thierry-Mieg. "Self-Loop Aggregation Product – A New Hybrid Approach to On-the-Fly LTL Model Checking". In: *Automated Technology for Verification and Analysis*. Vol. 6996. Lecture Notes in Computer Science. Springer, 2011, pp. 336–350. DOI: 10.1007/978-3-642-24372-1_24.

[EH00] K. Etessami and G. J. Holzmann. "Optimizing Büchi automata". In: *CONCUR 2000 – Concurrency Theory*. Vol. 1877. Lecture Notes in Computer Science. Springer, 2000, pp. 153–168.

[EL86] E. A. Emerson and C.-L. Lei. "Efficient Model Checking in Fragments of the Propositional Mu-Calculus (Extended Abstract)". In: *Proc. of the Symposium on Logic in Computer Science (LICS)*. IEEE Computer Society, 1986, pp. 267–278.

[ELS06] J. Ezekiel, G. Lüttgen, and R. Siminiceanu. "Can Saturation Be Parallelised?" In: *International Workshop on Formal Methods for Industrial Critical Systems*. Springer. 2006, pp. 331–346.

[EM00] J. Esparza and S. Melzer. "Verification of Safety Properties Using Integer Programming: Beyond the State Equation". In: *Formal Methods in System Design* 16.2 (Mar. 2000), pp. 159–189. ISSN: 1572-8102. DOI: 10.1023/A:1008743212620.

[EN94] J. Esparza and M. Nielsen. "Decidability issues for Petri nets". In: *BRICS Report Series* 1.8 (1994).

[ERV02] J. Esparza, S. Römer, and W. Vogler. "An Improvement of McMillan's Unfolding Algorithm". In: *Formal Methods in System Design* 20.3 (May 2002), pp. 285–310. ISSN: 1572-8102. DOI: 10.1023/A:1014746130920.

[ÉS15] D. Élő and A. Soltész. *Symbolic model checking and trace generation by guided search*. 1st prize. 2015.

[GA14] S. J. Galler and B. K. Aichernig. "Survey on test data generation tools". In: *International Journal on Software Tools for Technology Transfer* 16.6 (Nov. 2014), pp. 727–751. ISSN: 1433-2787. DOI: 10.1007/s10009-013-0272-3.

[Ger+95] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. "Simple on-the-fly automatic verification of linear temporal logic". In: *Proc. of the Int. Symp. on Protocol Specification, Testing and Verification*. Chapman & Hall, Ltd., 1995, pp. 3–18. ISBN: 0-412-71620-8.

[GZF12] S. Goldshtein, D. Zurbalev, and I. Flatow. *Pro .NET Performance*. Apress, 2012. ISBN: 978-1-4302-4458-5. DOI: 10.1007/978-1-4302-4459-2.

[HRS13] M. Heiner, C. Rohr, and M. Schwarick. "MARCIE – Model Checking and Reachability Analysis Done Efficiently". In: *Petri Nets 2013*. Ed. by J.-M. Colom and J. Desel. Vol. 7927. LNCS. Springer, 2013, pp. 389–399. ISBN: 978-3-642-38696-1. DOI: 10.1007/978-3-642-38697-8_21.

[HVH10] S. Hoda, W.-J. Van Hoeve, and J. N. Hooker. "A systematic approach to MDD-based constraint programming". In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2010, pp. 266–280.

[Ins10] Institute of Electrical and Electronics Engineers. "Systems and software engineering – Vocabulary". In: *ISO/IEC/IEEE 24765:2010(E)* (Dec. 2010), pp. 1–418. DOI: 10.1109/IEEESTD.2010.5733835.

[IS94] J. Izquierdo-Rocha and M. Sánchez-Perea. "Application of the Integrated Safety Assessment methodology to the emergency procedures of a SGTR of a PWR". In: *Reliability Engineering and System Safety* 45 (1994), pp. 159–173.

[ISO11] ISO/IEC. *15909-2:2011, Systems and software engineering – High-level Petri nets – Part 2: Transfer format*. Standard. ISO/IEC, 2011.

[JK09] K. Jensen and L. M. Kristensen. *Coloured Petri Nets - Modelling and Validation of Concurrent Systems*. Springer, 2009. ISBN: 978-3-642-00283-0.

[JKW07] K. Jensen, L. M. Kristensen, and L. Wells. "Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems". In: *International Journal on Software Tools for Technology Transfer* 9.3 (2007), pp. 213–254. ISSN: 1433-2787. DOI: 10.1007/s10009-007-0038-x.

[Kai+09] R. Kaivola et al. "Replacing Testing with Formal Verification in Intel CoreTM i7 Processor Execution Engine Validation". In: *Computer-Aided Verification*. Ed. by A. Bouajjani and O. Maler. Berlin, Heidelberg: Springer, 2009, pp. 414–429. ISBN: 978-3-642-02658-4. DOI: 10.1007/978-3-642-02658-4_32.

[Kan+15] G. Kant, A. Laarman, J. Meijer, J. van de Pol, S. Blom, and T. van Dijk. "LTSmin: High-Performance Language-Independent Model Checking". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C. Baier and C. Tinelli. Berlin, Heidelberg: Springer, 2015, pp. 692–707. ISBN: 978-3-662-46681-0. DOI: 10.1007/978-3-662-46681-0_61.

[Kes+93] Y. Kesten, Z. Manna, H. McGuire, and A. Pnueli. "A Decision Algorithm for Full Propositional Temporal Logic". In: *Computer Aided Verification*. Vol. 697. Lecture Notes in Computer Science. Springer, 1993, pp. 97–109. ISBN: 3-540-56922-7. DOI: 10.1007/3-540-56922-7_9.

[Kle+09] G. Klein et al. "seL4: Formal Verification of an OS Kernel". In: *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*. SOSP '09. Big Sky, Montana, USA: ACM, 2009, pp. 207–220. ISBN: 978-1-60558-752-3. DOI: 10.1145/1629575.1629596.

[Lip76] R. Lipton. *The Reachability Problem Requires Exponential Space*. Research report, Yale University, Dept. of Computer Science. 1976.

[LMM99] D. Latella, I. Majzik, and M. Massink. "Automatic Verification of a Behavioural Subset of UML Statechart Diagrams Using the SPIN Model-checker". In: *Formal Aspects of Computing* 11.6 (Dec. 1999), pp. 637–664. ISSN: 1433-299X. DOI: 10.1007/s001659970003.

[LS09] D. Leinenbach and T. Santen. "Verifying the Microsoft Hyper-V Hypervisor with VCC". In: *Proceedings of the 2Nd World Congress on Formal Methods*. FM '09. Eindhoven, The Netherlands: Springer, 2009, pp. 806–809. ISBN: 978-3-642-05088-6. DOI: 10.1007/978-3-642-05089-3_51.

[May81] E. W. Mayr. "An algorithm for the general Petri net reachability problem". In: *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing*. STOC '81. Milwaukee, Wisconsin, United States: ACM, 1981, pp. 238–246. DOI: 10.1145/800076.802477.

[Mic13] Z. Micskei. "Languages and frameworks for specifying test artifacts". PhD thesis. Budapest University of Technology and Economics, 2013.

[Mil+14] Á. Milánkovich, G. Ill, K. Lendvai, S. Imre, and S. Szabó. "Evaluation of Energy Efficiency of Aggregation in WSNs using Petri Nets". In: *Proc. of the 3rd Int. Conf. on Sensor Networks*. Science and Technology Publications, 2014, pp. 289–297. ISBN: 978-989-758-001-7. DOI: 10.5220/0004668402890297.

[MS82] J. Martínez and M. Silva. "A simple and Fast Algorithm to Obtain all Invariants of a Generalised Petri Net". In: *Application and Theory of Petri Nets*. Ed. by C. Girault and W. Reisig. Vol. 52. Informatik-Fachberichte. Springer, 1982, pp. 301–310. ISBN: 978-3-540-11189-4. DOI: 10.1007/978-3-642-68353-4_47.

[MSB11] G. J. Myers, C. Sandler, and T. Badgett. *The art of software testing*. John Wiley & Sons, 2011.

[Mur89] T. Murata. "Petri nets: Properties, analysis and applications". In: *Proceedings of the IEEE* 77.4 (Apr. 1989), pp. 541–580. ISSN: 0018-9219. DOI: 10.1109/5.24143.

[NB09] E. Németh and T. Bartha. "Formal Verification of Safety Functions by Reinterpretation of Functional Block Based Specifications". In: *Formal Methods for Industrial Critical Systems*. Ed. by D. Cofer and A. Fantechi. Vol. 5596. Lecture Notes in Computer Science. Springer, 2009, pp. 199–214. ISBN: 978-3-642-03239-4. DOI: 10.1007/978-3-642-03240-0_17.

[Ném+09] E. Németh, T. Bartha, C. Fazekas, and K. M. Hangos. "Verification of a primary-to-secondary leaking safety procedure in a nuclear power plant using coloured Petri nets". In: *Reliability Engineering and System Safety* 94 (5) (2009), pp. 942–953.

[SB00] F. Somenzi and R. Bloem. "Efficient Büchi Automata from LTL Formulae". In: *Computer Aided Verification*. Ed. by E. Emerson and A. Sistla. Vol. 1855. Lecture Notes in Computer Science. Springer, 2000, pp. 248–263. ISBN: 978-3-540-67770-3. DOI: 10.1007/10722167_21.

[SD10] W. Steiner and B. Dutertre. "SMT-based Formal Verification of a TTEthernet Synchronization Function". In: *Proceedings of the 15th International Conference on Formal Methods for Industrial Critical Systems*. FMICS'10. Antwerp, Belgium: Springer, 2010, pp. 148–163. ISBN: 3-642-15897-8, 978-3-642-15897-1.

[STV05] R. Sebastiani, S. Tonetta, and M. Y. Vardi. "Symbolic Systems, Explicit Properties: On Hybrid Approaches for LTL Symbolic Model Checking". In: *Computer Aided Verification*. Ed. by K. Etessami and S. K. Rajamani. Vol. 3576. Lecture Notes in Computer Science. Springer, 2005, pp. 350–363. ISBN: 978-3-540-27231-1.

[Thi15] Y. Thierry-Mieg. "Symbolic Model-Checking Using ITS-Tools". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C. Baier and C. Tinelli. Berlin, Heidelberg: Springer, 2015, pp. 231–237. ISBN: 978-3-662-46681-0. DOI: 10.1007/978-3-662-46681-0_20.

[Tót09] Z. Tóth Heinemann. "Modelling and verification of discrete industrial control systems using formal methods". [In Hungarian]. MA thesis. Budapest University of Technology and Economics, 2009.

[Var01] M. Y. Vardi. "Branching vs. Linear Time: Final Showdown". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by T. Margaria and W. Yi. Berlin, Heidelberg: Springer, 2001, pp. 1–22. ISBN: 978-3-540-45319-2. DOI: 10.1007/3-540-45319-9_1.

[VH10] A. Valmari and H. Hansen. "Can Stubborn Sets Be Optimal?" In: *Applications and Theory of Petri Nets*. Vol. 6128. Lecture Notes in Computer Science. Springer, 2010, pp. 43–62. ISBN: 978-3-642-13674-0.

[Wan+01] C. Wang, R. Bloem, G. D. Hachtel, K. Ravi, and F. Somenzi. "Divide and Compose: SCC Refinement for Language Emptiness". In: *CONCUR 2001 – Concurrency Theory*. Vol. 2154. Lecture Notes in Computer Scienc. Springer, 2001, pp. 456–471. ISBN: 3-540-42497-0.

[WW11] H. Wimmel and K. Wolf. "Applying CEGAR to the Petri Net State Equation". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by P. A. Abdulla and K. R. M. Leino. Vol. 6605. Lecture Notes in Computer Science. Springer, 2011, pp. 224–238. DOI: 10.1007/978-3-642-19835-9_19.

[YCL09] A. Yu, G. Ciardo, and G. Lüttgen. "Decision-diagram-based techniques for bounded reachability checking of asynchronous systems". In: *International Journal on Software Tools for Technology Transfer* 11 (2 2009), pp. 117–131. ISSN: 1433-2779. DOI: 10.1007/s10009-009-0099-0.

[ZC09] Y. Zhao and G. Ciardo. "Symbolic CTL Model Checking of Asynchronous Systems Using Constrained Saturation". In: *Automated Technology for Verification and Analysis*. Vol. 5799. Lecture Notes in Computer Science. Springer, 2009, pp. 368–381. ISBN: 978-3-642-04760-2. DOI: 10.1007/978-3-642-04761-9_27.